

1 The GraphBLAS C API Specification †:

2 Version 2.0.0

3 Benjamin Brock, Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira

4 Generated on 2021/11/15 at 12:03:10 EDT

†Based on *GraphBLAS Mathematics* by Jeremy Kepner

5 Copyright © 2017-2021 Carnegie Mellon University, The Regents of the University of California,
6 through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from
7 the U.S. Dept. of Energy), the Regents of the University of California (U.C. Davis and U.C.
8 Berkeley), Intel Corporation, International Business Machines Corporation, and Massachusetts
9 Institute of Technology Lincoln Laboratory.

10 Any opinions, findings and conclusions or recommendations expressed in this material are those of
11 the author(s) and do not necessarily reflect the views of the United States Department of Defense,
12 the United States Department of Energy, Carnegie Mellon University, the Regents of the University
13 of California, Intel Corporation, or the IBM Corporation.

14 NO WARRANTY. THIS MATERIAL IS FURNISHED ON AN AS-IS BASIS. THE COPYRIGHT
15 OWNERS AND/OR AUTHORS MAKE NO WARRANTIES OF ANY KIND, EITHER EX-
16 PRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WAR-
17 RANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RE-
18 SULTS OBTAINED FROM USE OF THE MATERIAL. THE COPYRIGHT OWNERS AND/OR
19 AUTHORS DO NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREE-
20 DOM FROM PATENT, TRADE MARK, OR COPYRIGHT INFRINGEMENT.

21 Except as otherwise noted, this material is licensed under a Creative Commons Attribution 4.0
22 license (<http://creativecommons.org/licenses/by/4.0/legalcode>), and examples are licensed under
23 the BSD License (<https://opensource.org/licenses/BSD-3-Clause>).

24 Contents

25	List of Tables	9
26	List of Figures	11
27	Acknowledgments	12
28	1 Introduction	13
29	2 Basic Concepts	15
30	2.1 Glossary	15
31	2.1.1 GraphBLAS API basic definitions	15
32	2.1.2 GraphBLAS objects and their structure	16
33	2.1.3 Algebraic structures used in the GraphBLAS	17
34	2.1.4 The execution of an application using the GraphBLAS C API	18
35	2.1.5 GraphBLAS methods: behaviors and error conditions	19
36	2.2 Notation	21
37	2.3 Mathematical Foundations	22
38	2.4 GraphBLAS Opaque Objects	23
39	2.5 Execution Model	24
40	2.5.1 Execution modes	25
41	2.5.2 Multi-threaded execution	26
42	2.6 Error Model	28
43	3 Objects	31
44	3.1 Enumerations for <code>init()</code> and <code>wait()</code>	31
45	3.2 Indices, Index Arrays, and Scalar Arrays	31
46	3.3 Types (Domains)	32

77	4.2.2.4	Scalar_nvals: Number of stored elements in a scalar	65
78	4.2.2.5	Scalar_setElement: Set the single element in a scalar	66
79	4.2.2.6	Scalar_extractElement: Extract a single element from a scalar.	67
80	4.2.3	Vector Methods	68
81	4.2.3.1	Vector_new: Construct new vector	68
82	4.2.3.2	Vector_dup: Construct a copy of a GraphBLAS vector	69
83	4.2.3.3	Vector_resize: Resize a vector	70
84	4.2.3.4	Vector_clear: Clear a vector	71
85	4.2.3.5	Vector_size: Size of a vector	72
86	4.2.3.6	Vector_nvals: Number of stored elements in a vector	73
87	4.2.3.7	Vector_build: Store elements from tuples into a vector	74
88	4.2.3.8	Vector_setElement: Set a single element in a vector	76
89	4.2.3.9	Vector_removeElement: Remove an element from a vector	78
90	4.2.3.10	Vector_extractElement: Extract a single element from a vector.	79
91	4.2.3.11	Vector_extractTuples: Extract tuples from a vector	81
92	4.2.4	Matrix Methods	82
93	4.2.4.1	Matrix_new: Construct new matrix	82
94	4.2.4.2	Matrix_dup: Construct a copy of a GraphBLAS matrix	84
95	4.2.4.3	Matrix_diag: Construct a diagonal GraphBLAS matrix	85
96	4.2.4.4	Matrix_resize: Resize a matrix	86
97	4.2.4.5	Matrix_clear: Clear a matrix	87
98	4.2.4.6	Matrix_nrows: Number of rows in a matrix	88
99	4.2.4.7	Matrix_ncols: Number of columns in a matrix	88
100	4.2.4.8	Matrix_nvals: Number of stored elements in a matrix	89
101	4.2.4.9	Matrix_build: Store elements from tuples into a matrix	90
102	4.2.4.10	Matrix_setElement: Set a single element in matrix	92
103	4.2.4.11	Matrix_removeElement: Remove an element from a matrix	94
104	4.2.4.12	Matrix_extractElement: Extract a single element from a matrix	95
105	4.2.4.13	Matrix_extractTuples: Extract tuples from a matrix	97
106	4.2.4.14	Matrix_exportHint: Provide a hint as to which storage format might be most efficient for exporting a matrix	99
107			

108	4.2.4.15	Matrix_exportSize: Return the array sizes necessary to export a GraphBLAS matrix object	100
109			
110	4.2.4.16	Matrix_export: Export a GraphBLAS matrix to a pre-defined format	101
111			
112	4.2.4.17	Matrix_import: Import a matrix into a GraphBLAS object	103
113	4.2.4.18	Matrix_serializeSize: Compute the serialize buffer size	105
114	4.2.4.19	Matrix_serialize: Serialize a GraphBLAS matrix.	106
115	4.2.4.20	Matrix_deserialize: Deserialize a GraphBLAS matrix.	107
116	4.2.5	Descriptor Methods	108
117	4.2.5.1	Descriptor_new: Create new descriptor	108
118	4.2.5.2	Descriptor_set: Set content of descriptor	109
119	4.2.6	free: Destroy an object and release its resources	110
120	4.2.7	wait: Return once an object is either <i>complete</i> or <i>materialized</i>	112
121	4.2.8	error: Retrieve an error string	113
122	4.3	GraphBLAS Operations	114
123	4.3.1	mxm: Matrix-matrix multiply	118
124	4.3.2	vxm: Vector-matrix multiply	123
125	4.3.3	mxv: Matrix-vector multiply	127
126	4.3.4	eWiseMult: Element-wise multiplication	131
127	4.3.4.1	eWiseMult: Vector variant	132
128	4.3.4.2	eWiseMult: Matrix variant	136
129	4.3.5	eWiseAdd: Element-wise addition	141
130	4.3.5.1	eWiseAdd: Vector variant	142
131	4.3.5.2	eWiseAdd: Matrix variant	146
132	4.3.6	extract: Selecting Sub-Graphs	152
133	4.3.6.1	extract: Standard vector variant	152
134	4.3.6.2	extract: Standard matrix variant	156
135	4.3.6.3	extract: Column (and row) variant	161
136	4.3.7	assign: Modifying Sub-Graphs	166
137	4.3.7.1	assign: Standard vector variant	166
138	4.3.7.2	assign: Standard matrix variant	171

139	4.3.7.3	assign: Column variant	177
140	4.3.7.4	assign: Row variant	182
141	4.3.7.5	assign: Constant vector variant	188
142	4.3.7.6	assign: Constant matrix variant	193
143	4.3.8	apply: Apply a function to the elements of an object	199
144	4.3.8.1	apply: Vector variant	199
145	4.3.8.2	apply: Matrix variant	204
146	4.3.8.3	apply: Vector-BinaryOp variants	208
147	4.3.8.4	apply: Matrix-BinaryOp variants	214
148	4.3.8.5	apply: Vector index unary operator variant	220
149	4.3.8.6	apply: Matrix index unary operator variant	225
150	4.3.9	select:	230
151	4.3.9.1	select: Vector variant	230
152	4.3.9.2	select: Matrix variant	235
153	4.3.10	reduce: Perform a reduction across the elements of an object	241
154	4.3.10.1	reduce: Standard matrix to vector variant	241
155	4.3.10.2	reduce: Vector-scalar variant	245
156	4.3.10.3	reduce: Matrix-scalar variant	249
157	4.3.11	transpose: Transpose rows and columns of a matrix	252
158	4.3.12	kroncker: Kronecker product of two matrices	256
159	5	Nonpolymorphic Interface	263
160	A	Revision History	275
161	B	Non-Opaque Data Format Definitions	279
162	B.1	GrB_Format: Specify the format for input/output of a GraphBLAS matrix.	279
163	B.1.1	GrB_CSR_FORMAT	279
164	B.1.2	GrB_CSC_FORMAT	280
165	B.1.3	GrB_COO_FORMAT	280
166	C	Examples	281

167	C.1 Example: level breadth-first search (BFS) in GraphBLAS	282
168	C.2 Example: level BFS in GraphBLAS using apply	283
169	C.3 Example: parent BFS in GraphBLAS	284
170	C.4 Example: betweenness centrality (BC) in GraphBLAS	285
171	C.5 Example: batched BC in GraphBLAS	287
172	C.6 Example: maximal independent set (MIS) in GraphBLAS	289
173	C.7 Example: counting triangles in GraphBLAS	291

List of Tables

175	2.1	Types of GraphBLAS opaque objects.	23
176	2.2	Methods that forced completion prior to GraphBLAS v2.0.	28
177	3.1	Enumeration literals and corresponding values input to various GraphBLAS methods.	32
178	3.2	Predefined GrB_Type values.	33
179	3.3	Operator input for relevant GraphBLAS operations.	34
180	3.4	Properties and recipes for building GraphBLAS algebraic objects.	35
181	3.5	Predefined unary and binary operators for GraphBLAS in C.	37
182	3.6	Predefined index unary operators for GraphBLAS in C.	38
183	3.7	Predefined monoids for GraphBLAS in C.	40
184	3.8	Predefined “true” semirings for GraphBLAS in C.	41
185	3.9	Other useful predefined semirings for GraphBLAS in C.	42
186	3.10	GrB_Format enumeration literals and corresponding values for matrix import and	
187		export methods.	44
188	3.11	Descriptor types and literals for fields and values.	47
189	3.12	Predefined GraphBLAS descriptors.	48
190	3.13	Enumeration literals and corresponding values returned by GraphBLAS methods	
191		and operations.	49
192	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in	
193		this specification.	115
194	5.1	Long-name, nonpolymorphic form of GraphBLAS methods.	263
195	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	264
196	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	265
197	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	266

198	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	267
199	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	268
200	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	269
201	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	270
202	5.9	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	271
203	5.10	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	272
204	5.11	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	273

205 **List of Figures**

206 3.1 Hierarchy of algebraic object classes in GraphBLAS. 43

207 4.1 Flowchart for the GraphBLAS operations. 116

208 B.1 Data layout for CSR format. 279

209 B.2 Data layout for CSC format. 280

210 B.3 Data layout for COO format. 280

211 Acknowledgments

212 This document represents the work of the people who have served on the C API Subcommittee of
213 the GraphBLAS Forum.

214 Those who served as C API Subcommittee members for GraphBLAS 2.0 are (in alphabetical order):

- 215 • Benjamin Brock (UC Berkeley)
- 216 • Aydın Buluç (Lawrence Berkeley National Laboratory)
- 217 • Timothy G. Mattson (Intel Corporation)
- 218 • Scott McMillan (Software Engineering Institute at Carnegie Mellon University)
- 219 • José Moreira (IBM Corporation)

220 Those who served as C API Subcommittee members for GraphBLAS 1.0 through 1.3 are (in al-
221 phabetical order):

- 222 • Aydın Buluç (Lawrence Berkeley National Laboratory)
- 223 • Timothy G. Mattson (Intel Corporation)
- 224 • Scott McMillan (Software Engineering Institute at Carnegie Mellon University)
- 225 • José Moreira (IBM Corporation)
- 226 • Carl Yang (UC Davis)

227 The GraphBLAS C API Specification is based upon work funded and supported in part by:

- 228 • NSF Graduate Research Fellowship under Grant No. DGE 1752814 and by the NSF under
229 Award No. 1823034 with the University of California, Berkeley
- 230 • The Department of Energy Office of Advanced Scientific Computing Research under contract
231 number DE-AC02-05CH11231
- 232 • Intel Corporation
- 233 • Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon Uni-
234 versity for the operation of the Software Engineering Institute [DM-0003727, DM19-0929,
235 DM21-0090]
- 236 • International Business Machines Corporation

237 The following people provided valuable input and feedback during the development of the specifica-
238 tion (in alphabetical order): David Bader, Hollen Barmer, Bob Cook, Tim Davis, Jeremy Kepner,
239 James Kitchen, Peter Kogge, Manoj Kumar, Roi Lipman, Andrew Mellinger, Maxim Naumov,
240 Nancy M. Ott, Michel Pelletier, Gabor Szarnyas, Ping Tak Peter Tang, Erik Welch, Michael Wolf,
241 Albert-Jan Yzelman.

Chapter 1

Introduction

The GraphBLAS standard defines a set of matrix and vector operations based on semiring algebraic structures. These operations can be used to express a wide range of graph algorithms. This document defines the C binding to the GraphBLAS standard. We refer to this as the *GraphBLAS C API* (Application Programming Interface).

The GraphBLAS C API is built on a collection of objects exposed to the C programmer as opaque data types. Functions that manipulate these objects are referred to as *methods*. These methods fully define the interface to GraphBLAS objects to create or destroy them, modify their contents, and copy the contents of opaque objects into non-opaque objects; the contents of which are under direct control of the programmer.

The GraphBLAS C API is designed to work with C99 (ISO/IEC 9899:199) extended with *static type-based* and *number of parameters-based* function polymorphism, and language extensions on par with the `_Generic` construct from C11 (ISO/IEC 9899:2011). Furthermore, the standard assumes programs using the GraphBLAS C API will execute on hardware that supports floating point arithmetic such as that defined by the IEEE 754 (IEEE 754-2008) standard.

The GraphBLAS C API assumes programs will run on a system that supports acquire-release memory orders. This is needed to support the memory models required for multithreaded execution as described in section 2.5.2.

Implementations of the GraphBLAS C API will target a wide range of platforms. We expect cases will arise where it will be prohibitive for a platform to support a particular type or a specific parameter for a method defined by the GraphBLAS C API. We want to encourage implementors to support the GraphBLAS C API even when such cases arise. Hence, an implementation may still call itself “conformant” as long as the following conditions hold.

- Every method and operation from chapter 4 is supported for the vast majority of cases.
- Any cases not supported must be documented as an implementation-defined feature of the GraphBLAS implementation. Unsupported cases must be caught as an API error (section 2.6) with the parameter `GrB_NOT_IMPLEMENTED` returned by the associated method call.
- It is permissible to omit the corresponding nonpolymorphic methods from chapter 5 when it

271 is not possible to express the signature of that method.

272 The number of allowed omitted cases is vague by design. We cannot anticipate the features of target
273 platforms, on the market today or in the future, that might cause problems for the GraphBLAS
274 specification. It is our expectation, however, that such omitted cases would be a minuscule fraction
275 of the total combination of methods, types, and parameters defined by the GraphBLAS C API
276 specification.

277 The remainder of this document is organized as follows:

- 278 • Chapter 2: Basic Concepts
- 279 • Chapter 3: Objects
- 280 • Chapter 4: Methods
- 281 • Chapter 5: Nonpolymorphic interface
- 282 • Appendix A: Revision history
- 283 • Appendix B: Non-opaque data format definitions
- 284 • Appendix C: Examples

285 Chapter 2

286 Basic concepts

287 The GraphBLAS C API is used to construct graph algorithms expressed “in the language of linear
288 algebra.” Graphs are expressed as matrices, and the operations over these matrices are generalized
289 through the use of a semiring algebraic structure.

290 In this chapter, we will define the basic concepts used to define the GraphBLAS C API. We provide
291 the following elements:

- 292 • Glossary of terms and notation used in this document.
- 293 • Algebraic structures and associated arithmetic foundations of the API.
- 294 • Functions that appear in the GraphBLAS algebraic structures and how they are managed.
- 295 • Domains of elements in the GraphBLAS.
- 296 • Indices, index arrays, scalar arrays, and external matrix formats used to expose the contents
297 of GraphBLAS objects.
- 298 • The GraphBLAS opaque objects.
- 299 • The execution and error models implied by the GraphBLAS C specification.
- 300 • Enumerations used by the API and their values.

301 2.1 Glossary

302 2.1.1 GraphBLAS API basic definitions

- 303 • *application*: A program that calls methods from the GraphBLAS C API to solve a problem.
- 304 • *GraphBLAS C API*: The application programming interface that fully defines the types,
305 objects, literals, and other elements of the C binding to the GraphBLAS.

- 306 • *function*: Refers to a named group of statements in the C programming language. Methods,
307 operators, and user-defined functions are typically implemented as C functions. When refer-
308 ring to the code programmers write, as opposed to the role of functions as an element of the
309 GraphBLAS, they may be referred to as such.
- 310 • *method*: A function defined in the GraphBLAS C API that manipulates GraphBLAS objects
311 or other opaque features of the implementation of the GraphBLAS API.
- 312 • *operator*: A function that performs an operation on the elements stored in GraphBLAS
313 matrices and vectors.
- 314 • *GraphBLAS operation*: A mathematical operation defined in the GraphBLAS mathematical
315 specification. These operations (not to be confused with *operators*) typically act on matrices
316 and vectors with elements defined in terms of an algebraic semiring.

317 2.1.2 GraphBLAS objects and their structure

- 318 • *non-opaque datatype*: Any datatype that exposes its internal structure and can be manipu-
319 lated directly by the user.
- 320 • *opaque datatype*: Any datatype that hides its internal structure and can be manipulated
321 only through an API.
- 322 • *GraphBLAS object*: An instance of an *opaque datatype* defined by the *GraphBLAS C API*
323 that is manipulated only through the GraphBLAS API. There are four kinds of GraphBLAS
324 opaque objects: *domains* (i.e., types), *algebraic objects* (operators, monoids and semirings),
325 *collections* (scalars, vectors, matrices and masks), and descriptors.
- 326 • *handle*: A variable that holds a reference to an instance of one of the GraphBLAS opaque
327 objects. The value of this variable holds a reference to a GraphBLAS object but not the
328 contents of the object itself. Hence, assigning a value to another variable copies the reference
329 to the GraphBLAS object of one handle but not the contents of the object.
- 330 • *domain*: The set of valid values for the elements stored in a GraphBLAS *collection* or
331 operated on by a GraphBLAS *operator*. Note that some GraphBLAS objects involve functions
332 that map values from one or more input domains onto values in an output domain. These
333 GraphBLAS objects would have multiple domains.
- 334 • *collection*: An opaque GraphBLAS object that holds a number of elements from a specified
335 *domain*. Because these objects are based on an opaque datatype, an implementation of the
336 GraphBLAS C API has the flexibility to optimize the data structures for a particular platform.
337 GraphBLAS objects are often implemented as sparse data structures, meaning only the subset
338 of the elements that have values are stored.
- 339 • *implied zero*: Any element that has a valid index (or indices) in a GraphBLAS vector or
340 matrix but is not explicitly identified in the list of elements of that vector or matrix. From a
341 mathematical perspective, an *implied zero* is treated as having the value of the zero element of
342 the relevant monoid or semiring. However, GraphBLAS operations are purposefully defined

343 using set notation in such a way that it makes it unnecessary to reason about implied zeros.
344 Therefore, this concept is not used in the definition of GraphBLAS methods and operators.

345 • *mask*: An internal GraphBLAS object used to control how values are stored in a method's
346 output object. The mask exists only inside a method; hence, it is called an *internal opaque*
347 *object*. A mask is formed from the elements of a collection object (vector or matrix) input as
348 a mask parameter to a method. GraphBLAS allows two types of masks:

349 1. In the default case, an element of the mask exists for each element that exists in the
350 input collection object when the value of that element, when cast to a Boolean type,
351 evaluates to `true`.

352 2. In the *structure only* case, masks have structure but no values. The input collection
353 describes a structure whereby an element of the mask exists for each element stored in
354 the input collection regardless of its value.

355 • *complement*: The *complement* of a GraphBLAS mask, M , is another mask, M' , where the
356 elements of M' are those elements from M that *do not* exist.

357 2.1.3 Algebraic structures used in the GraphBLAS

358 • *associative operator*: In an expression where a binary operator is used two or more times
359 consecutively, that operator is *associative* if the result does not change regardless of the way
360 operations are grouped (without changing their order). In other words, in a sequence of binary
361 operations using the same associative operator, the legal placement of parenthesis does not
362 change the value resulting from the sequence operations. Operators that are associative over
363 infinitely precise numbers (e.g., real numbers) are not strictly associative when applied to
364 numbers with finite precision (e.g., floating point numbers). Such non-associativity results,
365 for example, from roundoff errors or from the fact some numbers can not be represented
366 exactly as floating point numbers. In the GraphBLAS specification, as is common practice
367 in computing, we refer to operators as *associative* when their mathematical definition over
368 infinitely precise numbers is associative even when they are only approximately associative
369 when applied to finite precision numbers.

370 No GraphBLAS method will imply a predefined grouping over any associative operators.
371 Implementations of the GraphBLAS are encouraged to exploit associativity to optimize per-
372 formance of any GraphBLAS method with this requirement. This holds even if the definition
373 of the GraphBLAS method implies a fixed order for the associative operations.

374 • *commutative operator*: In an expression where a binary operator is used (usually two or more
375 times consecutively), that operator is *commutative* if the result does not change regardless of
376 the order the inputs are operated on.

377 No GraphBLAS method will imply a predefined ordering over any commutative operators.
378 Implementations of the GraphBLAS are encouraged to exploit commutativity to optimize per-
379 formance of any GraphBLAS method with this requirement. This holds even if the definition
380 of the GraphBLAS method implies a fixed order for the commutative operations.

- 381 • *GraphBLAS operators*: Binary or unary operators that act on elements of GraphBLAS ob-
382 jects. *GraphBLAS operators* are used to express algebraic structures used in the GraphBLAS
383 such as monoids and semirings. They are also used as arguments to several GraphBLAS meth-
384 ods. There are two types of *GraphBLAS operators*: (1) predefined operators found in Table 3.5
385 and (2) user-defined operators created using `GrB_UnaryOp_new()` or `GrB_BinaryOp_new()`
386 (see Section 4.2.1).
- 387 • *monoid*: An algebraic structure consisting of one domain, an associative binary operator,
388 and the identity of that operator. There are two types of GraphBLAS monoids: (1) predefined
389 monoids found in Table 3.7 and (2) user-defined monoids created using `GrB_Monoid_new()`
390 (see Section 4.2.1).
- 391 • *semiring*: An algebraic structure consisting of a set of allowed values (the *domain*), a
392 commutative and associative binary operator called addition, a binary operator called mul-
393 tiplication (where multiplication distributes over addition), and identities over addition (0)
394 and multiplication (1). The additive identity is an annihilator over multiplication.
- 395 • *GraphBLAS semiring*: is allowed to diverge from the mathematically rigorous definition of
396 a *semiring* since certain combinations of domains, operators, and identity elements are useful
397 in graph algorithms even when they do not strictly match the mathematical definition of a
398 semiring. There are two types of *GraphBLAS semirings*: (1) predefined semirings found in
399 Tables 3.8 and 3.9, and (2) user-defined semirings created using `GrB_Semiring_new()` (see
400 Section 4.2.1).
- 401 • *index unary operator*: A variation of the unary operator that operates on elements of
402 GraphBLAS vectors and matrices along with the index values representing their location in
403 the objects. There are predefined index unary operators found in Table 3.6), and user-defined
404 operators created using `GrB_IndexUnaryOp_new` (see Section 4.2.1).

405 2.1.4 The execution of an application using the GraphBLAS C API

- 406 • *program order*: The order of the GraphBLAS method calls in a thread, as defined by the
407 text of the program.
- 408 • *host programming environment*: The GraphBLAS specification defines an API. The functions
409 from the API appear in a program. This program is written using a programming language
410 and execution environment defined outside of the GraphBLAS. We refer to this programming
411 environment as the “host programming environment”.
- 412 • *execution time*: time expended while executing instructions defined by a program. This
413 term is specifically used in this specification in the context of computations carried out on
414 behalf of a call to a GraphBLAS method.
- 415 • *sequence*: A GraphBLAS application uniquely defines a directed acyclic graph (DAG) of
416 GraphBLAS method calls based on their program order. At any point in a program, the
417 state of any GraphBLAS object is defined by a subgraph of that DAG. An ordered collection
418 of GraphBLAS method calls in program order that defines that subgraph for a particular
419 object is the *sequence* for that object.

- 420 • *complete*: A GraphBLAS object is complete when it can be used in a happens-before relation-
421 ship with a method call that reads the variable on another thread. This concept is used when
422 reasoning about memory orders in multithreaded programs. A GraphBLAS object defined on
423 one thread that is complete can be safely used as an IN or INOUT argument in a method-call
424 on a second thread assuming the method calls are correctly synchronized so the definition on
425 the first thread *happens-before* it is used on the second thread. In blocking-mode, an object is
426 complete after a GraphBLAS method call that writes to that object returns. In nonblocking-
427 mode, an object is complete after a call to the `GrB_wait()` method with the `GrB_COMPLETE`
428 parameter.

- 429 • *materialize*: A GraphBLAS object is materialized when it is (1) complete, (2) the compu-
430 tations defined by the sequence that define the object have finished (either fully or stopped
431 at an error) and will not consume any additional computational resources, and (3) any errors
432 associated with that sequence are available to be read according to the GraphBLAS error
433 model. A GraphBLAS object that is never loaded into a non-opaque data structure may
434 potentially never be materialized. This might happen, for example, if the operations associ-
435 ated with the object are fused or otherwise changed by the runtime system that supports the
436 implementation of the GraphBLAS C API. An object can be materialized by a call to the
437 `materialize` mode of the `GrB_wait()` method.

- 438 • *context*: An instance of the GraphBLAS C API implementation as seen by an application.
439 An application can have only one context between the start and end of the application. A
440 context begins with the first thread that calls `GrB_init()` and ends with the first thread to
441 call `GrB_finalize()`. It is an error for `GrB_init()` or `GrB_finalize()` to be called more than one
442 time within an application. The context is used to constrain the behavior of an instance of
443 the GraphBLAS C API implementation and support various execution strategies. Currently,
444 the only supported constraints on a context pertain to the mode of program execution.

- 445 • *program execution mode*: Defines how a GraphBLAS sequence executes, and is associated
446 with the *context* of a GraphBLAS C API implementation. It is set by an application with
447 its call to `GrB_init()` to one of two possible states. In *blocking mode*, GraphBLAS methods
448 return after the computations complete and any output objects have been materialized. In
449 *nonblocking mode*, a method may return once the arguments are tested as consistent with
450 the method (i.e., there are no API errors), and potentially before any computation has taken
451 place.

452 2.1.5 GraphBLAS methods: behaviors and error conditions

- 453 • *implementation-defined behavior*: Behavior that must be documented by the implementation
454 and is allowed to vary among different compliant implementations.

- 455 • *undefined behavior*: Behavior that is not specified by the GraphBLAS C API. A conforming
456 implementation is free to choose results delivered from a method whose behavior is undefined.

- 457 • *thread-safe*: Consider a function called from multiple threads with arguments that do not
458 overlap in memory (i.e. the argument lists do not share memory). If the function is *thread-safe*

459 then it will behave the same when executed concurrently by multiple threads or sequentially
460 on a single thread.

461 • *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as param-
462 eters to a GraphBLAS method are dimension (or shape) compatible if they have the correct
463 number of dimensions and sizes for each dimension to satisfy the rules of the mathematical def-
464 inition of the operation associated with the method. If any *dimension compatibility* rule above
465 is violated, execution of the GraphBLAS method ends and the GrB_DIMENSION_MISMATCH
466 error is returned.

467 • *domain compatible*: Two domains for which values from one domain can be cast to values in
468 the other domain as per the rules of the C language. In particular, domains from Table 3.2
469 are all compatible with each other, and a domain from a user-defined type is only compatible
470 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS
471 method ends and the GrB_DOMAIN_MISMATCH error is returned.

Notation	Description
$D_{out}, D_{in}, D_{in_1}, D_{in_2}$	Refers to output and input domains of various GraphBLAS operators.
$\mathbf{D}_{out}(*), \mathbf{D}_{in}(*),$ $\mathbf{D}_{in_1}(*), \mathbf{D}_{in_2}(*)$	Evaluates to output and input domains of GraphBLAS operators (usually a unary or binary operator, or semiring).
$\mathbf{D}(*)$	Evaluates to the (only) domain of a GraphBLAS object (usually a monoid, vector, or matrix).
f	An arbitrary unary function, usually a component of a unary operator.
$\mathbf{f}(F_u)$	Evaluates to the unary function contained in the unary operator given as the argument.
\odot	An arbitrary binary function, usually a component of a binary operator.
$\odot(*)$	Evaluates to the binary function contained in the binary operator or monoid given as the argument.
\otimes	Multiplicative binary operator of a semiring.
\oplus	Additive binary operator of a semiring.
$\otimes(S)$	Evaluates to the multiplicative binary operator of the semiring given as the argument.
$\oplus(S)$	Evaluates to the additive binary operator of the semiring given as the argument.
$\mathbf{0}(*)$	The identity of a monoid, or the additive identity of a GraphBLAS semiring.
$\mathbf{L}(*)$	The contents (all stored values) of the vector or matrix GraphBLAS objects. For a vector, it is the set of (index, value) pairs, and for a matrix it is the set of (row, col, value) triples.
473 $\mathbf{v}(i)$ or v_i	The i^{th} element of the vector \mathbf{v} .
$\mathbf{size}(\mathbf{v})$	The size of the vector \mathbf{v} .
$\mathbf{ind}(\mathbf{v})$	The set of indices corresponding to the stored values of the vector \mathbf{v} .
$\mathbf{nrows}(\mathbf{A})$	The number of rows in the \mathbf{A} .
$\mathbf{ncols}(\mathbf{A})$	The number of columns in the \mathbf{A} .
$\mathbf{indrow}(\mathbf{A})$	The set of row indices corresponding to rows in \mathbf{A} that have stored values.
$\mathbf{indcol}(\mathbf{A})$	The set of column indices corresponding to columns in \mathbf{A} that have stored values.
$\mathbf{ind}(\mathbf{A})$	The set of (i, j) indices corresponding to the stored values of the matrix.
$\mathbf{A}(i, j)$ or A_{ij}	The element of \mathbf{A} with row index i and column index j .
$\mathbf{A}(:, j)$	The j^{th} column of matrix \mathbf{A} .
$\mathbf{A}(i, :)$	The i^{th} row of matrix \mathbf{A} .
\mathbf{A}^T	The transpose of matrix \mathbf{A} .
$\neg\mathbf{M}$	The complement of \mathbf{M} .
$\mathbf{s}(\mathbf{M})$	The structure of \mathbf{M} .
$\tilde{\mathbf{t}}$	A temporary object created by the GraphBLAS implementation.
$\langle type \rangle$	A method argument type that is <code>void *</code> or one of the types from Table 3.2.
<code>GrB_ALL</code>	A method argument literal to indicate that all indices of an input array should be used.
<code>GrB_Type</code>	A method argument type that is either a user defined type or one of the types from Table 3.2.
<code>GrB_Object</code>	A method argument type referencing any of the GraphBLAS object types.
<code>GrB_NULL</code>	The GraphBLAS NULL.

474 2.3 Mathematical foundations

475 Graphs can be represented in terms of matrices. The values stored in these matrices correspond to
476 attributes (often weights) of edges in the graph.¹ Likewise, information about vertices in a graph
477 are stored in vectors. The set of valid values that can be stored in either matrices or vectors is
478 referred to as their domain. Matrices are usually sparse because the lack of an edge between two
479 vertices means that nothing is stored at the corresponding location in the matrix. Vectors may be
480 sparse or dense, or they may start out sparse and become dense as algorithms traverse the graphs.

481 Operations defined by the GraphBLAS C API specification operate on these matrices and vectors
482 to carry out graph algorithms. These GraphBLAS operations are defined in terms of GraphBLAS
483 semiring algebraic structures. Modifying the underlying semiring changes the result of an operation
484 to support a wide range of graph algorithms. Inside a given algorithm, it is often beneficial to change
485 the GraphBLAS semiring that applies to an operation on a matrix. This has two implications for
486 the C binding of the GraphBLAS API.

487 First, it means that we define a separate object for the semiring to pass into methods. Since in many
488 cases the full semiring is not required, we also support passing monoids or even binary operators,
489 which means the semiring is implied rather than explicitly stated.

490 Second, the ability to change semirings impacts the meaning of the *implied zero* in a sparse rep-
491 resentation of a matrix or vector. This element in real arithmetic is zero, which is the identity of
492 the *addition* operator and the annihilator of the *multiplication* operator. As the semiring changes,
493 this implied zero changes to the identity of the *addition* operator and the annihilator (if present)
494 of the *multiplication* operator for the new semiring. Nothing changes regarding what is stored in
495 the sparse matrix or vector, but the implied zeros within them change with respect to a particular
496 operation. In all cases, the nature of the implied zero does not matter since the GraphBLAS C
497 API requires that implementations treat them as nonexistent elements of the matrix or vector.

498 As with matrices and vectors, GraphBLAS semirings have domains associated with their inputs and
499 outputs. The semirings in the GraphBLAS C API are defined with two domains associated with
500 the input operands and one domain associated with output. When used in the GraphBLAS C API
501 these domains may not match the domains of the matrices and vectors supplied in the operations.
502 In this case, only valid *domain compatible* casting is supported by the API.

503 The mathematical formalism for graph operations in the language of linear algebra often assumes
504 that we can operate in the field of real numbers. However, the GraphBLAS C binding is designed for
505 implementation on computers, which by necessity have a finite number of bits to represent numbers.
506 Therefore, we require a conforming implementation to use floating point numbers such as those
507 defined by the IEEE-754 standard (both single- and double-precision) wherever real numbers need
508 to be represented. The practical implications of these finite precision numbers is that the result of
509 a sequence of computations may vary from one execution to the next as the grouping of operands
510 (because of associativity) within the operations changes. While techniques are known to reduce
511 these effects, we do not require or even expect an implementation to use them as they may add

¹More information on the mathematical foundations can be found in the following paper: J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. Owens, C. Yang, M. Zalewski, and T. Mattson. 2016, September. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)* (pp. 1-9). IEEE.

Table 2.1: Types of GraphBLAS opaque objects.

GrB_Object types	Description
GrB_Type	Scalar type.
GrB_UnaryOp	Unary operator.
GrB_IndexUnaryOp	Unary operator, that operates on a single value and its location index values.
GrB_BinaryOp	Binary operator.
GrB_Monoid	Monoid algebraic structure.
GrB_Semiring	A GraphBLAS semiring algebraic structure.
GrB_Scalar	One element; could be empty.
GrB_Vector	One-dimensional collection of elements; can be sparse.
GrB_Matrix	Two-dimensional collection of elements; typically sparse.
GrB_Descriptor	Descriptor object, used to modify behavior of methods (specifically GraphBLAS operations).

512 considerable overhead. In most cases, these roundoff errors are not significant. When they are
 513 significant, the problem itself is ill-conditioned and needs to be reformulated.

514 2.4 GraphBLAS opaque objects

515 Objects defined in the GraphBLAS standard include types (the domains of elements), collections
 516 of elements (matrices, vectors, and scalars), operators on those elements (unary, index unary, and
 517 binary operators), algebraic structures (semirings and monoids), and descriptors. GraphBLAS
 518 objects are defined as opaque types; that is, they are managed, manipulated, and accessed solely
 519 through the GraphBLAS application programming interface. This gives an implementation of the
 520 GraphBLAS C specification flexibility to optimize objects for different scenarios or to meet the
 521 needs of different hardware platforms.

522 A GraphBLAS opaque object is accessed through its *handle*. A handle is a variable that references
 523 an instance of one of the types from Table 2.1. An implementation of the GraphBLAS specification
 524 has a great deal of flexibility in how these handles are implemented. All that is required is that the
 525 handle corresponds to a type defined in the C language that supports assignment and comparison
 526 for equality. The GraphBLAS specification defines a literal `GrB_INVALID_HANDLE` that is valid
 527 for each type. Using the logical equality operator from C, it must be possible to compare a handle
 528 to `GrB_INVALID_HANDLE` to verify that a handle is valid.

529 Every GraphBLAS object has a *lifetime*, which consists of the sequence of instructions executed
 530 in program order between the *creation* and the *destruction* of the object. The GraphBLAS C API
 531 predefines a number of these objects which are created when the GraphBLAS context is initialized
 532 by a call to `GrB_init` and are destroyed when the GraphBLAS context is terminated by a call to
 533 `GrB_finalize`.

534 An application using the GraphBLAS API can create additional objects by declaring variables of the
 535 appropriate type from Table 2.1 for the objects it will use. Before use, the object must be initialized

536 with a call call to one of the object’s respective *constructor* methods. Each kind of object has at
537 least one explicit constructor method of the form `GrB*_new` where ‘*’ is replaced with the type
538 of object (e.g., `GrB_Semiring_new`). Note that some objects, especially collections, have additional
539 constructor methods such as duplication, import, or deserialization. Objects explicitly created by
540 a call to a constructor should be destroyed by a call to `GrB_free`. The behavior of a program that
541 calls `GrB_free` on a pre-defined object is undefined.

542 These constructor and destructor methods are the only methods that change the value of a handle.
543 Hence, objects changed by these methods are passed into the method as pointers. In all other
544 cases, handles are not changed by the method and are passed by value. For example, even when
545 multiplying matrices, while the contents of the output product matrix changes, the handle for that
546 matrix is unchanged.

547 Several GraphBLAS constructor methods take other objects as input arguments and use these
548 objects to create a new object. For all these methods, the lifetime of the created object must
549 end strictly before the lifetime of any dependent input objects. For example, a vector constructor
550 `GrB_Vector_new` takes a `GrB_Type` object as input. That type object must not be destroyed until
551 after the created vector is destroyed. Similarly, a `GrB_Semiring_new` method takes a monoid and
552 a binary operator as inputs. Neither of these can be destroyed until after the created semiring is
553 destroyed.

554 Note that some constructor methods like `GrB_Vector_dup` and `GrB_Matrix_dup` behave differently.
555 In these cases, the input vector or matrix can be destroyed as soon as the call returns. However,
556 the original type object used to create the input vector or matrix cannot be destroyed until after
557 the vector or matrix created by `GrB_Vector_dup` or `GrB_Matrix_dup` is destroyed. This behavior
558 must hold for any chain of duplicating constructors.

559 Programmers using GraphBLAS handles must be careful to distinguish between a handle and the
560 object manipulated through a handle. For example, a program may declare two GraphBLAS objects
561 of the same type, initialize one, and then assign it to the other variable. That assignment, however,
562 only assigns the handle to the variable. It does not create a copy of that variable (to do that,
563 one would need to use the appropriate duplication method). If later the object is freed by calling
564 `GrB_free` with the first variable, the object is destroyed and the second variable is left referencing
565 an object that no longer exists (a so-called “dangling handle”).

566 In addition to opaque objects manipulated through handles, the GraphBLAS C API defines an
567 additional opaque object as an internal object; that is, the object is never exposed as a variable
568 within an application. This opaque object is the mask used to control which computed values can
569 be stored in the output operand of a *GraphBLAS operation*. Masks are described in Section 3.5.4.

570 2.5 Execution model

571 A program using the GraphBLAS C API is called a GraphBLAS application. The application con-
572 structs GraphBLAS objects, manipulates them to implement a graph algorithm, and then extracts
573 values from the GraphBLAS objects to produce the results for that algorithm. Functions defined
574 within the GraphBLAS C API that manipulate GraphBLAS objects are called *methods*. If the
575 method corresponds to one of the operations defined in the GraphBLAS mathematical specifica-

576 tion, we refer to the method as an *operation*.

577 The GraphBLAS application specifies an ordered collection of GraphBLAS method calls defined
578 by the order they appear in the text of the program (the *program order*). These define a directed
579 acyclic graph (DAG) where nodes are GraphBLAS method calls and edges are dependencies between
580 method calls.

581 Each method call in the DAG uniquely and unambiguously defines the output GraphBLAS objects
582 as long as there are no execution errors that put objects in an invalid state (see Section 2.6). An
583 ordered collection of method calls, a subgraph of the overall DAG for an application, defines the
584 state of a GraphBLAS object at any point in a program. This ordered collection is the *sequence*
585 for that object.

586 Since the GraphBLAS execution is defined in terms of a DAG and the GraphBLAS objects are
587 opaque, the semantics of the GraphBLAS specification affords an implementation considerable
588 flexibility to optimize performance. A GraphBLAS implementation can defer execution of nodes in
589 the DAG, fuse nodes, or even replace whole subgraphs within the DAG to optimize performance.
590 We discuss this topic further in section 2.5.1 when we describe *blocking* and *non-blocking* execution
591 modes.

592 A correct GraphBLAS application must be *race-free*. This means that the DAG produced by an
593 application and the results produced by execution of that DAG must be the same regardless of
594 how the threads are scheduled for execution. It is the application programmer's responsibility to
595 control memory orders and establish the required synchronized-with relationships to assure race-free
596 execution of a multi-threaded GraphBLAS application. Writing race-free GraphBLAS applications
597 is discussed further in Section 2.5.2.

598 2.5.1 Execution modes

599 The execution of the DAG defined by a GraphBLAS application depends on the *execution mode* of
600 the GraphBLAS program. There are two modes: *blocking* and *nonblocking*.

- 601 • *blocking*: In blocking mode, each method finishes the GraphBLAS operation defined by the
602 method and all output GraphBLAS objects are *materialized* before proceeding to the next
603 statement. Even mechanisms that break the opaqueness of the GraphBLAS objects (e.g.,
604 performance monitors, debuggers, memory dumps) will observe that the operation has fin-
605 ished.
- 606 • *nonblocking*: In nonblocking mode, each method may return once the input arguments have
607 been inspected and verified to define a well formed GraphBLAS operation. (That is, there
608 are no API errors; see Section 2.6.) The GraphBLAS method may not have finished, but the
609 output object is ready to be used by the next GraphBLAS method call. If needed, a call to
610 GrB_wait with GrB_COMPLETE or GrB_MATERIALIZE can be used to force the sequence
611 for a GraphBLAS object (obj) to finish its execution.

612 The *execution mode* is defined in the GraphBLAS C API when the context of the library invoca-
613 tion is defined. This occurs once before any GraphBLAS methods are called with a call to the

614 GrB_init() function. This function takes a single argument of type GrB_Mode with values shown
615 in Table 3.1(a).

616 An application executing in nonblocking mode is not required to return immediately after input
617 arguments have been verified. A conforming implementation of the GraphBLAS C API running
618 in nonblocking mode may choose to execute *as if* in blocking mode. A sequence of operations
619 in nonblocking mode where every GraphBLAS operation with output object obj is followed by a
620 GrB_wait(obj, GrB_MATERIALIZE) call is equivalent to the same sequence in blocking mode with
621 GrB_wait(obj, GrB_MATERIALIZE) calls removed.

622 Nonblocking mode allows for any execution strategy that satisfies the mathematical definition of
623 the sequence. The methods can be placed into a queue and deferred. They can be chained together
624 and fused (e.g., replacing a chained pair of matrix products with a matrix triple product). Lazy
625 evaluation, greedy evaluation, and asynchronous execution are all valid as long as the final result
626 agrees with the mathematical definition provided by the sequence of GraphBLAS method calls
627 appearing in program order.

628 Blocking mode forces an implementation to carry out precisely the GraphBLAS operations defined
629 by the methods and to complete each and every method call individually. It is valuable for debug-
630 ging or in cases where an external tool such as a debugger needs to evaluate the state of memory
631 during a sequence of operations.

632 In a sequence of operations free of execution errors, and with input objects that are well-conditioned,
633 the results from blocking and nonblocking modes should be identical outside of effects due to
634 roundoff errors associated with floating point arithmetic. Due to the great flexibility afforded to an
635 implementation when using nonblocking mode, we expect execution of a sequence in nonblocking
636 mode to potentially complete execution in less time.

637 It is important to note that, processing of nonopaque objects is never deferred in GraphBLAS.
638 That is, methods that consume nonopaque objects (e.g., GrB_Matrix_build(), Section 4.2.4.9) and
639 methods that produce nonopaque objects (e.g., GrB_Matrix_extractTuples(), Section 4.2.4.13) al-
640 ways finish consuming or producing those nonopaque objects before returning regardless of the
641 execution mode.

642 Finally, after all GraphBLAS method calls have been made, the context is terminated with a call
643 to GrB_finalize(). In the current version of the GraphBLAS C API, the context can be set only
644 once in the execution of a program. That is, after GrB_finalize() is called, a subsequent call to
645 GrB_init() is not allowed.

646 2.5.2 Multi-threaded execution

647 The GraphBLAS C API is designed to work with applications that utilize multiple threads executing
648 within a shared address space. This specification does not define how threads are created, managed
649 and synchronized. We expect the host programming environment to provide those services.

650 A conformant implementation of the GraphBLAS must be *thread safe*. A GraphBLAS library
651 is thread safe when independent method calls (i.e., GraphBLAS objects are not shared between
652 method calls) from multiple threads in a race-free program return the same results as would follow

653 from their sequential execution in some interleaved order. This is a common requirement in software
654 libraries.

655 Thread safety applies to the behavior of multiple independent threads. In the more general case
656 for multithreading, threads are not independent; they share variables and mix read and write
657 operations to those variables across threads. A memory consistency model defines which values
658 can be returned when reading an object shared between two or more threads. The GraphBLAS
659 specification does not define its own memory consistency model. Instead the specification defines
660 what must be done by a programmer calling GraphBLAS methods and by the implementor of a
661 GraphBLAS library so an implementation of the GraphBLAS specification can work correctly with
662 the memory consistency model for the host environment.

663 A memory consistency model is defined in terms of happens-before relations between methods in
664 different threads. The defining case is a method that writes to an object on one thread that is
665 read (i.e., used as an IN or INOUT argument) in a GraphBLAS method on a different thread. The
666 following steps must occur between the different threads.

- 667 • A sequence of GraphBLAS methods results in the definition of the GraphBLAS object.
- 668 • The GraphBLAS object is put into a state of completion by a call to `GrB_wait()` with the
669 `GrB_COMPLETE` parameter (see Table 3.1(b)). A GraphBLAS object is said to be *complete*
670 when it can be safely used as an IN or INOUT argument in a GraphBLAS method call from
671 a different thread.
- 672 • Completion happens before a synchronized-with relation that executes with *at least* a release
673 memory order.
- 674 • A synchronized-with relation on the other thread executes with *at least* an acquire memory
675 order.
- 676 • This synchronized-with relation happens-before the GraphBLAS method that reads the graph-
677 BLAS object.

678 We use the phrase *at least* when talking about the memory orders to indicate that a stronger
679 memory order such as *sequential consistency* can be used in place of the acquire-release order.

680 A program that violates these rules contains a data race. That is, its reads and writes are unordered
681 across threads making the final value of a variable undefined. A program that contains a data race
682 is invalid and the results of that program are undefined. We note that multi-threaded execution is
683 compatible with both blocking and non-blocking modes of execution.

684 Completion is the central concept that allows GraphBLAS objects to be used in happens-before
685 relations between threads. In earlier versions of GraphBLAS (1.X) completion was implied by
686 any operation that produced non-opaque values from a GraphBLAS object. These operations are
687 summarized in Table 2.2). In GraphBLAS 2.0, these methods no longer imply completion. This
688 change was made since there are cases where the non-opaque value is needed but the object from
689 which it is computed is not. We want implementations of the GraphBLAS to be able to exploit
690 this case and not form the opaque object when that object is not needed.

Table 2.2: Methods that extract values from a GraphBLAS object that forcing completion of the operations contributing to that particular object in GraphBLAS 1.X. In GraphBLAS 2.0, these methods *do not* force completion.

Method	Section
GrB_Vector_nvals	4.2.3.6
GrB_Vector_extractElement	4.2.3.10
GrB_Vector_extractTuples	4.2.3.11
GrB_Matrix_nvals	4.2.4.8
GrB_Matrix_extractElement	4.2.4.12
GrB_Matrix_extractTuples	4.2.4.13
GrB_reduce (vector-scalar value variant)	4.3.10.2
GrB_reduce (matrix-scalar value variant)	4.3.10.3

2.6 Error model

All GraphBLAS methods return a value of type GrB_Info (an enum) to provide information available to the system at the time the method returns. The returned value will be one of the defined values shown in Table 3.13. The return values fall into three groups: informational, API errors, and execution errors. While API and execution errors take on negative values, informational return values listed in Table 3.13(a) are non-negative and include GrB_SUCCESS (a value of 0) and GrB_NO_VALUE.

An API error (listed in Table 3.13(b)) means that a GraphBLAS method was called with parameters that violate the rules for that method. These errors are restricted to those that can be determined by inspecting the dimensions and domains of GraphBLAS objects, GraphBLAS operators, or the values of scalar parameters fixed at the time a method is called. API errors are deterministic and consistent across platforms and implementations. API errors are never deferred, even in nonblocking mode. That is, if a method is called in a manner that would generate an API error, it always returns with the appropriate API error value. If a GraphBLAS method returns with an API error, it is guaranteed that none of the arguments to the method (or any other program data) have been modified. The informational return value, GrB_NO_VALUE, is also deterministic and never deferred in nonblocking mode.

Execution errors (listed in Table 3.13(c)) indicate that something went wrong during the execution of a legal GraphBLAS method invocation. Their occurrence may depend on specifics of the execution environment and data values being manipulated. This does not mean that execution errors are the fault of the GraphBLAS implementation. For example, a memory leak could arise from an error in an application’s source code (a “program error”), but it may manifest itself in different points of a program’s execution (or not at all) depending on the platform, problem size, or what else is running at that time. Index out-of-bounds errors, for example, always indicate a program error.

If a GraphBLAS method returns with any execution error other than GrB_PANIC, it is guaranteed that the state of any argument used as input-only is unmodified. Output arguments may be left in an invalid state, and their use downstream in the program flow may cause additional errors. If a

719 GraphBLAS method returns with a `GrB_PANIC` execution error, no guarantees can be made about
720 the state of any program data.

721 In nonblocking mode, execution errors can be deferred. A return value of `GrB_SUCCESS` only
722 guarantees that there are no API errors in the method invocation. If an execution error value is
723 returned by a method with output object `obj` in nonblocking mode, it indicates that an error was
724 found during execution of any of the pending operations on `obj`, up to and including the `GrB_wait()`
725 method (Section 4.2.7) call that completes those pending operations. When possible, that return
726 value will provide information concerning the cause of the error.

727 As discussed in Section 4.2.7, a `GrB_wait(obj)` on a specific GraphBLAS object `obj` completes all
728 pending operations on that object. No additional errors on the methods that precede the call to
729 `GrB_wait` and have `obj` as an `OUT` or `INOUT` argument can be reported. From a GraphBLAS
730 perspective, those methods are *complete*. Details on the guaranteed state of objects after a call to
731 `GrB_wait` can be found in Section 4.2.7.

732 After a call to any GraphBLAS method that modifies an opaque object, the program can re-
733 trieve additional error information (beyond the error code returned by the method) though a call
734 to the function `GrB_error()`, passing the method's output object as described in Section 4.2.8.
735 The function returns a pointer to a NULL-terminated string, and the contents of that string are
736 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error
737 string. `GrB_error()` is a thread-safe function, in the sense that multiple threads can call it simul-
738 taneously and each will get its own error string back, referring to the object passed as an input
739 argument.

740 Chapter 3

741 Objects

742 In this chapter, all of the enumerations, literals, data types, and predefined opaque objects defined
743 in the GraphBLAS API are presented. Enumeration literals in GraphBLAS are assigned specific
744 values to ensure compatibility between different runtime library implementations. The chapter
745 starts by defining the enumerations that are used by the `init()` and `wait()` methods. Then a num-
746 ber of transparent (i.e., non-opaque) types that are used for interfacing with external data are
747 defined. Sections that follow describe the various types of opaque objects in GraphBLAS: types
748 (or *domains*), algebraic objects, collections and descriptors. Each of these sections also lists the
749 predefined instances of each opaque type that are required by the API. This chapter concludes with
750 a section on the definition for `GrB_Info` enumeration that is used as the return type of all methods.

751 3.1 Enumerations for `init()` and `wait()`

752 Table 3.1 lists the enumerations and the corresponding values used in the `GrB_init()` method to set
753 the execution mode and in the `GrB_wait()` method for completing or materializing opaque objects.

754 3.2 Indices, index arrays, and scalar arrays

755 In order to interface with third-party software (i.e., software other than an implementation of the
756 GraphBLAS), operations such as `GrB_Matrix_build` (Section 4.2.4.9) and `GrB_Matrix_extractTuples`
757 (Section 4.2.4.13) must specify how the data should be laid out in non-opaque data structures. To
758 this end we explicitly define the types for indices and the arrays used by these operations.

759 For indices a `typedef` is used to give a GraphBLAS name to a concrete type. We define it as follows:

```
760     typedef uint64_t GrB_Index;
```

761 The range of valid values for a variable of type `GrB_Index` is `[0, GrB_INDEX_MAX]` where the
762 largest index value permissible is defined with a macro, `GrB_INDEX_MAX`. For example:

763 `#define GrB_INDEX_MAX ((GrB_Index) 0xffffffffffffffff);`

764 An implementation is required to define and document this value.

765 An index array is a pointer to a set of `GrB_Index` values that are stored in a contiguous block of
766 memory (i.e., `GrB_Index*`). Likewise, a scalar array is a pointer to a contiguous block of memory
767 storing a number of scalar values as specified by the user. Some GraphBLAS operations (e.g.,
768 `GrB_assign`) include an input parameter with the type of an index array. This input index array
769 selects a subset of elements from a GraphBLAS vector or matrix object to be used in the operation.
770 In these cases, the literal `GrB_ALL` can be used in place of the index array input parameter to
771 indicate that all indices of the associated GraphBLAS vector or matrix object should be used. An
772 implementation of the GraphBLAS C API has considerable freedom in terms of how `GrB_ALL`
773 is defined. Since `GrB_ALL` is used as an argument for an array parameter, it must use a type
774 consistent with a pointer. `GrB_ALL` must also have a non-null value to distinguish it from the
775 erroneous case of passing a `NULL` pointer as an array.

776 3.3 Types (domains)

777 In GraphBLAS, domains correspond to the valid values for types from the host language (in our
778 case, the C programming language). GraphBLAS defines a number of operators that take elements
779 from one or more domains and produce elements of a (possibly) different domain. GraphBLAS
780 also defines three kinds of collections: matrices, vectors and scalars. For any given collection, the
781 elements of the collection belong to a *domain*, which is the set of valid values for the elements. For
782 any variable or object V in GraphBLAS we denote as $\mathbf{D}(V)$ the domain of V , that is, the set of
783 possible values that elements of V can take.

Table 3.1: Enumeration literals and corresponding values input to various GraphBLAS methods.

(a) `GrB_Mode` execution modes for the `GrB_init` method.

Symbol	Value	Description
<code>GrB_NONBLOCKING</code>	0	Specifies the nonblocking mode context.
<code>GrB_BLOCKING</code>	1	Specifies the blocking mode context.

(b) `GrB_WaitMode` wait modes for the `GrB_wait` method.

Symbol	Value	Description
<code>GrB_COMPLETE</code>	0	The object is in a state where it can be used in a happens-before relation so that multithreaded programs can be properly synchronized.
<code>GrB_MATERIALIZE</code>	1	The object is <i>complete</i> , and in addition, all computation of the object is finished and any error information is available.

Table 3.2: Predefined GrB_Type values, and the corresponding GraphBLAS domain suffixes, C type (for scalar parameters), and domains for GraphBLAS. The domain suffixes are used in place of I , F , and T in Tables 3.5, 3.6, 3.7, 3.8, and 3.9).

GrB_Type	Suffix	C type	Domain
GrB_BOOL	BOOL	bool	{false, true}
GrB_INT8	INT8	int8_t	$\mathbb{Z} \cap [-2^7, 2^7)$
GrB_UINT8	UINT8	uint8_t	$\mathbb{Z} \cap [0, 2^8)$
GrB_INT16	INT16	int16_t	$\mathbb{Z} \cap [-2^{15}, 2^{15})$
GrB_UINT16	UINT16	uint16_t	$\mathbb{Z} \cap [0, 2^{16})$
GrB_INT32	INT32	int32_t	$\mathbb{Z} \cap [-2^{31}, 2^{31})$
GrB_UINT32	UINT32	uint32_t	$\mathbb{Z} \cap [0, 2^{32})$
GrB_INT64	INT64	int64_t	$\mathbb{Z} \cap [-2^{63}, 2^{63})$
GrB_UINT64	UINT64	uint64_t	$\mathbb{Z} \cap [0, 2^{64})$
GrB_FP32	FP32	float	IEEE 754 binary32
GrB_FP64	FP64	double	IEEE 754 binary64

784 The domains for elements that can be stored in collections and operated on through GraphBLAS
785 methods are defined by GraphBLAS objects called GrB_Type. The predefined types and cor-
786 responding domains used in the GraphBLAS C API are shown in Table 3.2. The Boolean type
787 (bool) is defined in `stdbool.h`, the integral types (`int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`,
788 `uint32_t`, `int64_t`, `uint64_t`) are defined in `stdint.h`, and the floating-point types (`float`,
789 `double`) are native to the language and platform and in most cases defined by the IEEE-754
790 standard.

791 3.4 Algebraic objects, operators and associated functions

792 GraphBLAS operators operate on elements stored in GraphBLAS collections. A *binary operator*
793 is a function that maps two input values to one output value. A *unary operator* is a function that
794 maps one input value to one output value. Binary operators are defined over two input domains
795 and produce an output from a (possibly different) third domain. Unary operators are specified over
796 one input domain and produce an output from a (possibly different) second domain.

797 In addition to the operators that operate on stored values, GraphBLAS also supports *index unary*
798 *operators* that maps a stored value and the indices of its position in the matrix or vector to an
799 output value. That output value can be used in the index unary operator variants of `apply` (§ 4.3.8)
800 to compute a new stored value, or be used in the `select` operation (§ 4.3.9) to determine if the stored
801 input value should be kept or annihilated.

802 Some GraphBLAS operations require a monoid or semiring. A monoid contains an associative
803 binary operator where the input and output domains are the same. The monoid also includes an
804 identity value of the operator. The semiring consists of a binary operator – referred to as the
805 “times” operator – with up to three different domains (two inputs and one output) and a monoid

Table 3.3: Operator input for relevant GraphBLAS operations. The semiring add and times are shown if applicable.

Operation	Operator input
mxm, mxv, vxm	semiring
eWiseAdd	binary operator monoid semiring (add)
eWiseMult	binary operator monoid semiring (times)
reduce (to vector or GrB_Scalar)	binary operator monoid
reduce (to scalar value)	monoid
apply	unary operator binary operator with scalar index unary operator
select	index unary operator
kroncker	binary operator monoid semiring
dup argument (build methods)	binary operator
accum argument (various methods)	binary operator

806 – referred to as the “plus” operator – that is also commutative. Furthermore, the domain of the
807 monoid must be the same as the output domain of the “times” operator.

808 The GraphBLAS *algebraic objects* operators, monoids, and semirings are presented in this section.
809 These objects can be used as input arguments to various GraphBLAS operations, as shown in
810 Table 3.3. The specific rules for each algebraic object are explained in the respective sections of
811 those objects. A summary of the properties and recipes for building these GraphBLAS algebraic
812 objects is presented in Table 3.4.

813 A number of predefined operators are specified by the GraphBLAS C API. They are presented
814 in tables in their respective subsections below. Each of these operators is defined to operate on
815 specific GraphBLAS types and therefore, this type is built into the name of the object as a suffix.
816 These suffixes and the corresponding predefined GrB_Type objects that are listed in Table 3.2.

817 3.4.1 Operators

818 A GraphBLAS *unary operator* $F_u = \langle D_{out}, D_{in}, f \rangle$ is defined by two domains, D_{out} and D_{in} , and
819 an operation $f : D_{in} \rightarrow D_{out}$. For a given GraphBLAS unary operator $F_u = \langle D_{out}, D_{in}, f \rangle$, we
820 define $\mathbf{D}_{out}(F_u) = D_{out}$, $\mathbf{D}_{in}(F_u) = D_{in}$, and $\mathbf{f}(F_u) = f$.

821 A GraphBLAS *binary operator* $F_b = \langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$ is defined by three domains, D_{out} , D_{in_1} ,

Table 3.4: Properties and recipes for building GraphBLAS algebraic objects: unary operator, binary operator, monoid, and semiring (composed of operations *add* and *times*).

(a) Properties of algebraic objects.

Object	Must be commutative	Must be associative	Identity must exist	Number of domains
Unary operator	n/a	n/a	n/a	2
Binary operator	no	no	no	3
Monoid	no	yes	yes	1
Reduction add	yes	yes	yes (see Note 1)	1
Semiring add	yes	yes	yes	1
Semiring times	no	no	no	3 (see Note 2)

(b) Recipes for algebraic objects.

Object	Recipe	Number of domains
Unary operator	Function pointer	2
Binary operator	Function pointer	3
Monoid	Associative binary operator with identity	1
Semiring	Commutative monoid + binary operator	3

Note 1: Some high-performance GraphBLAS implementations may require an identity to perform reductions to sparse objects like GraphBLAS vectors and scalars. According to the descriptions of the corresponding GraphBLAS operations, however, this identity is mathematically not necessary. There are API signatures to support both.

Note 2: The output domain of the semiring times must be same as the domain of the semiring's add monoid. This ensures three domains for a semiring rather than four.

822 D_{in_2} , and an operation $\odot : D_{in_1} \times D_{in_2} \rightarrow D_{out}$. For a given GraphBLAS binary operator $F_b =$
823 $\langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$, we define $\mathbf{D}_{out}(F_b) = D_{out}$, $\mathbf{D}_{in_1}(F_b) = D_{in_1}$, $\mathbf{D}_{in_2}(F_b) = D_{in_2}$, and $\odot(F_b) =$
824 \odot . Note that \odot could be used in place of either \oplus or \otimes in other methods and operations.

825 A GraphBLAS *index unary operator* $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB_Index}), D_{in_2}, f_i \rangle$ is defined by three
826 domains, D_{out} , D_{in_1} , D_{in_2} , the domain of GraphBLAS indices, and an operation $f_i : D_{in_1} \times I_{U64}^2 \times$
827 $D_{in_2} \rightarrow D_{out}$ (where I_{U64} corresponds to the domain of a `GrB_Index`). For a given GraphBLAS
828 index operator F_i , we define $\mathbf{D}_{out}(F_i) = D_{out}$, $\mathbf{D}_{in_1}(F_i) = D_{in_1}$, $\mathbf{D}_{in_2}(F_i) = D_{in_2}$, and $\mathbf{f}(F_i) = f_i$.

829 User-defined operators can be created with calls to `GrB_UnaryOp_new`, `GrB_BinaryOp_new`, and
830 `GrB_IndexUnaryOp_new`, respectively. See Section 4.2.1 for information on these methods. The
831 GraphBLAS C API predefines a number of these operators. These are listed in Tables 3.5 and 3.6.
832 Note that most entries in these tables represent a “family” of predefined operators for a set of
833 different types represented by the T , I , or F in their names. For example, the multiplicative
834 inverse (`GrB_MINV_F`) function is only defined for floating-point types ($F = \text{FP32}$ or FP64). The
835 division (`GrB_DIV_T`) function is defined for all types, but only if $y \neq 0$ for integral and floating
836 point types and $y \neq \text{false}$ for the Boolean type.

Table 3.5: Predefined unary and binary operators for GraphBLAS in C. The T can be any suffix from Table 3.2, I can be any integer suffix from Table 3.2, and F can be any floating-point suffix from Table 3.2.

Operator type	GraphBLAS identifier	Domains	Description	
GrB_UnaryOp	GrB_IDENTITY_ T	$T \rightarrow T$	$f(x) = x,$	identity
GrB_UnaryOp	GrB_ABS_ T	$T \rightarrow T$	$f(x) = x ,$	absolute value
GrB_UnaryOp	GrB_AINV_ T	$T \rightarrow T$	$f(x) = -x,$	additive inverse
GrB_UnaryOp	GrB_MINV_ F	$F \rightarrow F$	$f(x) = \frac{1}{x},$	multiplicative inverse
GrB_UnaryOp	GrB_LNOT	$\text{bool} \rightarrow \text{bool}$	$f(x) = \neg x,$	logical inverse
GrB_UnaryOp	GrB_BNOT_ I	$I \rightarrow I$	$f(x) = \sim x,$	bitwise complement
GrB_BinaryOp	GrB_LOR	$\text{bool} \times \text{bool} \rightarrow \text{bool}$	$f(x, y) = x \vee y,$	logical OR
GrB_BinaryOp	GrB_LAND	$\text{bool} \times \text{bool} \rightarrow \text{bool}$	$f(x, y) = x \wedge y,$	logical AND
GrB_BinaryOp	GrB_LXOR	$\text{bool} \times \text{bool} \rightarrow \text{bool}$	$f(x, y) = x \oplus y,$	logical XOR
GrB_BinaryOp	GrB_LXNOR	$\text{bool} \times \text{bool} \rightarrow \text{bool}$	$f(x, y) = \overline{x \oplus y},$	logical XNOR
GrB_BinaryOp	GrB_BOR_ I	$I \times I \rightarrow I$	$f(x, y) = x y,$	bitwise OR
GrB_BinaryOp	GrB_BAND_ I	$I \times I \rightarrow I$	$f(x, y) = x \& y,$	bitwise AND
GrB_BinaryOp	GrB_BXOR_ I	$I \times I \rightarrow I$	$f(x, y) = x \hat{\ } y,$	bitwise XOR
GrB_BinaryOp	GrB_BXNOR_ I	$I \times I \rightarrow I$	$f(x, y) = \overline{x \hat{\ } y},$	bitwise XNOR
GrB_BinaryOp	GrB_EQ_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x == y)$	equal
GrB_BinaryOp	GrB_NE_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x \neq y)$	not equal
GrB_BinaryOp	GrB_GT_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x > y)$	greater than
GrB_BinaryOp	GrB_LT_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x < y)$	less than
GrB_BinaryOp	GrB_GE_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x \geq y)$	greater than or equal
GrB_BinaryOp	GrB_LE_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x \leq y)$	less than or equal
GrB_BinaryOp	GrB_ONEB_ T	$T \times T \rightarrow T$	$f(x, y) = 1,$	1 (cast to T)
GrB_BinaryOp	GrB_FIRST_ T	$T \times T \rightarrow T$	$f(x, y) = x,$	first argument
GrB_BinaryOp	GrB_SECOND_ T	$T \times T \rightarrow T$	$f(x, y) = y,$	second argument
GrB_BinaryOp	GrB_MIN_ T	$T \times T \rightarrow T$	$f(x, y) = (x < y) ? x : y,$	minimum
GrB_BinaryOp	GrB_MAX_ T	$T \times T \rightarrow T$	$f(x, y) = (x > y) ? x : y,$	maximum
GrB_BinaryOp	GrB_PLUS_ T	$T \times T \rightarrow T$	$f(x, y) = x + y,$	addition
GrB_BinaryOp	GrB_MINUS_ T	$T \times T \rightarrow T$	$f(x, y) = x - y,$	subtraction
GrB_BinaryOp	GrB_TIMES_ T	$T \times T \rightarrow T$	$f(x, y) = xy,$	multiplication
GrB_BinaryOp	GrB_DIV_ T	$T \times T \rightarrow T$	$f(x, y) = \frac{x}{y},$	division

Table 3.6: Predefined index unary operators for GraphBLAS in C. The T can be any suffix from Table 3.2. I_{U64} refers to the unsigned 64-bit, GrB_Index, integer type, I_{32} refers to the signed, 32-bit integer type, and I_{64} refers to signed, 64-bit integer type. The parameters, u_i or A_{ij} , are the stored values from the containers where the i and j parameters are set to the row and column indices corresponding to the location of the stored value. When operating on vectors, j will be passed with a zero value. Finally, s is an additional scalar value used in the operators. The expressions in the “Description” column are to be treated as mathematical specifications. That is, for the index arithmetic functions in the first two groups below, each one of i , j , and s is interpreted as an integer number in the set \mathbb{Z} . Functions are evaluated using arithmetic in \mathbb{Z} , producing a result value that is also in \mathbb{Z} . The result value is converted to the output type according to the rules of the C language. In particular, if the value cannot be represented as a signed 32- or 64-bit integer type, the output is implementation defined. Any deviations from this ideal behavior, including limitations on the values of i , j , and s , or possible overflow and underflow conditions, must be defined by the implementation.

Operator type Type	GraphBLAS Name	Domains (– is don’t care)				Description
		A, u	i, j	s	result	
GrB_IndexUnaryOp	GrB_ROWINDEX_ $I_{32/64}$	–	I_{U64}	$I_{32/64}$	$I_{32/64}$	$f(A_{ij}, i, j, s) = (i + s)$, replace with its row index (+ s)
GrB_IndexUnaryOp	GrB_COLINDEX_ $I_{32/64}$	–	I_{U64}	$I_{32/64}$	$I_{32/64}$	$f(u_i, i, 0, s) = (i + s)$ $f(A_{ij}, i, j, s) = (j + s)$ replace with its column index (+ s)
GrB_IndexUnaryOp	GrB_DIAGINDEX_ $I_{32/64}$	–	I_{U64}	$I_{32/64}$	$I_{32/64}$	$f(A_{ij}, i, j, s) = (j - i + s)$ replace with its diagonal index (+ s)
GrB_IndexUnaryOp	GrB_TRIL	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s) = (j \leq i + s)$ triangle on or below diagonal s
GrB_IndexUnaryOp	GrB_TRIU	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s) = (j \geq i + s)$ triangle on or above diagonal s
GrB_IndexUnaryOp	GrB_DIAG	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s) = (j == i + s)$ diagonal s
GrB_IndexUnaryOp	GrB_OFFDIAG	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s) = (j \neq i + s)$ all but diagonal s
GrB_IndexUnaryOp	GrB_COLLE	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s) = (j \leq s)$ columns less or equal to s
GrB_IndexUnaryOp	GrB_COLGT	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s) = (j > s)$ columns greater than s
GrB_IndexUnaryOp	GrB_ROWLE	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s) = (i \leq s)$, rows less or equal to s
GrB_IndexUnaryOp	GrB_ROWGT	–	I_{U64}	I_{64}	bool	$f(u_i, i, 0, s) = (i \leq s)$ $f(A_{ij}, i, j, s) = (i > s)$, rows greater than s
GrB_IndexUnaryOp	GrB_VALUEEQ_ T	T	–	T	bool	$f(A_{ij}, i, j, s) = (A_{ij} == s)$, elements equal to value s
GrB_IndexUnaryOp	GrB_VALUENE_ T	T	–	T	bool	$f(u_i, i, 0, s) = (u_i == s)$ $f(A_{ij}, i, j, s) = (A_{ij} \neq s)$, elements not equal to value s
GrB_IndexUnaryOp	GrB_VALUELT_ T	T	–	T	bool	$f(u_i, i, 0, s) = (u_i \neq s)$ $f(A_{ij}, i, j, s) = (A_{ij} < s)$, elements less than value s
GrB_IndexUnaryOp	GrB_VALUELE_ T	T	–	T	bool	$f(u_i, i, 0, s) = (u_i < s)$ $f(A_{ij}, i, j, s) = (A_{ij} \leq s)$, elements less or equal to value s
GrB_IndexUnaryOp	GrB_VALUEGT_ T	T	–	T	bool	$f(u_i, i, 0, s) = (u_i \leq s)$ $f(A_{ij}, i, j, s) = (A_{ij} > s)$, elements greater than value s
GrB_IndexUnaryOp	GrB_VALUEGE_ T	T	–	T	bool	$f(u_i, i, 0, s) = (u_i > s)$ $f(A_{ij}, i, j, s) = (A_{ij} \geq s)$, elements greater or equal to value s
		T	–	T	bool	$f(u_i, i, 0, s) = (u_i \geq s)$

837 3.4.2 Monoids

838 A GraphBLAS *monoid* $M = \langle D, \odot, 0 \rangle$ is defined by a single domain D , an *associative*¹ operation
839 $\odot : D \times D \rightarrow D$, and an identity element $0 \in D$. For a given GraphBLAS monoid $M = \langle D, \odot, 0 \rangle$
840 we define $\mathbf{D}(M) = D$, $\odot(M) = \odot$, and $\mathbf{0}(M) = 0$. A GraphBLAS monoid is equivalent to the
841 conventional *monoid* algebraic structure.

842 Let $F = \langle D, D, D, \odot \rangle$ be an associative GraphBLAS binary operator with identity element $0 \in D$.
843 Then $M = \langle F, 0 \rangle = \langle D, \odot, 0 \rangle$ is a GraphBLAS monoid. If \odot is commutative, then M is said to be
844 a *commutative monoid*. If a monoid M is created using an operator \odot that is not associative, the
845 outcome of GraphBLAS operations using such a monoid is undefined.

846 User-defined monoids can be created with calls to `GrB_Monoid_new` (see Section 4.2.1). The
847 GraphBLAS C API predefines a number of monoids that are listed in Table 3.7. Predefined
848 monoids are named `GrB_op_MONOID_T`, where *op* is the name of the predefined GraphBLAS
849 operator used as the associative binary operation of the monoid and T is the domain (type) of the
850 monoid.

851 3.4.3 Semirings

852 A GraphBLAS *semiring* $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$ is defined by three domains D_{out} , D_{in_1} , and
853 D_{in_2} ; an *associative*¹ and commutative additive operation $\oplus : D_{out} \times D_{out} \rightarrow D_{out}$; a multiplicative
854 operation $\otimes : D_{in_1} \times D_{in_2} \rightarrow D_{out}$; and an identity element $0 \in D_{out}$. For a given GraphBLAS
855 semiring $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$ we define $\mathbf{D}_{in_1}(S) = D_{in_1}$, $\mathbf{D}_{in_2}(S) = D_{in_2}$, $\mathbf{D}_{out}(S) =$
856 D_{out} , $\oplus(S) = \oplus$, $\otimes(S) = \otimes$, and $\mathbf{0}(S) = 0$.

857 Let $F = \langle D_{out}, D_{in_1}, D_{in_2}, \otimes \rangle$ be an operator and let $A = \langle D_{out}, \oplus, 0 \rangle$ be a commutative monoid,
858 then $S = \langle A, F \rangle = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$ is a semiring.

859 In a GraphBLAS semiring, the multiplicative operator does not have to distribute over the additive
860 operator. This is unlike the conventional *semiring* algebraic structure.

861 Note: There must be one GraphBLAS monoid in every semiring which serves as the semiring's
862 additive operator and specifies the same domain for its inputs and output parameters. If this
863 monoid is not a commutative monoid, the outcome of GraphBLAS operations using the semiring
864 is undefined.

865 A UML diagram of the conceptual hierarchy of object classes in GraphBLAS algebra (binary
866 operators, monoids, and semirings) is shown in Figure 3.1.

867 User-defined semirings can be created with calls to `GrB_Semiring_new` (see Section 4.2.1). A list of
868 predefined true semirings and convenience semirings can be found in Tables 3.8 and 3.9, respectively.
869 Predefined semirings are named `GrB_add_mul_SEMIRING_T`, where *add* is the semiring additive
870 operation, *mul* is the semiring multiplicative operation and T is the domain (type) of the semiring.

¹It is expected that implementations of the GraphBLAS will utilize floating point arithmetic such as that defined in the IEEE-754 standard even though floating point arithmetic is not strictly associative.

Table 3.7: Predefined monoids for GraphBLAS in C. Maximum and minimum values for the various integral types are defined in `stdint.h`. Floating-point infinities are defined in `math.h`. The x in `UINT x` or `INT x` can be one of 8, 16, 32, or 64; whereas in `FP x` , it can be 32 or 64.

GraphBLAS identifier	Domains, T ($T \times T \rightarrow T$)	Identity	Description
GrB_PLUS_MONOID_ T	UINT x	0	addition
	INT x	0	
	FP x	0	
GrB_TIMES_MONOID_ T	UINT x	1	multiplication
	INT x	1	
	FP x	1	
GrB_MIN_MONOID_ T	UINT x	UINT x _MAX	minimum
	INT x	INT x _MAX	
	FP x	INFINITY	
GrB_MAX_MONOID_ T	UINT x	0	maximum
	INT x	INT x _MIN	
	FP x	-INFINITY	
GrB_LOR_MONOID_BOOL	BOOL	false	logical OR
GrB_LAND_MONOID_BOOL	BOOL	true	logical AND
GrB_LXOR_MONOID_BOOL	BOOL	false	logical XOR (not equal)
GrB_LXNOR_MONOID_BOOL	BOOL	true	logical XNOR (equal)

Table 3.8: Predefined true semirings for GraphBLAS in C where the additive identity is the multiplicative annihilator. The x can be one of 8, 16, 32, or 64 in $UINTx$ or $INTx$, and can be 32 or 64 in FPx .

GraphBLAS identifier	Domains, T ($T \times T \rightarrow T$)	+ identity \times annihilator	Description
GrB_PLUS_TIMES_SEMIRING_T	$UINTx$ $INTx$ FPx	0 0 0	arithmetic semiring
GrB_MIN_PLUS_SEMIRING_T	$UINTx$ $INTx$ FPx	$UINTx_MAX$ $INTx_MAX$ INFINITY	min-plus semiring
GrB_MAX_PLUS_SEMIRING_T	$INTx$ FPx	$INTx_MIN$ -INFINITY	max-plus semiring
GrB_MIN_TIMES_SEMIRING_T	$UINTx$	$UINTx_MAX$	min-times semiring
GrB_MIN_MAX_SEMIRING_T	$UINTx$ $INTx$ FPx	$UINTx_MAX$ $INTx_MAX$ INFINITY	min-max semiring
GrB_MAX_MIN_SEMIRING_T	$UINTx$ $INTx$ FPx	0 $INTx_MIN$ -INFINITY	max-min semiring
GrB_MAX_TIMES_SEMIRING_T	$UINTx$	0	max-times semiring
GrB_PLUS_MIN_SEMIRING_T	$UINTx$	0	plus-min semiring
GrB_LOR_LAND_SEMIRING_BOOL	BOOL	false	Logical semiring
GrB_LAND_LOR_SEMIRING_BOOL	BOOL	true	"and-or" semiring
GrB_LXOR_LAND_SEMIRING_BOOL	BOOL	false	same as NE_LAND
GrB_LXNOR_LOR_SEMIRING_BOOL	BOOL	true	same as EQ_LOR

Table 3.9: Other useful predefined semirings for GraphBLAS in C that don't have a multiplicative annihilator. The x can be one of 8, 16, 32, or 64 in $UINTx$ or $INTx$, and can be 32 or 64 in FPx .

GraphBLAS identifier	Domains, T ($T \times T \rightarrow T$)	+ identity	Description
GrB_MAX_PLUS_SEMIRING_	$UINTx$	0	max-plus semiring
GrB_MIN_TIMES_SEMIRING_	$INTx$	$INTx_MAX$	min-times semiring
	FPx	$INFINITY$	
GrB_MAX_TIMES_SEMIRING_	$INTx$	$INTx_MIN$	max-times semiring
	FPx	$-INFINITY$	
GrB_PLUS_MIN_SEMIRING_	$INTx$	0	plus-min semiring
	FPx	0	
GrB_MIN_FIRST_SEMIRING_	$UINTx$	$UINTx_MAX$	min-select first semiring
	$INTx$	$INTx_MAX$	
	FPx	$INFINITY$	
GrB_MIN_SECOND_SEMIRING_	$UINTx$	$UINTx_MAX$	min-select second semiring
	$INTx$	$INTx_MAX$	
	FPx	$INFINITY$	
GrB_MAX_FIRST_SEMIRING_	$UINTx$	0	max-select first semiring
	$INTx$	$INTx_MIN$	
	FPx	$-INFINITY$	
GrB_MAX_SECOND_SEMIRING_	$UINTx$	0	max-select second semiring
	$INTx$	$INTx_MIN$	
	FPx	$-INFINITY$	

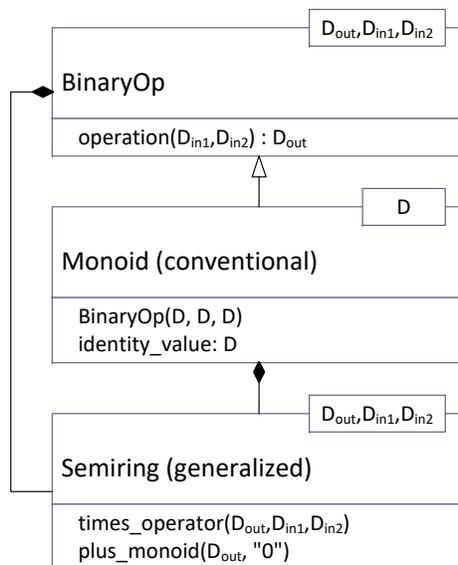


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

871 3.5 Collections

872 3.5.1 Scalars

873 A *GraphBLAS scalar*, $s = \langle D, \{\sigma\} \rangle$, is defined by a domain D , and a set of zero or one *scalar value*,
 874 σ , where $\sigma \in D$. We define $\mathbf{size}(s) = 1$ (constant), and $\mathbf{L}(s) = \{\sigma\}$. The set $\mathbf{L}(s)$ is called the
 875 *contents* of the GraphBLAS scalar s . We also define $\mathbf{D}(s) = D$. Finally, $\mathbf{val}(s)$ is a reference to
 876 the scalar value, σ , if the GraphBLAS scalar is not empty, and is undefined otherwise.

877 3.5.2 Vectors

878 A vector $\mathbf{v} = \langle D, N, \{(i, v_i)\} \rangle$ is defined by a domain D , a size $N > 0$, and a set of tuples (i, v_i)
 879 where $0 \leq i < N$ and $v_i \in D$. A particular value of i can appear at most once in \mathbf{v} . We define
 880 $\mathbf{size}(\mathbf{v}) = N$ and $\mathbf{L}(\mathbf{v}) = \{(i, v_i)\}$. The set $\mathbf{L}(\mathbf{v})$ is called the *content* of vector \mathbf{v} . We also define
 881 the set $\mathbf{ind}(\mathbf{v}) = \{i : (i, v_i) \in \mathbf{L}(\mathbf{v})\}$ (called the *structure* of \mathbf{v}), and $\mathbf{D}(\mathbf{v}) = D$. For a vector \mathbf{v} ,
 882 $\mathbf{v}(i)$ is a reference to v_i if $(i, v_i) \in \mathbf{L}(\mathbf{v})$ and is undefined otherwise.

883 3.5.3 Matrices

884 A matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$ is defined by a domain D , its number of rows $M > 0$, its
 885 number of columns $N > 0$, and a set of tuples (i, j, A_{ij}) where $0 \leq i < M$, $0 \leq j < N$, and
 886 $A_{ij} \in D$. A particular pair of values i, j can appear at most once in \mathbf{A} . We define $\mathbf{ncols}(\mathbf{A}) = N$,
 887 $\mathbf{nrows}(\mathbf{A}) = M$, and $\mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\}$. The set $\mathbf{L}(\mathbf{A})$ is called the *content* of matrix \mathbf{A} . We also
 888 define the sets $\mathbf{indrow}(\mathbf{A}) = \{i : \exists(i, j, A_{ij}) \in \mathbf{A}\}$ and $\mathbf{indcol}(\mathbf{A}) = \{j : \exists(i, j, A_{ij}) \in \mathbf{A}\}$. (These
 889 are the sets of nonempty rows and columns of \mathbf{A} , respectively.) The *structure* of matrix \mathbf{A} is the
 890 set $\mathbf{ind}(\mathbf{A}) = \{(i, j) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}$, and $\mathbf{D}(\mathbf{A}) = D$. For a matrix \mathbf{A} , $\mathbf{A}(i, j)$ is a reference to
 891 A_{ij} if $(i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})$ and is undefined otherwise.

892 If \mathbf{A} is a matrix and $0 \leq j < N$, then $\mathbf{A}(:, j) = \langle D, M, \{(i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\} \rangle$ is a
 893 vector called the j -th *column* of \mathbf{A} . Correspondingly, if \mathbf{A} is a matrix and $0 \leq i < M$, then
 894 $\mathbf{A}(i, :) = \langle D, N, \{(j, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\} \rangle$ is a vector called the i -th *row* of \mathbf{A} .

895 Given a matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$, its *transpose* is another matrix $\mathbf{A}^T = \langle D, N, M, \{(j, i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\} \rangle$.

897 3.5.3.1 External matrix formats

898 The specification also supports the export and import of matrices to/from a number of commonly
 899 used formats, such as COO, CSR, and CSC formats. When importing or exporting a matrix to or
 900 from a GraphBLAS object using `GrB_Matrix_import` (§ 4.2.4.17) or `GrB_Matrix_export` (§ 4.2.4.16),
 901 it is necessary to specify the data format for the matrix data external to GraphBLAS, which is
 902 being imported from or exported to. This non-opaque data format is specified using an argument of
 903 enumeration type `GrB_Format` that is used to indicate one of a number of predefined formats. The
 904 predefined values of `GrB_Format` are specified in Table 3.10. A precise definition of the non-opaque
 905 data formats can be found in Appendix B.

Table 3.10: `GrB_Format` enumeration literals and corresponding values for matrix import and export methods.

Symbol	Value	Description
<code>GrB_CSR_FORMAT</code>	0	Specifies the compressed sparse row matrix format.
<code>GrB_CSC_FORMAT</code>	1	Specifies the compressed sparse column matrix format.
<code>GrB_COO_FORMAT</code>	2	Specifies the sparse coordinate matrix format.

906 3.5.4 Masks

907 The GraphBLAS C API defines an opaque object called a *mask*. The mask is used to control how
 908 computed values are stored in the output from a method. The mask is an *internal* opaque object;
 909 that is, it is never exposed as a variable within an application.

910 The mask is formed from input objects to the method that uses the mask. For example, a Graph-
 911 BLAS method may be called with a matrix as the mask parameter. The internal mask object is

912 constructed from the input matrix in one of two ways. In the default case, an element of the mask
 913 is created for each tuple that exists in the matrix for which the value of the tuple cast to Boolean
 914 evaluates to `true`. Alternatively, the user can specify *structure*-only behavior where an element of
 915 the mask is created for each tuple that exists in the matrix *regardless* of the value stored in the
 916 input matrix.

917 The internal mask object can be either a one- or a two-dimensional construct. One- and two-
 918 dimensional masks, described more formally below, are similar to vectors and matrices, respectively,
 919 except that they have structure (indices) but no values. When needed, a value is implied for the
 920 elements of a mask with an implied value of `true` for elements that exist and an implied value
 921 of `false` for elements that do not exist (i.e., the locations of the mask that do not have a stored
 922 value imply a value of `false`). Hence, even though a mask does not contain any values, it can be
 923 considered to imply values from a Boolean domain.

924 A one-dimensional mask $\mathbf{m} = \langle N, \{i\} \rangle$ is defined by its number of elements $N > 0$, and a set
 925 $\mathbf{ind}(\mathbf{m})$ of indices $\{i\}$ where $0 \leq i < N$. A particular value of i can appear at most once in \mathbf{m} . We
 926 define $\mathbf{size}(\mathbf{m}) = N$. The set $\mathbf{ind}(\mathbf{m})$ is called the *structure* of mask \mathbf{m} .

927 A two-dimensional mask $\mathbf{M} = \langle M, N, \{(i, j)\} \rangle$ is defined by its number of rows $M > 0$, its number
 928 of columns $N > 0$, and a set $\mathbf{ind}(\mathbf{M})$ of tuples (i, j) where $0 \leq i < M, 0 \leq j < N$. A particular pair
 929 of values i, j can appear at most once in \mathbf{M} . We define $\mathbf{ncols}(\mathbf{M}) = N$, and $\mathbf{nrows}(\mathbf{M}) = M$. We
 930 also define the sets $\mathbf{indrow}(\mathbf{M}) = \{i : \exists(i, j) \in \mathbf{ind}(\mathbf{M})\}$ and $\mathbf{indcol}(\mathbf{M}) = \{j : \exists(i, j) \in \mathbf{ind}(\mathbf{M})\}$.
 931 These are the sets of nonempty rows and columns of \mathbf{M} , respectively. The set $\mathbf{ind}(\mathbf{M})$ is called the
 932 *structure* of mask \mathbf{M} .

933 One common operation on masks is the *complement*. For a one-dimensional mask \mathbf{m} this is denoted
 934 as $\neg\mathbf{m}$. For a two-dimensional mask \mathbf{M} , this is denoted as $\neg\mathbf{M}$. The complement of a one-
 935 dimensional mask \mathbf{m} is defined as $\mathbf{ind}(\neg\mathbf{m}) = \{i : 0 \leq i < N, i \notin \mathbf{ind}(\mathbf{m})\}$. It is the set of all
 936 possible indices that do not appear in \mathbf{m} . The complement of a two-dimensional mask \mathbf{M} is defined
 937 as the set $\mathbf{ind}(\neg\mathbf{M}) = \{(i, j) : 0 \leq i < M, 0 \leq j < N, (i, j) \notin \mathbf{ind}(\mathbf{M})\}$. It is the set of all possible
 938 indices that do not appear in \mathbf{M} .

939 3.6 Descriptors

940 Descriptors are used to modify the behavior of a GraphBLAS method. When present in the
 941 signature of a method, they appear as the last argument in the method. Descriptors specify how
 942 the other input arguments corresponding to GraphBLAS collections – vectors, matrices, and masks
 943 – should be processed (modified) before the main operation of a method is performed. A complete
 944 list of what descriptors are capable of are presented in this section.

945 The descriptor is a lightweight object. It is composed of (*field*, *value*) pairs where the *field* selects
 946 one of the GraphBLAS objects from the argument list of a method and the *value* defines the
 947 indicated modification associated with that object. For example, a descriptor may specify that a
 948 particular input matrix needs to be transposed or that a mask needs to be complemented (defined
 949 in Section 3.5.4) before using it in the operation.

950 For the purpose of constructing descriptors, the arguments of a method that can be modified

951 are identified by specific field names. The output parameter (typically the first parameter in a
952 GraphBLAS method) is indicated by the field name, `GrB_OUTP`. The mask is indicated by the
953 `GrB_MASK` field name. The input parameters corresponding to the input vectors and matrices are
954 indicated by `GrB_INP0` and `GrB_INP1` in the order they appear in the signature of the GraphBLAS
955 method. The descriptor is an opaque object and hence we do not define how objects of this type
956 should be implemented. When referring to *(field, value)* pairs for a descriptor, however, we often use
957 the informal notation `desc[GrB_Desc_Field].GrB_Desc_Value` without implying that a descriptor is
958 to be implemented as an array of structures (in fact, field values can be used in conjunction with
959 multiple values that are composable). We summarize all types, field names, and values used with
960 descriptors in Table 3.11.

961 In the definitions of the GraphBLAS methods, we often refer to the *default behavior* of a method
962 with respect to the action of a descriptor. If a descriptor is not provided or if the value associated
963 with a particular field in a descriptor is not set, the default behavior of a GraphBLAS method is
964 defined as follows:

- 965 • Input matrices are not transposed.
- 966 • The mask is used, as is, without complementing, and stored values are examined to determine
967 whether they evaluate to `true` or `false`.
- 968 • Values of the output object that are not directly modified by the operation are preserved.

969 GraphBLAS specifies all of the valid combinations of (field, value) pairs as predefined descriptors.
970 Their identifiers and the corresponding set of (field, value) pairs for that identifier are shown in
971 Table 3.12.

972 3.7 GrB_Info return values

973 All GraphBLAS methods return a `GrB_Info` enumeration value. The three types of return codes
974 (informational, API error, and execution error) and their corresponding values are listed in Ta-
975 ble 3.13.

Table 3.11: Descriptors are GraphBLAS objects passed as arguments to GraphBLAS operations to modify other GraphBLAS objects in the operation’s argument list. A descriptor, `desc`, has one or more (*field, value*) pairs indicated as `desc[GrB_Desc_Field].GrB_Desc_Value`. In this table, we define all types and literals used with descriptors.

(a) Types used with GraphBLAS descriptors.

Type	Description
<code>GrB_Descriptor</code>	Type of a GraphBLAS descriptor object.
<code>GrB_Desc_Field</code>	The descriptor field enumeration.
<code>GrB_Desc_Value</code>	The descriptor value enumeration.

(b) Descriptor field names of type `GrB_Desc_Field` enumeration and corresponding values.

Field Name	Value	Description
<code>GrB_OUTP</code>	0	Field name for the output GraphBLAS object.
<code>GrB_MASK</code>	1	Field name for the mask GraphBLAS object.
<code>GrB_INP0</code>	2	Field name for the first input GraphBLAS object.
<code>GrB_INP1</code>	3	Field name for the second input GraphBLAS object.

(c) Descriptor field values of type `GrB_Desc_Value` enumeration and corresponding values.

Value Name	Value	Description
(reserved)	0	Unused
<code>GrB_REPLACE</code>	1	Clear the output object before assigning computed values.
<code>GrB_COMP</code>	2	Use the complement of the associated object. When combined with <code>GrB_STRUCTURE</code> , the complement of the structure of the associated object is used without evaluating the values stored.
<code>GrB_TRAN</code>	3	Use the transpose of the associated object.
<code>GrB_STRUCTURE</code>	4	The write mask is constructed from the structure (pattern of stored values) of the associated object. The stored values are not examined.

Table 3.12: Predefined GraphBLAS descriptors. The list includes all possible descriptors, according to the current standard. Columns list the possible fields and entries list the value(s) associated with those fields for a given descriptor.

Identifier	GrB_OUTP	GrB_MASK	GrB_INP0	GrB_INP1
GrB_NULL	–	–	–	–
GrB_DESC_T1	–	–	–	GrB_TRAN
GrB_DESC_T0	–	–	GrB_TRAN	–
GrB_DESC_T0T1	–	–	GrB_TRAN	GrB_TRAN
GrB_DESC_C	–	GrB_COMP	–	–
GrB_DESC_S	–	GrB_STRUCTURE	–	–
GrB_DESC_CT1	–	GrB_COMP	–	GrB_TRAN
GrB_DESC_ST1	–	GrB_STRUCTURE	–	GrB_TRAN
GrB_DESC_CT0	–	GrB_COMP	GrB_TRAN	–
GrB_DESC_ST0	–	GrB_STRUCTURE	GrB_TRAN	–
GrB_DESC_CT0T1	–	GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_ST0T1	–	GrB_STRUCTURE	GrB_TRAN	GrB_TRAN
GrB_DESC_SC	–	GrB_STRUCTURE, GrB_COMP	–	–
GrB_DESC_SCT1	–	GrB_STRUCTURE, GrB_COMP	–	GrB_TRAN
GrB_DESC_SCT0	–	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	–
GrB_DESC_SCT0T1	–	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_R	GrB_REPLACE	–	–	–
GrB_DESC_RT1	GrB_REPLACE	–	–	GrB_TRAN
GrB_DESC_RT0	GrB_REPLACE	–	GrB_TRAN	–
GrB_DESC_RT0T1	GrB_REPLACE	–	GrB_TRAN	GrB_TRAN
GrB_DESC_RC	GrB_REPLACE	GrB_COMP	–	–
GrB_DESC_RS	GrB_REPLACE	GrB_STRUCTURE	–	–
GrB_DESC_RCT1	GrB_REPLACE	GrB_COMP	–	GrB_TRAN
GrB_DESC_RST1	GrB_REPLACE	GrB_STRUCTURE	–	GrB_TRAN
GrB_DESC_RCT0	GrB_REPLACE	GrB_COMP	GrB_TRAN	–
GrB_DESC_RST0	GrB_REPLACE	GrB_STRUCTURE	GrB_TRAN	–
GrB_DESC_RCT0T1	GrB_REPLACE	GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_RST0T1	GrB_REPLACE	GrB_STRUCTURE	GrB_TRAN	GrB_TRAN
GrB_DESC_RSC	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	–	–
GrB_DESC_RSCT1	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	–	GrB_TRAN
GrB_DESC_RSCT0	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	–
GrB_DESC_RSCT0T1	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	GrB_TRAN

Table 3.13: Enumeration literals and corresponding values returned by GraphBLAS methods and operations.

(a) Informational return values

Symbol	Value	Description
GrB_SUCCESS	0	The method/operation completed successfully (blocking mode), or encountered no API errors (non-blocking mode).
GrB_NO_VALUE	1	A location in a matrix or vector is being accessed that has no stored value at the specified location.

(b) API errors

Symbol	Value	Description
GrB_UNINITIALIZED_OBJECT	-1	A GraphBLAS object is passed to a method before new was called on it.
GrB_NULL_POINTER	-2	A NULL is passed for a pointer parameter.
GrB_INVALID_VALUE	-3	Miscellaneous incorrect values.
GrB_INVALID_INDEX	-4	Indices passed are larger than dimensions of the matrix or vector being accessed.
GrB_DOMAIN_MISMATCH	-5	A mismatch between domains of collections and operations when user-defined domains are in use.
GrB_DIMENSION_MISMATCH	-6	Operations on matrices and vectors with incompatible dimensions.
GrB_OUTPUT_NOT_EMPTY	-7	An attempt was made to build a matrix or vector using an output object that already contains valid tuples (elements).
GrB_NOT_IMPLEMENTED	-8	An attempt was made to call a GraphBLAS method for a combination of input parameters that is not supported by a particular implementation.

(c) Execution errors

Symbol	Value	Description
GrB_PANIC	-101	Unknown internal error.
GrB_OUT_OF_MEMORY	-102	Not enough memory for operations.
GrB_INSUFFICIENT_SPACE	-103	The array provided is not large enough to hold output.
GrB_INVALID_OBJECT	-104	One of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error.
GrB_INDEX_OUT_OF_BOUNDS	-105	Reference to a vector or matrix element that is outside the defined dimensions of the object.
GrB_EMPTY_OBJECT	-106	One of the opaque GraphBLAS objects does not have a stored value.

976 Chapter 4

977 Methods

978 This chapter defines the behavior of all the methods in the GraphBLAS C API. All methods can
979 be declared for use in programs by including the `GraphBLAS.h` header file.

980 We would like to emphasize that no GraphBLAS method will imply a predefined order over any
981 associative operators. Implementations of the GraphBLAS are encouraged to exploit associativity
982 to optimize performance of any GraphBLAS method. This holds even if the definition of the
983 GraphBLAS method implies a fixed order for the associative operations.

984 4.1 Context methods

985 The methods in this section set up and tear down the GraphBLAS context within which all Graph-
986 BLAS methods must be executed. The initialization of this context also includes the specification
987 of which execution mode is to be used.

988 4.1.1 `init`: Initialize a GraphBLAS context

989 Creates and initializes a GraphBLAS C API context.

990 C Syntax

```
991     GrB_Info GrB_init(GrB_Mode mode);
```

992 Parameters

993 mode Mode for the GraphBLAS context. Must be either `GrB_BLOCKING` or `GrB_NONBLOCKING`.

994 **Return Values**

995 `GrB_SUCCESS` operation completed successfully.

996 `GrB_PANIC` unknown internal error.

997 `GrB_INVALID_VALUE` invalid mode specified, or method called multiple times.

998 **Description**

999 The `init` method creates and initializes a GraphBLAS C API context. The argument to `GrB_init`
1000 defines the mode for the context. The two available modes are:

- 1001 • `GrB_BLOCKING`: In this mode, each method in a sequence returns after its computations have
1002 completed and output arguments are available to subsequent statements in an application.
1003 When executing in `GrB_BLOCKING` mode, the methods execute in program order.
- 1004 • `GrB_NONBLOCKING`: In this mode, methods in a sequence may return after arguments in
1005 the method have been tested for dimension and domain compatibility within the method
1006 but potentially before their computations complete. Output arguments are available to sub-
1007 sequent GraphBLAS methods in an application. When executing in `GrB_NONBLOCKING`
1008 mode, the methods in a sequence may execute in any order that preserves the mathematical
1009 result defined by the sequence.

1010 An application can only create one context per execution instance. An application may only call
1011 `GrB_Init` once. Calling `GrB_Init` more than once results in undefined behavior.

1012 **4.1.2 finalize: Finalize a GraphBLAS context**

1013 Terminates and frees any internal resources created to support the GraphBLAS C API context.

1014 **C Syntax**

1015 `GrB_Info GrB_finalize();`

1016 **Return Values**

1017 `GrB_SUCCESS` operation completed successfully.

1018 `GrB_PANIC` unknown internal error.

1019 **Description**

1020 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS
1021 C API context. `GrB_finalize` may only be called after a context has been initialized by calling
1022 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-
1023 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined
1024 behavior.

1025 **4.1.3 getVersion: Get the version number of the standard.**

1026 Query the library for the version number of the standard that this library implements.

1027 **C Syntax**

```
1028         GrB_Info GrB_getVersion(unsigned int *version,  
1029                               unsigned int *subversion);
```

1030 **Parameters**

1031 `version` (OUT) On successful return will hold the value of the major version number.

1032 `subversion` (OUT) On successful return will hold the value of the subversion number.

1033 **Return Values**

1034 `GrB_SUCCESS` operation completed successfully.

1035 `GrB_PANIC` unknown internal error.

1036 **Description**

1037 The `getVersion` method is used to query the major and minor version number of the GraphBLAS
1038 C API specification that the library implements at runtime. To support compile time queries the
1039 following two macros shall also be defined by the library.

```
1040     #define GRB_VERSION      2  
1041     #define GRB_SUBVERSION  0
```

1042 **4.2 Object methods**

1043 This section describes methods that setup and operate on GraphBLAS opaque objects but are not
1044 part of the the GraphBLAS math specification.

1045 4.2.1 Algebra methods

1046 4.2.1.1 Type_new: Construct a new GraphBLAS (user-defined) type

1047 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,
1048 monoids, semirings, vectors and matrices.

1049 C Syntax

```
1050         GrB_Info GrB_Type_new(GrB_Type  *utype,  
1051                               size_t    sizeof(ctype));
```

1052 Parameters

1053 utype (INOUT) On successful return, contains a handle to the newly created user-defined
1054 GraphBLAS type object.

1055 ctype (IN) A C type that defines the new GraphBLAS user-defined type.

1056 Return Values

1057 GrB_SUCCESS operation completed successfully.

1058 GrB_PANIC unknown internal error.

1059 GrB_OUT_OF_MEMORY not enough memory available for operation.

1060 GrB_NULL_POINTER utype pointer is NULL.

1061 Description

1062 Given a C type ctype, the Type_new method returns in utype a handle to a new GraphBLAS type
1063 that is equivalent to the C type. Variables of this ctype must be a struct, union, or fixed-size array.
1064 In particular, given two variables, src and dst, of type ctype, the following operation must be a
1065 valid way to copy the contents of src to dst:

```
1066         memcpy(&dst, &src, sizeof(ctype))
```

1067 A new, user-defined type utype should be destroyed with a call to GrB_free(utype) when no longer
1068 needed.

1069 It is not an error to call this method more than once on the same variable; however, the handle to
1070 the previously created object will be overwritten.

1071 **4.2.1.2 UnaryOp_new: Construct a new GraphBLAS unary operator**

1072 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types
1073 (domains).

1074 **C Syntax**

```
1075     GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1076                             void          (*unary_func)(void*, const void*),  
1077                             GrB_Type      d_out,  
1078                             GrB_Type      d_in);
```

1079 **Parameters**

1080 unary_op (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1081 unary operator object.

1082 unary_func (IN) a pointer to a user-defined function that takes one input parameter of d_in's
1083 type and returns a value of d_out's type, both passed as void pointers. Specifically
1084 the signature of the function is expected to be of the form:

```
1085         void func(void *out, const void *in);  
1086
```

1087 d_out (IN) The GrB_Type of the return value of the unary operator being created. Should
1088 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1089 BLAS type.

1090 d_in (IN) The GrB_Type of the input argument of the unary operator being created.
1091 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1092 GraphBLAS type.

1093 **Return Values**

1094 GrB_SUCCESS operation completed successfully.

1095 GrB_PANIC unknown internal error.

1096 GrB_OUT_OF_MEMORY not enough memory available for operation.

1097 GrB_UNINITIALIZED_OBJECT any GrB_Type parameter (for user-defined types) has not been ini-
1098 tialized by a call to GrB_Type_new.

1099 GrB_NULL_POINTER unary_op or unary_func pointers are NULL.

1100 **Description**

1101 The `UnaryOp_new` method creates a new GraphBLAS unary operator

1102 $f_u = \langle \mathbf{D}(d_out), \mathbf{D}(d_in), unary_func \rangle$

1103 and returns a handle to it in `unary_op`.

1104 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments
1105 are aliased. In other words, for all invocations of the function:

1106 `unary_func(out, in);`

1107 the value of `out` must be the same as if the following code was executed:

```
1108     D(d_in) *tmp = malloc(sizeof(D(d_in)));  
1109     memcpy(tmp, in, sizeof(D(d_in)));  
1110     unary_func(out, tmp);  
1111     free(tmp);
```

1112 It is not an error to call this method more than once on the same variable; however, the handle to
1113 the previously created object will be overwritten.

1114 **4.2.1.3 BinaryOp_new: Construct a new GraphBLAS binary operator**

1115 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types
1116 (domains).

1117 **C Syntax**

```
1118     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
1119                             void          (*binary_func)(void*,  
1120                                                         const void*,  
1121                                                         const void*),  
1122                             GrB_Type      d_out,  
1123                             GrB_Type      d_in1,  
1124                             GrB_Type      d_in2);
```

1125 **Parameters**

1126 `binary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1127 binary operator object.

1128 `binary_func` (IN) A pointer to a user-defined function that takes two input parameters of types
1129 `d_in1` and `d_in2` and returns a value of type `d_out`, all passed as void pointers.
1130 Specifically the signature of the function is expected to be of the form:

```
1131         void func(void *out, const void *in1, const void *in2);  
1132
```

1133 `d_out` (IN) The `GrB_Type` of the return value of the binary operator being created. Should
1134 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1135 BLAS type.

1136 `d_in1` (IN) The `GrB_Type` of the left hand argument of the binary operator being created.
1137 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1138 GraphBLAS type.

1139 `d_in2` (IN) The `GrB_Type` of the right hand argument of the binary operator being cre-
1140 ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-
1141 defined GraphBLAS type.

1142 Return Values

1143 `GrB_SUCCESS` operation completed successfully.

1144 `GrB_PANIC` unknown internal error.

1145 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1146 `GrB_UNINITIALIZED_OBJECT` the `GrB_Type` (for user-defined types) has not been initialized by a
1147 call to `GrB_Type_new`.

1148 `GrB_NULL_POINTER` `binary_op` or `binary_func` pointer is NULL.

1149 Description

1150 The `BinaryOp_new` methods creates a new GraphBLAS binary operator

```
1151      $f_b = \langle \mathbf{D}(d\_out), \mathbf{D}(d\_in1), \mathbf{D}(d\_in2), \text{binary\_func} \rangle$ 
```

1152 and returns a handle to it in `binary_op`.

1153 The implementation of `binary_func` must be such that it works even if any of the `d_out`, `d_in1`, and
1154 `d_in2` arguments are aliased to each other. In other words, for all invocations of the function:

```
1155     binary_func(out, in1, in2);
```

1156 the value of `out` must be the same as if the following code was executed:

```

1157     D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1158     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1159     memcpy(tmp1,in1,sizeof(D(d_in1)));
1160     memcpy(tmp2,in2,sizeof(D(d_in2)));
1161     binary_func(out,tmp1,tmp2);
1162     free(tmp2);
1163     free(tmp1);

```

1164 It is not an error to call this method more than once on the same variable; however, the handle to
1165 the previously created object will be overwritten.

1166 4.2.1.4 Monoid_new: Construct a new GraphBLAS monoid

1167 Creates a new monoid with specified binary operator and identity value.

1168 C Syntax

```

1169     GrB_Info GrB_Monoid_new(GrB_Monoid *monoid,
1170                           GrB_BinaryOp binary_op,
1171                           <type>      identity);

```

1172 Parameters

1173 **monoid** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1174 monoid object.

1175 **binary_op** (IN) An existing GraphBLAS associative binary operator whose input and output
1176 types are the same.

1177 **identity** (IN) The value of the identity element of the monoid. Must be the same type as
1178 the type used by the **binary_op** operator.

1179 Return Values

1180 **GrB_SUCCESS** operation completed successfully.

1181 **GrB_PANIC** unknown internal error.

1182 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1183 **GrB_UNINITIALIZED_OBJECT** the **GrB_BinaryOp** (for user-defined operators) has not been initial-
1184 ized by a call to **GrB_BinaryOp_new**.

1185 **GrB_NULL_POINTER** monoid pointer is NULL.

1186 **GrB_DOMAIN_MISMATCH** all three argument types of the binary operator and the type of the
1187 identity value are not the same.

1188 **Description**

1189 The `Monoid_new` method creates a new monoid $M = \langle \mathbf{D}(\text{binary_op}), \text{binary_op}, \text{identity} \rangle$ and re-
1190 turns a handle to it in `monoid`.

1191 If `binary_op` is not associative, the results of GraphBLAS operations that require associativity of
1192 this monoid will be undefined.

1193 It is not an error to call this method more than once on the same variable; however, the handle to
1194 the previously created object will be overwritten.

1195 **4.2.1.5 Semiring_new: Construct a new GraphBLAS semiring**

1196 Creates a new semiring with specified domain, operators, and elements.

1197 **C Syntax**

```
1198         GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1199                                 GrB_Monoid   add_op,  
1200                                 GrB_BinaryOp  mul_op);
```

1201 **Parameters**

1202 `semiring` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1203 `semiring`.

1204 `add_op` (IN) An existing GraphBLAS commutative monoid that specifies the addition op-
1205 erator and its identity.

1206 `mul_op` (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-
1207 plication operator. In addition, `mul_op`'s output domain, $\mathbf{D}_{out}(\text{mul_op})$, must be
1208 the same as the `add_op`'s domain $\mathbf{D}(\text{add_op})$.

1209 **Return Values**

1210 `GrB_SUCCESS` operation completed successfully.

1211 `GrB_PANIC` unknown internal error.

1212 `GrB_OUT_OF_MEMORY` not enough memory available for this method to complete.

1213 `GrB_UNINITIALIZED_OBJECT` the `add_op` (for user-define monoids) object has not been initialized
1214 with a call to `GrB_Monoid_new` or the `mul_op` (for user-defined
1215 operators) object has not been not been initialized by a call to
1216 `GrB_BinaryOp_new`.

1217 GrB_NULL_POINTER semiring pointer is NULL.

1218 GrB_DOMAIN_MISMATCH the output domain of mul_op does not match the domain of the
1219 add_op monoid.

1220 Description

1221 The Semiring_new method creates a new semiring:

1222 $S = \langle \mathbf{D}_{out}(\text{mul_op}), \mathbf{D}_{in_1}(\text{mul_op}), \mathbf{D}_{in_2}(\text{mul_op}), \text{add_op}, \text{mul_op}, \mathbf{0}(\text{add_op}) \rangle$

1223 and returns a handle to it in semiring. Note that $\mathbf{D}_{out}(\text{mul_op})$ must be the same as $\mathbf{D}(\text{add_op})$.

1224 If add_op is not commutative, then GraphBLAS operations using this semiring will be undefined.

1225 It is not an error to call this method more than once on the same variable; however, the handle to
1226 the previously created object will be overwritten.

1227 4.2.1.6 IndexUnaryOp_new: Construct a new GraphBLAS index unary operator

1228 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its
1229 types (domains).

1230 C Syntax

```
1231 GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp *index_unary_op,  
1232                               void (*index_unary_func)(void*,  
1233                                                         const void*,  
1234                                                         GrB_Index,  
1235                                                         GrB_Index,  
1236                                                         const void*),  
1237                               GrB_Type d_out,  
1238                               GrB_Type d_in1,  
1239                               GrB_Type d_in2);
```

1240 Parameters

1241 index_unary_op (INOUT) On successful return, contains a handle to the newly created Graph-
1242 BLAS index unary operator object.

1243 index_unary_func (IN) A pointer to a user-defined function that takes input parameters of types
1244 d_in1, GrB_Index, GrB_Index and d_in2 and returns a value of type d_out. Ex-
1245 cept for the GrB_Index parameters, all are passed as void pointers. Specifically
1246 the signature of the function is expected to be of the form:

```

1247         void func(void      *out,
1248                 const void *in1,
1249                 GrB_Index  row_index,
1250                 GrB_Index  col_index,
1251                 const void *in2);
1252

```

1253 **d_out** (IN) The GrB_Type of the return value of the index unary operator being created.
 1254 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
 1255 GraphBLAS type.

1256 **d_in1** (IN) The GrB_Type of the first input argument of the index unary operator being
 1257 created and corresponds to the stored values of the GrB_Vector or GrB_Matrix
 1258 being operated on. Should be one of the predefined GraphBLAS types in Ta-
 1259 ble 3.2, or a user-defined GraphBLAS type.

1260 **d_in2** (IN) The GrB_Type of the last input argument of the index unary operator be-
 1261 ing created and corresponds to a scalar provided by the GraphBLAS operation
 1262 that uses this operator. Should be one of the predefined GraphBLAS types in
 1263 Table 3.2, or a user-defined GraphBLAS type.

1264 Return Values

1265 GrB_SUCCESS operation completed successfully.

1266 GrB_PANIC unknown internal error.

1267 GrB_OUT_OF_MEMORY not enough memory available for operation.

1268 GrB_UNINITIALIZED_OBJECT the GrB_Type (for user-defined types) has not been initialized by a
 1269 call to GrB_Type_new.

1270 GrB_NULL_POINTER index_unary_op or index_unary_func pointer is NULL.

1271 Description

1272 The IndexUnaryOp_new methods creates a new GraphBLAS index unary operator

1273 $f_i = \langle \mathbf{D}(d_out), \mathbf{D}(d_in1), \mathbf{D}(GrB_Index), \mathbf{D}(GrB_Index), \mathbf{D}(d_in2), index_unary_func) \rangle$

1274 and returns a handle to it in index_unary_op.

1275 The implementation of index_unary_func must be such that it works even if any of the d_out,
 1276 d_in1, and d_in2 arguments are aliased to each other. In other words, for all invocations of the
 1277 function:

```

1278     index_unary_func(out, in1, row_index, col_index, n, in2);

```

1279 the value of out must be the same as if the following code was executed (shown here for matrices):

```
1280     GrB_Index row_index = ...;
1281     GrB_Index col_index = ...;
1282     D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1283     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1284     memcpy(tmp1,in1,sizeof(D(d_in1)));
1285     memcpy(tmp2,in2,sizeof(D(d_in2)));
1286     index_unary_func(out,tmp1,row_index,col_index,tmp2);
1287     free(tmp2);
1288     free(tmp1);
```

1289 It is not an error to call this method more than once on the same variable; however, the handle to
1290 the previously created object will be overwritten.

1291 4.2.2 Scalar methods

1292 4.2.2.1 Scalar_new: Construct a new scalar

1293 Creates a new empty scalar with specified domain.

1294 C Syntax

```
1295     GrB_Info GrB_Scalar_new(GrB_Scalar *s,
1296                             GrB_Type    d);
```

1297 Parameters

1298 s (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1299 scalar.

1300 d (IN) The type corresponding to the domain of the scalar being created. Can be
1301 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1302 GraphBLAS type.

1303 Return Values

1304 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1305 blocking mode, this indicates that the API checks for the input
1306 arguments passed successfully. Either way, output scalar s is ready
1307 to be used in the next method of the sequence.

1308 GrB_PANIC Unknown internal error.

1309 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1310 GraphBLAS objects (input or output) is in an invalid state caused
1311 by a previous execution error. Call GrB_error() to access any error
1312 messages generated by the implementation.

1313 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1314 GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to GrB_Type_new
1315 (needed for user-defined types).

1316 GrB_NULL_POINTER The s pointer is NULL.

1317 **Description**

1318 Creates a new GraphBLAS scalar s of domain $\mathbf{D}(d)$ and empty $\mathbf{L}(s)$. The method returns a handle
1319 to the new scalar in s .

1320 It is not an error to call this method more than once on the same variable; however, the handle to
1321 the previously created object will be overwritten.

1322 **4.2.2.2 Scalar_dup: Construct a copy of a GraphBLAS scalar**

1323 Creates a new scalar with the same domain and contents as another scalar.

1324 **C Syntax**

```
1325           GrB_Info GrB_Scalar_dup(GrB_Scalar        *t,  
1326                                    const GrB_Scalar  s);
```

1327 **Parameters**

1328 t (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1329 scalar.

1330 s (IN) The GraphBLAS scalar to be duplicated.

1331 **Return Values**

1332 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1333 blocking mode, this indicates that the API checks for the input
1334 arguments passed successfully. Either way, output scalar t is ready
1335 to be used in the next method of the sequence.

1336 GrB_PANIC Unknown internal error.

1337 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1338 GraphBLAS objects (input or output) is in an invalid state caused
1339 by a previous execution error. Call GrB_error() to access any error
1340 messages generated by the implementation.

1341 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1342 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to
1343 Scalar_new or Scalar_dup.

1344 GrB_NULL_POINTER The *t* pointer is NULL.

1345 **Description**

1346 Creates a new scalar *t* of domain $\mathbf{D}(s)$ and contents $\mathbf{L}(s)$. The method returns a handle to the new
1347 scalar in *t*.

1348 It is not an error to call this method more than once with the same output variable; however, the
1349 handle to the previously created object will be overwritten.

1350 **4.2.2.3 Scalar_clear: Clear/remove a stored value from a scalar**

1351 Removes the stored value from a scalar.

1352 **C Syntax**

1353 GrB_Info GrB_Scalar_clear(GrB_Scalar s);

1354 **Parameters**

1355 *s* (INOUT) An existing GraphBLAS scalar to clear.

1356 **Return Values**

1357 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1358 blocking mode, this indicates that the API checks for the input
1359 arguments passed successfully. Either way, output scalar *s* is ready
1360 to be used in the next method of the sequence.

1361 GrB_PANIC Unknown internal error.

1362 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1363 GraphBLAS objects (input or output) is in an invalid state caused
1364 by a previous execution error. Call GrB_error() to access any error
1365 messages generated by the implementation.

1366 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1367 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to
1368 Scalar_new or Scalar_dup.

1369 **Description**

1370 Removes the stored value from an existing scalar. After the call, **L(s)** is empty. The size of the
1371 scalar does not change.

1372 **4.2.2.4 Scalar_nvals: Number of stored elements in a scalar**

1373 Retrieve the number of stored elements in a scalar (either zero or one).

1374 **C Syntax**

```
1375                GrB_Info GrB_Scalar_nvals(GrB_Index                *nvals,  
1376                                                const GrB_Scalar s);
```

1377 **Parameters**

1378 *nvals* (OUT) On successful return, this is set to the number of stored elements in the
1379 scalar (zero or one).

1380 *s* (IN) An existing GraphBLAS scalar being queried.

1381 **Return Values**

1382 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1383 cessfully and the value of *nvals* has been set.

1384 GrB_PANIC Unknown internal error.

1385 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1386 GraphBLAS objects (input or output) is in an invalid state caused
1387 by a previous execution error. Call GrB_error() to access any error
1388 messages generated by the implementation.

1389 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1390 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to
1391 Scalar_new or Scalar_dup.

1392 GrB_NULL_POINTER The *nvals* pointer is NULL.

1393 **Description**

1394 Return `nvals(s)` in `nvals`. This is the number of stored elements in scalar `s`, which is the size of
1395 `L(s)`, and can only be either zero or one (see Section 3.5.1).

1396 **4.2.2.5 Scalar_setElement: Set the single element in a scalar**

1397 Set the single element of a scalar to a given value.

1398 **C Syntax**

```
1399         GrB_Info GrB_Scalar_setElement(GrB_Scalar  s,  
1400                                     <type>      val);
```

1401 **Parameters**

1402 `s` (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1403 `val` (IN) Scalar value to assign. The type must be compatible with the domain of `s`.

1404 **Return Values**

1405 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1406 blocking mode, this indicates that the compatibility tests on in-
1407 dex/dimensions and domains for the input arguments passed suc-
1408 cessfully. Either way, the output scalar `s` is ready to be used in the
1409 next method of the sequence.

1410 `GrB_PANIC` Unknown internal error.

1411 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1412 GraphBLAS objects (input or output) is in an invalid state caused
1413 by a previous execution error. Call `GrB_error()` to access any error
1414 messages generated by the implementation.

1415 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1416 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to
1417 `Scalar_new` or `Scalar_dup`.

1418 `GrB_DOMAIN_MISMATCH` The domains of `s` and `val` are incompatible.

1419 **Description**

1420 First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ must
1421 be compatible with $\mathbf{D}(s)$. Two domains are compatible with each other if values from one domain
1422 can be cast to values in the other domain as per the rules of the C language. In particular, domains
1423 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-
1424 patible with itself. If any compatibility rule above is violated, execution of `GrB_Scalar_setElement`
1425 ends and the domain mismatch error listed above is returned.

1426 We are now ready to carry out the assignment `val`; that is:

1427
$$s(0) = \text{val}$$

1428 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1429 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1430 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1431 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be
1432 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1433 **4.2.2.6 Scalar_extractElement: Extract a single element from a scalar.**

1434 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

1435 **C Syntax**

```
1436     GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1437                                     const GrB_Scalar s);
```

1438 **Parameters**

1439 `val` (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain
1440 of scalar `s`. On successful return, `val` holds the result of the operation, and any
1441 previous value in `val` is overwritten.

1442 `s` (IN) The GraphBLAS scalar from which an element is extracted.

1443 **Return Values**

1444 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
1445 cessfully. This indicates that the compatibility tests on dimensions
1446 and domains for the input arguments passed successfully, and the
1447 output scalar, `val`, has been computed and is ready to be used in
1448 the next method of the sequence.

1449 `GrB_PANIC` Unknown internal error.

1450 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1451 GraphBLAS objects (input or output) is in an invalid state caused
1452 by a previous execution error. Call GrB_error() to access any error
1453 messages generated by the implementation.

1454 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1455 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, s, has not been initialized by a call to
1456 Scalar_new or Scalar_dup.

1457 GrB_NULL_POINTER val pointer is NULL.

1458 GrB_DOMAIN_MISMATCH The domains of the scalar or scalar are incompatible.

1459 GrB_NO_VALUE There is no stored value in the scalar.

1460 **Description**

1461 First, val and input GraphBLAS scalar are tested for domain compatibility as follows: $D(\text{val})$
1462 must be compatible with $D(s)$. Two domains are compatible with each other if values from
1463 one domain can be cast to values in the other domain as per the rules of the C language. In
1464 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1465 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1466 GrB_Scalar_extractElement ends and the domain mismatch error listed above is returned.

1467 Then, if no value is currently stored in the GraphBLAS scalar, the method returns GrB_NO_VALUE
1468 and val remains unchanged.

1469 Finally the extract into the output argument, val can be performed; that is:

$$1470 \qquad \text{val} = s(0)$$

1471 In both GrB_BLOCKING mode GrB_NONBLOCKING mode if the method exits with return value
1472 GrB_SUCCESS, the new contents of val are as defined above.

1473 **4.2.3 Vector methods**

1474 **4.2.3.1 Vector_new: Construct new vector**

1475 Creates a new vector with specified domain and size.

1476 **C Syntax**

```
1477       GrB_Info GrB_Vector_new(GrB_Vector *v,
1478                               GrB_Type   d,
1479                               GrB_Index  nsize);
```

1480 **Parameters**

1481 \mathbf{v} (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1482 vector.

1483 \mathbf{d} (IN) The type corresponding to the domain of the vector being created. Can be
1484 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1485 GraphBLAS type.

1486 \mathbf{nsz} (IN) The size of the vector being created.

1487 **Return Values**

1488 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1489 blocking mode, this indicates that the API checks for the input
1490 arguments passed successfully. Either way, output vector \mathbf{v} is ready
1491 to be used in the next method of the sequence.

1492 `GrB_PANIC` Unknown internal error.

1493 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1494 GraphBLAS objects (input or output) is in an invalid state caused
1495 by a previous execution error. Call `GrB_error()` to access any error
1496 messages generated by the implementation.

1497 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1498 `GrB_UNINITIALIZED_OBJECT` The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`
1499 (needed for user-defined types).

1500 `GrB_NULL_POINTER` The \mathbf{v} pointer is NULL.

1501 `GrB_INVALID_VALUE` \mathbf{nsz} is zero or outside the range of the type `GrB_Index`.

1502 **Description**

1503 Creates a new vector \mathbf{v} of domain $\mathbf{D}(\mathbf{d})$, size \mathbf{nsz} , and empty $\mathbf{L}(\mathbf{v})$. The method returns a handle
1504 to the new vector in \mathbf{v} .

1505 It is not an error to call this method more than once on the same variable; however, the handle to
1506 the previously created object will be overwritten.

1507 **4.2.3.2 Vector_dup: Construct a copy of a GraphBLAS vector**

1508 Creates a new vector with the same domain, size, and contents as another vector.

1509 **C Syntax**

```
1510         GrB_Info GrB_Vector_dup(GrB_Vector      *w,  
1511                                 const GrB_Vector u);
```

1512 **Parameters**

1513 **w** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1514 vector.

1515 **u** (IN) The GraphBLAS vector to be duplicated.

1516 **Return Values**

1517 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1518 blocking mode, this indicates that the API checks for the input
1519 arguments passed successfully. Either way, output vector **w** is ready
1520 to be used in the next method of the sequence.

1521 **GrB_PANIC** Unknown internal error.

1522 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1523 GraphBLAS objects (input or output) is in an invalid state caused
1524 by a previous execution error. Call **GrB_error()** to access any error
1525 messages generated by the implementation.

1526 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1527 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **u**, has not been initialized by a call to
1528 **Vector_new** or **Vector_dup**.

1529 **GrB_NULL_POINTER** The **w** pointer is NULL.

1530 **Description**

1531 Creates a new vector **w** of domain **D(u)**, size **size(u)**, and contents **L(u)**. The method returns a
1532 handle to the new vector in **w**.

1533 It is not an error to call this method more than once on the same variable; however, the handle to
1534 the previously created object will be overwritten.

1535 **4.2.3.3 Vector_resize: Resize a vector**

1536 Changes the size of an existing vector.

1537 **C Syntax**

```
1538         GrB_Info GrB_Vector_resize(GrB_Vector w,  
1539                                   GrB_Index  nsize);
```

1540 **Parameters**

1541 `w` (INOUT) An existing Vector object that is being resized.

1542 `nsize` (IN) The new size of the vector. It can be smaller or larger than the current size.

1543 **Return Values**

1544 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1545 blocking mode, this indicates that the API checks for the input
1546 arguments passed successfully. Either way, output vector `w` is ready
1547 to be used in the next method of the sequence.

1548 `GrB_PANIC` Unknown internal error.

1549 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1550 GraphBLAS objects (input or output) is in an invalid state caused
1551 by a previous execution error. Call `GrB_error()` to access any error
1552 messages generated by the implementation.

1553 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1554 `GrB_NULL_POINTER` The `w` pointer is NULL.

1555 `GrB_INVALID_VALUE` `nsize` is zero or outside the range of the type `GrB_Index`.

1556 **Description**

1557 Changes the size of `w` to `nsize`. The domain $\mathbf{D}(w)$ of vector `w` remains the same. The contents $\mathbf{L}(w)$
1558 are modified as described below.

1559 Let $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$ when the method is called. When the method returns, $w = \langle \mathbf{D}(w), nsize, \mathbf{L}'(w) \rangle$
1560 where $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < nsize)\}$. That is, all elements of `w` with index greater
1561 than or equal to the new vector size (`nsize`) are dropped.

1562 **4.2.3.4 Vector_clear: Clear a vector**

1563 Removes all the elements (tuples) from a vector.

1564 **C Syntax**

1565 `GrB_Info GrB_Vector_clear(GrB_Vector v);`

1566 **Parameters**

1567 `v` (INOUT) An existing GraphBLAS vector to clear.

1568 **Return Values**

1569 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1570 blocking mode, this indicates that the API checks for the input
1571 arguments passed successfully. Either way, output vector `v` is ready
1572 to be used in the next method of the sequence.

1573 `GrB_PANIC` Unknown internal error.

1574 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1575 GraphBLAS objects (input or output) is in an invalid state caused
1576 by a previous execution error. Call `GrB_error()` to access any error
1577 messages generated by the implementation.

1578 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1579 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS vector, `v`, has not been initialized by a call to
1580 `Vector_new` or `Vector_dup`.

1581 **Description**

1582 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,
1583 $L(v) = \emptyset$. The size of the vector does not change.

1584 **4.2.3.5 Vector_size: Size of a vector**

1585 Retrieve the size of a vector.

1586 **C Syntax**

1587 `GrB_Info GrB_Vector_size(GrB_Index *nsize,`
1588 `const GrB_Vector v);`

1614 **Return Values**

1615 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1616 cessfully and the value of `nvals` has been set.

1617 GrB_PANIC Unknown internal error.

1618 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1619 GraphBLAS objects (input or output) is in an invalid state caused
1620 by a previous execution error. Call `GrB_error()` to access any error
1621 messages generated by the implementation.

1622 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1623 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to
1624 `Vector_new` or `Vector_dup`.

1625 GrB_NULL_POINTER The `nvals` pointer is NULL.

1626 **Description**

1627 Return `nvals(v)` in `nvals`. This is the number of stored elements in vector `v`, which is the size of
1628 `L(v)` (see Section 3.5.2).

1629 **4.2.3.7 Vector_build: Store elements from tuples into a vector**

1630 **C Syntax**

```
1631           GrB_Info GrB_Vector_build(GrB_Vector            w,  
1632                                    const GrB_Index       *indices,  
1633                                    const <type>         *values,  
1634                                    GrB_Index            n,  
1635                                    const GrB_BinaryOp    dup);
```

1636 **Parameters**

1637 w (INOUT) An existing Vector object to store the result.

1638 indices (IN) Pointer to an array of indices.

1639 values (IN) Pointer to an array of scalars of a type that is compatible with the domain of
1640 vector `w`.

1641 n (IN) The number of entries contained in each array (the same for indices and values).

1642 **dup** (IN) An associative and commutative binary operator to apply when duplicate
 1643 values for the same location are present in the input arrays. All three domains of
 1644 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
 1645 then duplicate locations will result in an error.

1646 Return Values

1647 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 1648 blocking mode, this indicates that the API checks for the input
 1649 arguments passed successfully. Either way, output vector **w** is
 1650 ready to be used in the next method of the sequence.

1651 **GrB_PANIC** Unknown internal error.

1652 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
 1653 opaque GraphBLAS objects (input or output) is in an invalid
 1654 state caused by a previous execution error. Call **GrB_error()** to
 1655 access any error messages generated by the implementation.

1656 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1657 **GrB_UNINITIALIZED_OBJECT** Either **w** has not been initialized by a call to by **GrB_Vector_new**
 1658 or by **GrB_Vector_dup**, or **dup** has not been initialized by a call
 1659 to by **GrB_BinaryOp_new**.

1660 **GrB_NULL_POINTER** indices or values pointer is **NULL**.

1661 **GrB_INDEX_OUT_OF_BOUNDS** A value in indices is outside the allowed range for **w**.

1662 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
 1663 not all the same, or the domains of **values** and **w** are incompatible
 1664 with each other or D_{dup} .

1665 **GrB_OUTPUT_NOT_EMPTY** Output vector **w** already contains valid tuples (elements). In
 1666 other words, **GrB_Vector_nvals(C)** returns a positive value.

1667 **GrB_INVALID_VALUE** indices contains a duplicate location and **dup** is **GrB_NULL**.

1668 Description

1669 If **dup** is not **GrB_NULL**, an internal vector $\tilde{\mathbf{w}} = \langle D_{dup}, \mathbf{size}(\mathbf{w}), \emptyset \rangle$ is created, which only differs
 1670 from **w** in its domain; otherwise, $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$.

1671 Each tuple $\{\text{indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the output in the form of

$$1672 \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \mathbf{dup} \neq \mathbf{GrB_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1673 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,
 1674 `dup` is used to reduce the values before assignment into $\tilde{\mathbf{w}}$ as follows:

$$1675 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1676 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{w}}$ is copied into `w` via typecasting its
 1677 values to `D(w)` if necessary. If \oplus is not associative or not commutative, the result is undefined.

1678 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1679 It is an error to call this function on an output object with existing elements. In other words,
 1680 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1681 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`
 1682 or `values`.

1683 4.2.3.8 Vector_setElement: Set a single element in a vector

1684 Set one element of a vector to a given value.

1685 C Syntax

```
1686 // scalar value
1687 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1688                               <type>        val,
1689                               GrB_Index      index);
1690
1691 // GraphBLAS scalar
1692 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1693                               const GrB_Scalar s,
1694                               GrB_Index      index);
```

1695 Parameters

1696 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1697 `val` or `s` (IN) Scalar assign. Its domain (type) must be compatible with the domain of `w`.

1698 `index` (IN) The location of the element to be assigned.

1699 Return Values

1700 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 1701 blocking mode, this indicates that the compatibility tests on in-
 1702 dex/dimensions and domains for the input arguments passed suc-

1703 cessfully. Either way, the output vector w is ready to be used in
1704 the next method of the sequence.

1705 `GrB_PANIC` Unknown internal error.

1706 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1707 GraphBLAS objects (input or output) is in an invalid state caused
1708 by a previous execution error. Call `GrB_error()` to access any error
1709 messages generated by the implementation.

1710 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1711 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS vector, w , or GraphBLAS scalar, s , has not been
1712 initialized by a call to a respective constructor.

1713 `GrB_INVALID_INDEX` index specifies a location that is outside the dimensions of w .

1714 `GrB_DOMAIN_MISMATCH` The domains of the vector and the scalar are incompatible.

1715 Description

1716 First, the scalar and output vector are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$
1717 must be compatible with $\mathbf{D}(w)$. Two domains are compatible with each other if values from
1718 one domain can be cast to values in the other domain as per the rules of the C language. In
1719 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1720 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1721 `GrB_Vector_setElement` ends and the domain mismatch error listed above is returned.

1722 Then, the index parameter is checked for a valid value where the following condition must hold:

$$1723 \quad 0 \leq \text{index} < \text{size}(w)$$

1724 If this condition is violated, execution of `GrB_Vector_setElement` ends and the invalid index error
1725 listed above is returned.

We are now ready to carry out the assignment; that is:

$$w(\text{index}) = \begin{cases} \mathbf{L}(s), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

1726 In the case of a transparent scalar or if $\mathbf{L}(s)$ is not empty, then a value will be stored at the
1727 specified location in w , overwriting any value that may have been stored there before. In the case
1728 of a GraphBLAS scalar, if $\mathbf{L}(s)$ is empty, then any value stored at the specified location in w will
1729 be removed.

1730 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1731 of w is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1732 return value `GrB_SUCCESS` and the new contents of vector w is as defined above but may not be
1733 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1734 **4.2.3.9 Vector_removeElement: Remove an element from a vector**

1735 Remove (annihilate) one stored element from a vector.

1736 **C Syntax**

```
1737         GrB_Info GrB_Vector_removeElement(GrB_Vector  w,  
1738                                         GrB_Index   index);
```

1739 **Parameters**

1740 w (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1741 index (IN) The location of the element to be removed.

1742 **Return Values**

1743 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1744 blocking mode, this indicates that the compatibility tests on in-
1745 dex/dimensions and domains for the input arguments passed suc-
1746 cessfully. Either way, the output vector w is ready to be used in
1747 the next method of the sequence.

1748 GrB_PANIC Unknown internal error.

1749 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1750 GraphBLAS objects (input or output) is in an invalid state caused
1751 by a previous execution error. Call GrB_error() to access any error
1752 messages generated by the implementation.

1753 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1754 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, w, has not been initialized by a call to
1755 Vector_new or Vector_dup.

1756 GrB_INVALID_INDEX index specifies a location that is outside the dimensions of w.

1757 **Description**

1758 First, the index parameter is checked for a valid value where the following condition must hold:

$$1759 \quad 0 \leq \text{index} < \text{size}(w)$$

1760 If this condition is violated, execution of GrB_Vector_removeElement ends and the invalid index
1761 error listed above is returned.

1762 We are now ready to carry out the removal of a value that may be stored at the location specified
1763 by `index`. If a value does not exist at the specified location in `w`, no error is reported and the
1764 operation has no effect on the state of `w`. In either case, the following will be true on return from
1765 the method: `index` \notin `ind(w)`.

1766 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1767 of `w` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1768 return value `GrB_SUCCESS` and the new content of vector `w` is as defined above but may not be
1769 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1770 4.2.3.10 `Vector_extractElement`: Extract a single element from a vector.

1771 Extract one element of a vector into a scalar.

1772 C Syntax

```
1773     // scalar value
1774     GrB_Info GrB_Vector_extractElement(<type>          *val,
1775                                     const GrB_Vector  u,
1776                                     GrB_Index         index);
1777
1778     // GraphBLAS scalar
1779     GrB_Info GrB_Vector_extractElement(GrB_Scalar     s,
1780                                     const GrB_Vector  u,
1781                                     GrB_Index         index);
```

1782 Parameters

1783 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector
1784 `u`. On successful return, this scalar holds the result of the extract. Any previous
1785 value stored in `val` or `s` is overwritten.

1786 `u` (IN) The GraphBLAS vector from which an element is extracted.

1787 `index` (IN) The location in `u` to extract.

1788 Return Values

1789 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
1790 cessfully. This indicates that the compatibility tests on dimensions
1791 and domains for the input arguments passed successfully, and the
1792 output scalar, `val` or `s`, has been computed and is ready to be used
1793 in the next method of the sequence.

1794 GrB_NO_VALUE When using the transparent scalar, `val`, this is returned when there
1795 is no stored value at specified location.

1796 GrB_PANIC Unknown internal error.

1797 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1798 GraphBLAS objects (input or output) is in an invalid state caused
1799 by a previous execution error. Call `GrB_error()` to access any error
1800 messages generated by the implementation.

1801 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1802 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `u`, or scalar, `s`, has not been initialized by
1803 a call to a corresponding constructor.

1804 GrB_NULL_POINTER `val` pointer is NULL.

1805 GrB_INVALID_INDEX `index` specifies a location that is outside the dimensions of `w`.

1806 GrB_DOMAIN_MISMATCH The domains of the vector and scalar are incompatible.

1807 **Description**

1808 First, the scalar and input vector are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$
1809 must be compatible with $\mathbf{D}(u)$. Two domains are compatible with each other if values from
1810 one domain can be cast to values in the other domain as per the rules of the C language. In
1811 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1812 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1813 `GrB_Vector_extractElement` ends and the domain mismatch error listed above is returned.

1814 Then, the `index` parameter is checked for a valid value where the following condition must hold:

1815
$$0 \leq \text{index} < \text{size}(u)$$

1816 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the invalid index
1817 error listed above is returned.

We are now ready to carry out the extract into the output scalar; that is:

$$\left. \begin{array}{l} \mathbf{L}(s) \\ \text{val} \end{array} \right\} = u(\text{index})$$

1818 If $\text{index} \in \mathbf{ind}(u)$, then the corresponding value from `u` is copied into `s` or `val` with casting as
1819 necessary. If $\text{index} \notin \mathbf{ind}(u)$, then one of the follow occurs depending on output scalar type:

- 1820 • The GraphBLAS scalar, `s`, is cleared and `GrB_SUCCESS` is returned.
- 1821 • The non-opaque scalar, `val`, is unchanged, and `GrB_NO_VALUE` is returned.

1822 When using the non-opaque scalar variant (`val`) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
1823 mode, the new contents of `val` are as defined above if the method exits with return value `GrB_SUCCESS`
1824 or `GrB_NO_VALUE`.

1825 When using the GraphBLAS scalar variant (`s`) with a `GrB_SUCCESS` return value, the method
1826 exits and the new contents of `s` is as defined above and fully computed in `GrB_BLOCKING` mode.
1827 In `GrB_NONBLOCKING` mode, the new contents of `s` is as defined above but may not be fully
1828 computed; however, it can be used in the next GraphBLAS method call in a sequence.

1829 4.2.3.11 `Vector_extractTuples`: Extract tuples from a vector

1830 Extract the contents of a GraphBLAS vector into non-opaque data structures.

1831 C Syntax

```
1832     GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,  
1833                                     <type>         *values,  
1834                                     GrB_Index      *n,  
1835                                     const GrB_Vector v);  
1836
```

1837 `indices` (OUT) Pointer to an array of indices that is large enough to hold all of the stored
1838 values' indices.

1839 `values` (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
1840 the stored values whose type is compatible with `D(v)`.

1841 `n` (INOUT) Pointer to a value indicating (on input) the number of elements the
1842 `values` and `indices` arrays can hold. Upon return, it will contain the number of
1843 values written to the arrays.

1844 `v` (IN) An existing GraphBLAS vector.

1845 Return Values

1846 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
1847 cessfully. This indicates that the compatibility tests on the input
1848 argument passed successfully, and the output arrays, `indices` and
1849 `values`, have been computed.

1850 `GrB_PANIC` Unknown internal error.

1851 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1852 GraphBLAS objects (input or output) is in an invalid state caused
1853 by a previous execution error. Call `GrB_error()` to access any error
1854 messages generated by the implementation.

1855 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1856 GrB_INSUFFICIENT_SPACE Not enough space in indices and values (as indicated by the n parameter) to hold all of the tuples that will be extracted.

1857

1858 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to Vector_new or Vector_dup.

1859

1860 GrB_NULL_POINTER indices, values, or n pointer is NULL.

1861 GrB_DOMAIN_MISMATCH The domains of the v vector or values array are incompatible with one another.

1862

1863 Description

1864 This method will extract all the tuples from the GraphBLAS vector v. The values associated with those tuples are placed in the values array and the indices are placed in the indices array. Both indices and values must be pre-allocated by the user to have enough space to hold at least GrB_Vector_nvals(v) elements before calling this function.

1866

1867

1868 Upon return of this function, n will be set to the number of values (and indices) copied. Also, the entries of indices are unique, but not necessarily sorted. Each tuple (i, v_i) in v is unzipped and copied into a distinct kth location in output vectors:

1869

1870

$$\{\text{indices}[k], \text{values}[k]\} \leftarrow (i, v_i),$$

1871 where $0 \leq k < \text{GrB_Vector_nvals}(v)$. No gaps in output vectors are allowed; that is, if indices[k] and values[k] exist upon return, so does indices[j] and values[j] for all j such that $0 \leq j < k$.

1872

1873 Note that if the value in n on input is less than the number of values contained in the vector v, then a GrB_INSUFFICIENT_SPACE error is returned because it is undefined which subset of values would be extracted otherwise.

1874

1875

1876 In both GrB_BLOCKING mode GrB_NONBLOCKING mode if the method exits with return value GrB_SUCCESS, the new contents of the arrays indices and values are as defined above.

1877

1878 4.2.4 Matrix methods

1879 4.2.4.1 Matrix_new: Construct new matrix

1880 Creates a new matrix with specified domain and dimensions.

1881 C Syntax

```
1882        GrB_Info GrB_Matrix_new(GrB_Matrix *A,
1883                                GrB_Type    d,
```

1884 GrB_Index nrows,
1885 GrB_Index ncols);

1886 **Parameters**

1887 **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1888 matrix.

1889 **d** (IN) The type corresponding to the domain of the matrix being created. Can be
1890 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1891 GraphBLAS type.

1892 **nrows** (IN) The number of rows of the matrix being created.

1893 **ncols** (IN) The number of columns of the matrix being created.

1894 **Return Values**

1895 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1896 blocking mode, this indicates that the API checks for the input ar-
1897 guments passed successfully. Either way, output matrix **A** is ready
1898 to be used in the next method of the sequence.

1899 **GrB_PANIC** Unknown internal error.

1900 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1901 GraphBLAS objects (input or output) is in an invalid state caused
1902 by a previous execution error. Call `GrB_error()` to access any error
1903 messages generated by the implementation.

1904 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1905 **GrB_UNINITIALIZED_OBJECT** The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`
1906 (needed for user-defined types).

1907 **GrB_NULL_POINTER** The **A** pointer is NULL.

1908 **GrB_INVALID_VALUE** **nrows** or **ncols** is zero or outside the range of the type `GrB_Index`.

1909 **Description**

1910 Creates a new matrix **A** of domain **D**(**d**), size **nrows** × **ncols**, and empty **L**(**A**). The method returns
1911 a handle to the new matrix in **A**.

1912 It is not an error to call this method more than once on the same variable; however, the handle to
1913 the previously created object will be overwritten.

1914 **4.2.4.2 Matrix_dup: Construct a copy of a GraphBLAS matrix**

1915 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

1916 **C Syntax**

```
1917         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
1918                               const GrB_Matrix A);
```

1919 **Parameters**

1920 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1921 matrix.

1922 A (IN) The GraphBLAS matrix to be duplicated.

1923 **Return Values**

1924 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1925 blocking mode, this indicates that the API checks for the input
1926 arguments passed successfully. Either way, output matrix C is ready
1927 to be used in the next method of the sequence.

1928 GrB_PANIC Unknown internal error.

1929 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1930 GraphBLAS objects (input or output) is in an invalid state caused
1931 by a previous execution error. Call GrB_error() to access any error
1932 messages generated by the implementation.

1933 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1934 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
1935 any matrix constructor.

1936 GrB_NULL_POINTER The C pointer is NULL.

1937 **Description**

1938 Creates a new matrix **C** of domain **D(A)**, size **nrows(A) × ncols(A)**, and contents **L(A)**. It returns
1939 a handle to it in **C**.

1940 It is not an error to call this method more than once on the same variable; however, the handle to
1941 the previously created object will be overwritten.

1942 **4.2.4.3 Matrix_diag: Construct a diagonal GraphBLAS matrix**

1943 Creates a new matrix with the same domain and contents as a `GrB_Vector`, and square dimensions
1944 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

1945 **C Syntax**

```
1946         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
1947                                 const GrB_Vector v,  
1948                                 int64_t         k);
```

1949 **Parameters**

1950 `C` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1951 matrix. The matrix is square with each dimension equal to $\text{size}(v) + |k|$.

1952 `v` (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the
1953 matrix.

1954 `k` (IN) The diagonal to which the vector is assigned. $k = 0$ represents the main
1955 diagonal, $k > 0$ is above the main diagonal, and $k < 0$ is below.

1956 **Return Values**

1957 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1958 blocking mode, this indicates that the API checks for the input
1959 arguments passed successfully. Either way, output matrix `C` is ready
1960 to be used in the next method of the sequence.

1961 `GrB_PANIC` Unknown internal error.

1962 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1963 GraphBLAS objects (input or output) is in an invalid state caused
1964 by a previous execution error. Call `GrB_error()` to access any error
1965 messages generated by the implementation.

1966 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

1967 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS vector, `v`, has not been initialized by a call to
1968 `Vector_new` or `Vector_dup`.

1969 `GrB_NULL_POINTER` The `C` pointer is `NULL`.

1970 **Description**

1971 Creates a new matrix **C** of domain **D(v)**, size $(\mathbf{size}(v) + |k|) \times (\mathbf{size}(v) + |k|)$, and contents

1972
$$\mathbf{L}(C) = \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(v)\} \text{ if } k \geq 0 \text{ or}$$

1973
$$\mathbf{L}(C) = \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(v)\} \text{ if } k < 0.$$

1974 It returns a handle to it in **C**. It is not an error to call this method more than once on the same
1975 variable; however, the handle to the previously created object will be overwritten.

1976 **4.2.4.4 Matrix_resize: Resize a matrix**

1977 Changes the dimensions of an existing matrix.

1978 **C Syntax**

```
1979     GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
1980                               GrB_Index nrows,  
1981                               GrB_Index ncols);
```

1982 **Parameters**

1983 **C** (INOUT) An existing Matrix object that is being resized.

1984 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the
1985 current number of rows.

1986 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than
1987 the current number of columns.

1988 **Return Values**

1989 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1990 blocking mode, this indicates that the API checks for the input
1991 arguments passed successfully. Either way, output matrix **C** is ready
1992 to be used in the next method of the sequence.

1993 **GrB_PANIC** Unknown internal error.

1994 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1995 GraphBLAS objects (input or output) is in an invalid state caused
1996 by a previous execution error. Call **GrB_error()** to access any error
1997 messages generated by the implementation.

1998 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1999 GrB_NULL_POINTER The C pointer is NULL.

2000 GrB_INVALID_VALUE nrows or ncols is zero or outside the range of the type GrB_Index.

2001 **Description**

2002 Changes the number of rows and columns of **C** to **nrows** and **ncols**, respectively. The domain **D(C)**
2003 of matrix **C** remains the same. The contents **L(C)** are modified as described below.

2004 Let $C = \langle \mathbf{D}(C), M, N, \mathbf{L}(C) \rangle$ when the method is called. When the method returns **C** is modified
2005 to $C = \langle \mathbf{D}(C), \text{nrows}, \text{ncols}, \mathbf{L}'(C) \rangle$ where $\mathbf{L}'(C) = \{(i, j, C_{ij}) : (i, j, C_{ij}) \in \mathbf{L}(C) \wedge (i < \text{nrows}) \wedge (j < \text{ncols})\}$. That is, all elements of **C** with row index greater than or equal to **nrows** or column index
2006 greater than or equal to **ncols** are dropped.
2007

2008 **4.2.4.5 Matrix_clear: Clear a matrix**

2009 Removes all elements (tuples) from a matrix.

2010 **C Syntax**

2011 GrB_Info GrB_Matrix_clear(GrB_Matrix A);

2012 **Parameters**

2013 A (IN) An existing GraphBLAS matrix to clear.

2014 **Return Values**

2015 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2016 blocking mode, this indicates that the API checks for the input argu-
2017 ments passed successfully. Either way, output matrix **A** is ready
2018 to be used in the next method of the sequence.

2019 GrB_PANIC Unknown internal error.

2020 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2021 GraphBLAS objects (input or output) is in an invalid state caused
2022 by a previous execution error. Call GrB_error() to access any error
2023 messages generated by the implementation.

2024 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2025 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, **A**, has not been initialized by a call to
2026 any matrix constructor.

2027 **Description**

2028 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,
2029 $\mathbf{L}(\mathbf{A}) = \emptyset$. The dimensions of the matrix do not change.

2030 **4.2.4.6 Matrix_nrows: Number of rows in a matrix**

2031 Retrieve the number of rows in a matrix.

2032 **C Syntax**

```
2033         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2034                                 const GrB_Matrix A);
```

2035 **Parameters**

2036 nrows (OUT) On successful return, contains the number of rows in the matrix.

2037 A (IN) An existing GraphBLAS matrix being queried.

2038 **Return Values**

2039 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2040 cessfully and the value of `nrows` has been set.

2041 GrB_PANIC Unknown internal error.

2042 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2043 GraphBLAS objects (input or output) is in an invalid state caused
2044 by a previous execution error. Call `GrB_error()` to access any error
2045 messages generated by the implementation.

2046 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
2047 any matrix constructor.

2048 GrB_NULL_POINTER `nrows` pointer is NULL.

2049 **Description**

2050 Return `nrows(A)` in `nrows` (the number of rows).

2051 **4.2.4.7 Matrix_ncols: Number of columns in a matrix**

2052 Retrieve the number of columns in a matrix.

2053 **C Syntax**

```
2054     GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2055                               const GrB_Matrix A);
```

2056 **Parameters**

2057 ncols (OUT) On successful return, contains the number of columns in the matrix.

2058 A (IN) An existing GraphBLAS matrix being queried.

2059 **Return Values**

2060 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2061 cessfully and the value of ncols has been set.

2062 GrB_PANIC Unknown internal error.

2063 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2064 GraphBLAS objects (input or output) is in an invalid state caused
2065 by a previous execution error. Call GrB_error() to access any error
2066 messages generated by the implementation.

2067 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2068 any matrix constructor.

2069 GrB_NULL_POINTER ncols pointer is NULL.

2070 **Description**

2071 Return `ncols(A)` in `ncols` (the number of columns).

2072 **4.2.4.8 Matrix_nvals: Number of stored elements in a matrix**

2073 Retrieve the number of stored elements (tuples) in a matrix.

2074 **C Syntax**

```
2075     GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2076                               const GrB_Matrix A);
```

2077 **Parameters**

2078 nvals (OUT) On successful return, contains the number of stored elements (tuples) in
2079 the matrix.

2080 A (IN) An existing GraphBLAS matrix being queried.

2081 **Return Values**

2082 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2083 cessfully and the value of nvals has been set.

2084 GrB_PANIC Unknown internal error.

2085 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2086 GraphBLAS objects (input or output) is in an invalid state caused
2087 by a previous execution error. Call GrB_error() to access any error
2088 messages generated by the implementation.

2089 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2090 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2091 any matrix constructor.

2092 GrB_NULL_POINTER The nvals pointer is NULL.

2093 **Description**

2094 Return nvals(A) in nvals. This is the number of tuples stored in matrix A, which is the size of
2095 $L(A)$ (see Section 3.5.3).

2096 **4.2.4.9 Matrix_build: Store elements from tuples into a matrix**

2097 **C Syntax**

```
GrB_Info GrB_Matrix_build(GrB_Matrix      C,  
                           const GrB_Index *row_indices,  
                           const GrB_Index *col_indices,  
                           const <type>   *values,  
                           GrB_Index      n,  
                           const GrB_BinaryOp dup);
```

2098 **Parameters**

2099 C (INOUT) An existing Matrix object to store the result.

2100 `row_indices` (IN) Pointer to an array of row indices.

2101 `col_indices` (IN) Pointer to an array of column indices.

2102 `values` (IN) Pointer to an array of scalars of a type that is compatible with the domain of
2103 matrix, `C`.

2104 `n` (IN) The number of entries contained in each array (the same for `row_indices`,
2105 `col_indices`, and `values`).

2106 `dup` (IN) An associative and commutative binary operator to apply when duplicate
2107 values for the same location are present in the input arrays. All three domains of
2108 `dup` must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If `dup` is `GrB_NULL`,
2109 then duplicate locations will result in an error.

2110 Return Values

2111 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
2112 blocking mode, this indicates that the API checks for the input
2113 arguments passed successfully. Either way, output matrix `C` is
2114 ready to be used in the next method of the sequence.

2115 `GrB_PANIC` Unknown internal error.

2116 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the
2117 opaque GraphBLAS objects (input or output) is in an invalid
2118 state caused by a previous execution error. Call `GrB_error()` to
2119 access any error messages generated by the implementation.

2120 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

2121 `GrB_UNINITIALIZED_OBJECT` Either `C` has not been initialized by a call to any matrix construc-
2122 tor, or `dup` has not been initialized by a call to `GrB_BinaryOp_new`.

2123 `GrB_NULL_POINTER` `row_indices`, `col_indices` or `values` pointer is `NULL`.

2124 `GrB_INDEX_OUT_OF_BOUNDS` A value in `row_indices` or `col_indices` is outside the allowed range
2125 for `C`.

2126 `GrB_DOMAIN_MISMATCH` Either the domains of the GraphBLAS binary operator `dup` are
2127 not all the same, or the domains of `values` and `C` are incompatible
2128 with each other or D_{dup} .

2129 `GrB_OUTPUT_NOT_EMPTY` Output matrix `C` already contains valid tuples (elements). In
2130 other words, `GrB_Matrix_nvals(C)` returns a positive value.

2131 `GrB_INVALID_VALUE` `indices` contains a duplicate location and `dup` is `GrB_NULL`.

2132 **Description**

2133 If `dup` is not `GrB_NULL`, an internal matrix $\tilde{\mathbf{C}} = \langle D_{dup}, \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$ is created, which
2134 only differs from \mathbf{C} in its domain; otherwise, $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$.

2135 Each tuple $\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the
2136 output in the form of

$$2137 \quad \tilde{\mathbf{C}}(\text{row_indices}[k], \text{col_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2138 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,
2139 `dup` is used to reduce the values before assignment into $\tilde{\mathbf{C}}$ as follows:

$$2140 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row_indices}[k]=i \wedge \text{col_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2141 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{C}}$ is copied into \mathbf{C} via typecasting its
2142 values to $\mathbf{D}(\mathbf{C})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

2143 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2144 It is an error to call this function on an output object with existing elements. In other words,
2145 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2146 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,
2147 `col_indices`, or `values`.

2148 **4.2.4.10 Matrix_setElement: Set a single element in matrix**

2149 Set one element of a matrix to a given value.

2150 **C Syntax**

```
2151 // scalar value
2152 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2153                               <type>         val,
2154                               GrB_Index       row_index,
2155                               GrB_Index       col_index);
2156
2157 // GraphBLAS scalar
2158 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2159                               const GrB_Scalar s,
2160                               GrB_Index       row_index,
2161                               GrB_Index       col_index);
```

2162 **Parameters**

2163 C (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.
2164 val or s (IN) Scalar to assign. Its domain (type) must be compatible with the domain of
2165 C.
2166 row_index (IN) Row index of element to be assigned
2167 col_index (IN) Column index of element to be assigned

2168 **Return Values**

2169 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2170 blocking mode, this indicates that the compatibility tests on in-
2171 dex/dimensions and domains for the input arguments passed suc-
2172 cessfully. Either way, the output matrix C is ready to be used in
2173 the next method of the sequence.
2174 GrB_PANIC Unknown internal error.
2175 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2176 GraphBLAS objects (input or output) is in an invalid state caused
2177 by a previous execution error. Call GrB_error() to access any error
2178 messages generated by the implementation.
2179 GrB_OUT_OF_MEMORY Not enough memory available for operation.
2180 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, or GraphBLAS scalar, s, has not been
2181 initialized by a call to a respective constructor.
2182 GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e., not less
2183 than **nrows(C)** or **ncols(C)**, respectively).
2184 GrB_DOMAIN_MISMATCH The domains of the matrix and the scalar are incompatible.

2185 **Description**

2186 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or
2187 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from
2188 one domain can be cast to values in the other domain as per the rules of the C language. In
2189 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
2190 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
2191 **GrB_Matrix_setElement** ends and the domain mismatch error listed above is returned.

2192 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2193 \quad & 0 \leq \text{row_index} < \mathbf{nrows(C)}, \\ & 0 \leq \text{col_index} < \mathbf{ncols(C)} \end{aligned}$$

2194 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid
2195 index error listed above is returned.

We are now ready to carry out the assignment; that is:

$$C(\text{row_index}, \text{col_index}) = \begin{cases} \mathbf{L}(s), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

2196 In the case of a transparent scalar or if $\mathbf{L}(s)$ is not empty, then a value will be stored at the
2197 specified location in C , overwriting any value that may have been stored there before. In the case
2198 of a GraphBLAS scalar and if $\mathbf{L}(s)$ is empty, then any value stored at the specified location in C
2199 will be removed.

2200 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
2201 of C is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
2202 return value `GrB_SUCCESS` and the new content of vector C is as defined above but may not be
2203 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2204 4.2.4.11 `Matrix_removeElement`: Remove an element from a matrix

2205 Remove (annihilate) one stored element from a matrix.

2206 C Syntax

```
2207     GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,  
2208                                     GrB_Index   row_index,  
2209                                     GrB_Index   col_index);
```

2210 Parameters

2211 `C` (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2212 `row_index` (IN) Row index of element to be removed

2213 `col_index` (IN) Column index of element to be removed

2214 Return Values

2215 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
2216 blocking mode, this indicates that the compatibility tests on in-
2217 dex/dimensions and domains for the input arguments passed suc-
2218 cessfully. Either way, the output matrix C is ready to be used in
2219 the next method of the sequence.

2220 `GrB_PANIC` Unknown internal error.

2221 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2222 GraphBLAS objects (input or output) is in an invalid state caused
2223 by a previous execution error. Call GrB_error() to access any error
2224 messages generated by the implementation.

2225 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2226 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to
2227 any matrix constructor.

2228 GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e., not less
2229 than **nrows(C)** or **ncols(C)**, respectively).

2230 Description

2231 First, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned}
 &0 \leq \text{row_index} < \mathbf{nrows(C)}, \\
 &0 \leq \text{col_index} < \mathbf{ncols(C)}
 \end{aligned}$$

2233 If either of these conditions is violated, execution of GrB_Matrix_removeElement ends and the
2234 invalid index error listed above is returned.

2235 We are now ready to carry out the removal of a value that may be stored at the location specified by
2236 (row_index, col_index). If a value does not exist at the specified location in C, no error is reported
2237 and the operation has no effect on the state of C. In either case, the following will be true on return
2238 from this method: (row_index, col_index) \notin **ind(C)**

2239 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
2240 of C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
2241 return value GrB_SUCCESS and the new content of vector C is as defined above but may not be
2242 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2243 4.2.4.12 Matrix_extractElement: Extract a single element from a matrix

2244 Extract one element of a matrix into a scalar.

2245 C Syntax

```

2246           // scalar value
2247           GrB_Info GrB_Matrix_extractElement(<type>           *val,
2248                                           const GrB_Matrix    A,
2249                                           GrB_Index           row_index,
2250                                           GrB_Index           col_index);
2251
2252           // GraphBLAS scalar

```

```

2253         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2254                                           const GrB_Matrix A,
2255                                           GrB_Index      row_index,
2256                                           GrB_Index      col_index);
2257

```

2258 Parameters

2259 `val` or `s` (INOUT) An existing scalar whose domain is compatible with the domain of matrix
2260 `A`. On successful return, this scalar holds the result of the extract. Any previous
2261 value stored in `val` or `s` is overwritten.

2262 `A` (IN) The GraphBLAS matrix from which an element is extracted.

2263 `row_index` (IN) The row index of location in `A` to extract.

2264 `col_index` (IN) The column index of location in `A` to extract.

2265 Return Values

2266 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
2267 cessfully. This indicates that the compatibility tests on dimensions
2268 and domains for the input arguments passed successfully, and the
2269 output scalar, `val` or `s`, has been computed and is ready to be used
2270 in the next method of the sequence.

2271 `GrB_NO_VALUE` When using the transparent scalar, `val`, this is returned when there
2272 is no stored value at specified location.

2273 `GrB_PANIC` Unknown internal error.

2274 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
2275 GraphBLAS objects (input or output) is in an invalid state caused
2276 by a previous execution error. Call `GrB_error()` to access any error
2277 messages generated by the implementation.

2278 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

2279 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS matrix, `A`, or scalar, `s`, has not been initialized by
2280 a call to a corresponding constructor.

2281 `GrB_NULL_POINTER` `val` pointer is NULL.

2282 `GrB_INVALID_INDEX` `row_index` or `col_index` is outside the allowable range (i.e. less than
2283 zero or greater than or equal to `nrows(A)` or `ncols(A)`, respec-
2284 tively).

2285 `GrB_DOMAIN_MISMATCH` The domains of the matrix and scalar are incompatible.

2286 **Description**

2287 First, the scalar and input matrix are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$
2288 must be compatible with $\mathbf{D}(A)$. Two domains are compatible with each other if values from
2289 one domain can be cast to values in the other domain as per the rules of the C language. In
2290 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
2291 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
2292 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2293 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2294 \quad & 0 \leq \text{row_index} < \mathbf{nrows}(A), \\ & 0 \leq \text{col_index} < \mathbf{ncols}(A) \end{aligned}$$

2295 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index
2296 error listed above is returned.

We are now ready to carry out the extract into the output scalar; that is,

$$\left. \begin{array}{l} \mathbf{L}(s) \\ \text{val} \end{array} \right\} = A(\text{row_index}, \text{col_index})$$

2297 If $(\text{row_index}, \text{col_index}) \in \mathbf{ind}(A)$, then the corresponding value from A is copied into s or val
2298 with casting as necessary. If $(\text{row_index}, \text{col_index}) \notin \mathbf{ind}(A)$, then one of the follow occurs
2299 depending on output scalar type:

- 2300 • The GraphBLAS scalar, s , is cleared and `GrB_SUCCESS` is returned.
- 2301 • The non-opaque scalar, val , is unchanged, and `GrB_NO_VALUE` is returned.

2302 When using the non-opaque scalar variant (val) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
2303 mode, the new contents of val are as defined above if the method exits with return value `GrB_SUCCESS`
2304 or `GrB_NO_VALUE`.

2305 When using the GraphBLAS scalar variant (s) with a `GrB_SUCCESS` return value, the method
2306 exits and the new contents of s is as defined above and fully computed in `GrB_BLOCKING` mode.
2307 In `GrB_NONBLOCKING` mode, the new contents of s is as defined above but may not be fully
2308 computed; however, it can be used in the next GraphBLAS method call in a sequence.

2309 **4.2.4.13 Matrix_extractTuples: Extract tuples from a matrix**

2310 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

2311 **C Syntax**

```
2312     GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,  
2313                                     GrB_Index      *col_indices,
```

```

2314                                     <type>           *values,
2315                                     GrB_Index        *n,
2316                                     const GrB_Matrix  A);

```

2317 Parameters

2318 `row_indices` (OUT) Pointer to an array of row indices that is large enough to hold all of the
2319 row indices.

2320 `col_indices` (OUT) Pointer to an array of column indices that is large enough to hold all of the
2321 column indices.

2322 `values` (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
2323 the stored values whose type is compatible with $\mathbf{D}(\mathbf{A})$.

2324 `n` (INOUT) Pointer to a value indicating (in input) the number of elements the `values`,
2325 `row_indices`, and `col_indices` arrays can hold. Upon return, it will contain the
2326 number of values written to the arrays.

2327 `A` (IN) An existing GraphBLAS matrix.

2328 Return Values

2329 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
2330 cessfully. This indicates that the compatibility tests on the input
2331 argument passed successfully, and the output arrays, `indices` and
2332 `values`, have been computed.

2333 `GrB_PANIC` Unknown internal error.

2334 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
2335 GraphBLAS objects (input or output) is in an invalid state caused
2336 by a previous execution error. Call `GrB_error()` to access any error
2337 messages generated by the implementation.

2338 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

2339 `GrB_INSUFFICIENT_SPACE` Not enough space in `row_indices`, `col_indices`, and `values` (as indi-
2340 cated by the `n` parameter) to hold all of the tuples that will be
2341 extracted.

2342 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS matrix, `A`, has not been initialized by a call to
2343 any matrix constructor.

2344 `GrB_NULL_POINTER` `row_indices`, `col_indices`, `values` or `n` pointer is `NULL`.

2345 `GrB_DOMAIN_MISMATCH` The domains of the `A` matrix and `values` array are incompatible
2346 with one another.

2347 **Description**

2348 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with
2349 those tuples are placed in the **values** array, the column indices are placed in the **col_indices** array,
2350 and the row indices are placed in the **row_indices** array. These output arrays are pre-allocated by
2351 the user before calling this function such that each output array has enough space to hold at least
2352 **GrB_Matrix_nvals(A)** elements.

2353 Upon return of this function, a pair of $\{\text{row_indices}[k], \text{col_indices}[k]\}$ are unique for every valid
2354 k , but they are not required to be sorted in any particular order. Each tuple (i, j, A_{ij}) in **A** is
2355 unzipped and copied into a distinct k th location in output vectors:

$$\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\} \leftarrow (i, j, A_{ij}),$$

2356 where $0 \leq k < \text{GrB_Matrix_nvals}(v)$. No gaps in output vectors are allowed; that is, if **row_indices**[k],
2357 **col_indices**[k] and **values**[k] exist upon return, so does **row_indices**[j], **col_indices**[j] and **values**[j] for
2358 all j such that $0 \leq j < k$.

2359 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,
2360 then a **GrB_INSUFFICIENT_SPACE** error is returned since it is undefined which subset of values
2361 would be extracted.

2362 In both **GrB_BLOCKING** mode **GrB_NONBLOCKING** mode if the method exits with return value
2363 **GrB_SUCCESS**, the new contents of the arrays **row_indices**, **col_indices** and **values** are as defined
2364 above.

2365 **4.2.4.14 Matrix_exportHint: Provide a hint as to which storage format might be most**
2366 **efficient for exporting a matrix**

2367 **C Syntax**

```
GrB_Info GrB_Matrix_exportHint(GrB_Format          *hint,  
                               GrB_Matrix         A);
```

2368 **Parameters**

2369 **hint** (OUT) Pointer to a value of type **GrB_Format**.

2370 **A** (IN) A GraphBLAS matrix object.

2371 **Return Values**

2372 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2373 cessfully and the value of **hint** has been set.

2374 **GrB_PANIC** Unknown internal error.

2375 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
2376 opaque GraphBLAS objects (input or output) is in an invalid
2377 state caused by a previous execution error. Call GrB_error() to
2378 access any error messages generated by the implementation.

2379 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2380 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2381 any matrix constructor.

2382 GrB_NULL_POINTER hint is NULL.

2383 GrB_NO_VALUE If the implementation does not have a preferred format, it may
2384 return the value GrB_NO_VALUE.

2385 Description

2386 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for
2387 exporting the matrix A. GraphBLAS implementations might return the current storage format of
2388 the matrix, or the format to which it could most efficiently be exported. However, implementations
2389 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is
2390 free to refuse to provide a format hint, returning GrB_NO_VALUE.

2391 4.2.4.15 Matrix_exportSize: Return the array sizes necessary to export a GraphBLAS 2392 matrix object

2393 C Syntax

```
GrB_Info GrB_Matrix_exportSize(GrB_Index                   *n_indptr,  
                                  GrB_Index               *n_indices,  
                                  GrB_Index               *n_values,  
                                  GrB_Format              format,  
                                  GrB_Matrix              A);
```

2394 Parameters

2395 n_indptr (OUT) Pointer to a value of type GrB_Index.

2396 n_indices (OUT) Pointer to a value of type GrB_Index.

2397 n_values (OUT) Pointer to a value of type GrB_Index.

2398 format (IN) a value indicating the format in which the matrix will be exported, as defined
2399 in Section 3.5.3.1.

2400 A (IN) A GraphBLAS matrix object.

2401 **Return Values**

2402 GrB_SUCCESS In blocking mode or non-blocking mode, the operation com-
2403 pleted successfully. This indicates that the API checks for the
2404 input arguments passed successfully, and the number of elements
2405 necessary for the export buffers have been written to `n_indptr`,
2406 `n_indices`, and `n_values`, respectively.

2407 GrB_PANIC Unknown internal error.

2408 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
2409 opaque GraphBLAS objects (input or output) is in an invalid
2410 state caused by a previous execution error. Call `GrB_error()` to
2411 access any error messages generated by the implementation.

2412 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2413 GrB_UNINITIALIZED_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to
2414 any matrix constructor.

2415 GrB_NULL_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

2416 **Description**

2417 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to
2418 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and
2419 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be
2420 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user
2421 as one of the values defined in Section 3.5.3.1.

2422 **4.2.4.16 Matrix_export: Export a GraphBLAS matrix to a pre-defined format**

2423 **C Syntax**

```
GrB_Info GrB_Matrix_export(GrB_Index          *indptr,  
                           GrB_Index          *indices,  
                           <type>           *values,  
                           GrB_Index          *n_indptr,  
                           GrB_Index          *n_indices,  
                           GrB_Index          *n_values,  
                           GrB_Format         format,  
                           GrB_Matrix        A);
```

2424 Parameters

- 2425 **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-
2426 dices, depending on the value of **format**. It must be large enough to hold at
2427 least **n_indptr** elements of type **GrB_Index**, where **n_indices** was returned from
2428 **GrB_Matrix_exportSize()** method.
- 2429 **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements
2430 in **values**, depending on the value of **format**. It must be large enough to hold at
2431 least **n_indices** elements of type **GrB_Index**, where **n_indices** was returned from
2432 **GrB_Matrix_exportSize()** method.
- 2433 **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-
2434 ment must match the type of the values stored in **A**. It must be large enough
2435 to hold at least **n_values** elements of that type, where **n_values** was returned from
2436 **GrB_Matrix_exportSize**.
- 2437 **n_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**
2438 array can hold. Upon return, it will contain the number of elements written to the
2439 array.
- 2440 **n_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**
2441 array can hold. Upon return, it will contain the number of elements written to the
2442 array.
- 2443 **n_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**
2444 array can hold. Upon return, it will contain the number of elements written to the
2445 array.
- 2446 **format** (IN) a value indicating the format in which the matrix will be exported, as defined
2447 in Section 3.5.3.1.
- 2448 **A** (IN) A GraphBLAS matrix object.

2449 Return Values

- 2450 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2451 cessfully. This indicates that the compatibility tests on the input
2452 argument passed successfully, and the output arrays, **indptr**, **in-**
2453 **dices** and **values**, have been computed.
- 2454 **GrB_PANIC** Unknown internal error.
- 2455 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2456 opaque GraphBLAS objects (input or output) is in an invalid
2457 state caused by a previous execution error. Call **GrB_error()** to
2458 access any error messages generated by the implementation.
- 2459 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2460 GrB_INSUFFICIENT_SPACE Not enough space in `indptr`, `indices`, and/or `values` (as indicated
2461 by the corresponding `n_*` parameter) to hold all of the corre-
2462 sponding elements that will be extracted.

2463 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
2464 any matrix constructor.

2465 GrB_NULL_POINTER `indptr`, `indices`, `values` `n_indptr`, `n_indices`, `n_values` pointer is
2466 NULL.

2467 GrB_DOMAIN_MISMATCH The domain of `A` does not match with the type of `values`.

2468 Description

2469 Given a matrix `A`, this method exports the contents of the matrix into one of the pre-defined
2470 `GrB_Format` formats from Section 3.5.3.1. The user-allocated arrays pointed to by `indptr`, `indices`,
2471 and `values` must be at least large enough to hold the corresponding number of elements returned
2472 by calling `GrB_Matrix_exportSize`. The value of `format` can be chosen arbitrarily, but a call to
2473 `GrB_Matrix_exportHint` may suggest a format that results in the most efficient export. Details
2474 of the contents of `indptr`, `indices`, and `values` corresponding to each supported format is given in
2475 Appendix B.

2476 4.2.4.17 Matrix_import: Import a matrix into a GraphBLAS object

2477 C Syntax

```

GrB_Info GrB_Matrix_import(GrB_Matrix      *A,
                          GrB_Type        d,
                          GrB_Index       nrows,
                          GrB_Index       ncols
                          const GrB_Index *indptr,
                          const GrB_Index *indices,
                          const <type>   *values,
                          GrB_Index       n_indptr,
                          GrB_Index       n_indices,
                          GrB_Index       n_values,
                          GrB_Format      format);

```

2478 Parameters

2479 `A` (INOUT) On a successful return, contains a handle to the newly created Graph-
2480 BLAS matrix.

2481 `d` (IN) The type corresponding to the domain of the matrix being created. Can be
2482 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
2483 GraphBLAS type.

2484 `nrows` (IN) Integer value holding the number of rows in the matrix.

2485 `ncols` (IN) Integer value holding the number of columns in the matrix.

2486 `indptr` (IN) Pointer to an array of row or column offsets, or row indices, depending on the
2487 value of `format`.

2488 `indices` (IN) Pointer to an array row or column indices of the elements in `values`, depending
2489 on the value of `format`.

2490 `values` (IN) Pointer to an array of values. Type must match the type of `d`.

2491 `n_indptr` (IN) Integer value holding the number of elements in the array pointed to by `indptr`.

2492 `n_indices` (IN) Integer value holding the number of elements in the array pointed to by `indices`.

2493 `n_values` (IN) Integer value holding the number of elements in the array pointed to by `values`.

2494 `format` (IN) a value indicating the format of the matrix being imported, as defined in
2495 Section 3.5.3.1.

2496 **Return Values**

2497 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
2498 blocking mode, this indicates that the API checks for the input
2499 arguments passed successfully and the input arrays have been
2500 consumed. Either way, output matrix `A` is ready to be used in
2501 the next method of the sequence.

2502 `GrB_PANIC` Unknown internal error.

2503 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

2504 `GrB_UNINITIALIZED_OBJECT` The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`
2505 (needed for user-defined types).

2506 `GrB_NULL_POINTER` `A`, `indptr`, `indices` or `values` pointer is `NULL`.

2507 `GrB_INDEX_OUT_OF_BOUNDS` A value in `indptr` or `indices` is outside the allowed range for indices
2508 in `A` and or the size of `values`, `n_values`, depending on the value
2509 of `format`.

2510 `GrB_INVALID_VALUE` `nrows` or `ncols` is zero or outside the range of the type `GrB_Index`.

2511 `GrB_DOMAIN_MISMATCH` The domain given in parameter `d` does not match the element
2512 type of `values`.

2513 **Description**

2514 Creates a new matrix **A** of domain **D**(d) and dimension `nrows` × `ncols`. The new GraphBLAS
2515 matrix will be filled with the contents of the matrix pointed to by `indptr`, and `indices`, and `values`.
2516 The method returns a handle to the new matrix in **A**. The structure of the data being imported is
2517 defined by `format`, which must be equal to one of the values defined in Section 3.5.3.1. Details of
2518 the contents of `indptr`, `indices` and `values` for each supported format is given in Appendix B.

2519 It is not an error to call this method more than once on the same output matrix; however, the
2520 handle to the previously created object will be overwritten.

2521 **4.2.4.18 Matrix_serializeSize: Compute the serialize buffer size**

2522 Compute the buffer size (in bytes) necessary to serialize a `GrB_Matrix` using `GrB_Matrix_serialize`.

2523 **C Syntax**

```
GrB_Info GrB_Matrix_serializeSize(GrB_Index *size,  
                                GrB_Matrix A);
```

2524 **Parameters**

2525 `size` (OUT) Pointer to `GrB_Index` value where size in bytes of serialized object will be
2526 written.

2527 `A` (IN) A GraphBLAS matrix object.

2528 **Return Values**

2529 `GrB_SUCCESS` The operation completed successfully and the value pointed to
2530 by `*size` has been computed and is ready to use.

2531 `GrB_PANIC` Unknown internal error.

2532 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

2533 `GrB_NULL_POINTER` `size` is NULL.

2534 **Description**

2535 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object **A**.
2536 Users may then allocate a buffer of `size` bytes to pass as a parameter to `GrB_Matrix_serialize`.

2537 **4.2.4.19 Matrix_serialize: Serialize a GraphBLAS matrix.**

2538 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2539 **C Syntax**

```
GrB_Info GrB_Matrix_serialize(void      *serialized_data,  
                               GrB_Index *serialized_size,  
                               GrB_Matrix A);
```

2540 **Parameters**

2541 `serialized_data` (INOUT) Pointer to the preallocated buffer where the serialized matrix will be
2542 written.

2543 `serialized_size` (INOUT) On input, the size in bytes of the buffer pointed to by `serialized_data`.
2544 On output, the number of bytes written to `serialized_data`.

2545 `A` (IN) A GraphBLAS matrix object.

2546 **Return Values**

2547 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
2548 cessfully. This indicates that the compatibility tests on the in-
2549 put argument passed successfully, and the output buffer `serial-
2550 ized_data` and `serialized_size`, have been computed and are ready
2551 to use.

2552 `GrB_PANIC` Unknown internal error.

2553 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the
2554 opaque GraphBLAS objects (input or output) is in an invalid
2555 state caused by a previous execution error. Call `GrB_error()` to
2556 access any error messages generated by the implementation.

2557 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

2558 `GrB_NULL_POINTER` `serialized_data` or `serialize_size` is NULL.

2559 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS matrix, `A`, has not been initialized by a call to
2560 any matrix constructor.

2561 `GrB_INSUFFICIENT_SPACE` The size of the buffer `serialized_data` (provided as an input `seri-
2562 alized_size`) was not large enough.

2563 **Description**

2564 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,
2565 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must
2566 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the
2567 serialized matrix written to `serialized_data` is provided upon completion as an output written to
2568 `serialized_size`.

2569 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created
2570 with one library implementation is not necessarily valid for deserialization with another implemen-
2571 tation.

2572 **4.2.4.20 Matrix_deserialize: Deserialize a GraphBLAS matrix.**

2573 Construct a new GraphBLAS matrix from a serialized object.

2574 **C Syntax**

```
GrB_Info GrB_Matrix_deserialize(GrB_Matrix *A,  
                                GrB_Type   d,  
                                const void *serialized_data,  
                                GrB_Index  serialized_size);
```

2575 **Parameters**

2576 `A` (INOUT) On a successful return, contains a handle to the newly created Graph-
2577 BLAS matrix.

2578 `d` (IN) the type of the matrix that was serialized in `serialized_data`.

2579 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2580 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

2581 **Return Values**

2582 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
2583 blocking mode, this indicates that the API checks for the input
2584 arguments passed successfully. Either way, output matrix `A` is
2585 ready to be used in the next method of the sequence.

2586 `GrB_PANIC` Unknown internal error.

2587 `GrB_INVALID_OBJECT` This is returned if `serialized_data` is invalid or corrupted.

2588 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

2589 GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to GrB_Type_new
2590 (needed for user-defined types).

2591 GrB_NULL_POINTER serialized_data or A is NULL.

2592 GrB_DOMAIN_MISMATCH The type given in d does not match the type of the matrix
2593 serialized in serialized_data.

2594 Description

2595 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object
2596 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The
2597 domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized
2598 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2599 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix
2600 serialized in one library implementation can be deserialized by another.

2601 It is not an error to call this method more than once on the same output matrix; however, the
2602 handle to the previously created object will be overwritten.

2603 4.2.5 Descriptor methods

2604 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-
2605 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

2606 4.2.5.1 Descriptor_new: Create new descriptor

2607 Creates a new (empty or default) descriptor.

2608 C Syntax

```
2609 GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);
```

2610 Parameters

2611 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2612 descriptor.

2613 Return Value

2614 GrB_SUCCESS The method completed successfully.

2615 GrB_PANIC unknown internal error.

2616 GrB_OUT_OF_MEMORY not enough memory available for operation.

2617 GrB_NULL_POINTER desc pointer is NULL.

2618 **Description**

2619 Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can
2620 be populated by calls to Descriptor_set.

2621 It is not an error to call this method more than once on the same variable; however, the handle to
2622 the previously created object will be overwritten.

2623 **4.2.5.2 Descriptor_set: Set content of descriptor**

2624 Sets the content for a field for an existing descriptor.

2625 **C Syntax**

```
2626 GrB_Info GrB_Descriptor_set(GrB_Descriptor desc,  
2627                             GrB_Desc_Field field,  
2628                             GrB_Desc_Value val);
```

2629 **Parameters**

2630 desc (IN) An existing GraphBLAS descriptor to be modified.

2631 field (IN) The field being set.

2632 val (IN) New value for the field being set.

2633 **Return Values**

2634 GrB_SUCCESS operation completed successfully.

2635 GrB_PANIC unknown internal error.

2636 GrB_OUT_OF_MEMORY not enough memory available for operation.

2637 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to new.

2638 GrB_INVALID_VALUE invalid value set on the field, or invalid field.

2639 Description

2640 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor
2641 to set the value associated with that field. Valid values for the `field` parameter include the following:

2642 `GrB_OUTP` refers to the output parameter (result) of the operation.

2643 `GrB_MASK` refers to the mask parameter of the operation.

2644 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2645 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2646 Valid values for the `val` parameter are:

2647 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask
2648 (`GrB_MASK`) parameter.

2649 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-
2650 eter. When combined with `GrB_STRUCTURE`, the complement of the
2651 structure of the mask is used without evaluating the values stored.

2652 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input
2653 matrix parameters only).

2654 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear
2655 the matrix first (or clear the non-masked entries). The default behavior
2656 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`
2657 parameter only.

2658 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,
2659 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).
2660 A value for a given field may be set multiple times but will have no additional effect. Fields that
2661 have no values set result in their default behavior, as defined in Section 3.6.

2662 4.2.6 free: Destroy an object and release its resources

2663 Destroys a previously created GraphBLAS object and releases any resources associated with the
2664 object.

2665 C Syntax

```
2666 GrB_Info GrB_free(<GrB_Object> *obj);
```

2667 **Parameters**

2668 obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have
2669 been created by an explicit call to a GraphBLAS constructor. It can be any of the
2670 opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,
2671 binary op, unary op, or type. On successful completion of GrB_free, obj behaves
2672 as an uninitialized object.

2673 **Return Values**

2674 GrB_SUCCESS operation completed successfully

2675 GrB_PANIC unknown internal error. If this return value is encountered when
2676 in nonblocking mode, the error responsible for the panic condition
2677 could be from any method involved in the computation of the input
2678 object. The GrB_error() method should be called for additional
2679 information.

2680 **Description**

2681 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime
2682 system. A call to GrB_free frees those resources so they are available for use by other GraphBLAS
2683 objects.

2684 The parameter passed into GrB_free is a handle referencing a GraphBLAS opaque object of a data
2685 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-
2686 structor. The behavior of a program that calls GrB_free on a pre-defined object is implementation
2687 defined.

2688 After the GrB_free method returns, the object referenced by the input handle is destroyed and the
2689 handle has the value GrB_INVALID_HANDLE. The handle can be used in subsequent GraphBLAS
2690 methods but only after the handle has been reinitialized with a call the the appropriate _new or
2691 _dup method.

2692 Note that unlike other GraphBLAS methods, calling GrB_free with an object with an invalid handle
2693 is legal. The system may attempt to free resources that might be associated with that object, if
2694 possible, and return normally.

2695 When using GrB_free it is possible to create a dangling reference to an object. This would occur
2696 when a handle is assigned to a second variable of the same opaque type. This creates two handles
2697 that reference the same object. If GrB_free is called with one of the variables, the object is destroyed
2698 and the handle associated with the other variable no longer references a valid object. This is not an
2699 error condition that the implementation of the GraphBLAS API can be expected to catch, hence
2700 programmers must take care to prevent this situation from occurring.

2701 **4.2.7 wait: Return once an object is either *complete* or *materialized***

2702 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2703 **C Syntax**

2704 `GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);`

2705 **Parameters**

2706 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an
2707 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2708 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2709 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another
2710 thread (completion) or all computing to produce `obj` by all GraphBLAS operations
2711 in its sequence have finished (materialization).

2712 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the
2713 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or
2714 `GrB_MATERIALIZE`.

2715 **Return values**

2716 `GrB_SUCCESS` operation completed successfully.

2717 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-
2718 pletion of pending operations.

2719 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion
2720 of pending operations.

2721 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,
2722 or other constructor, method.

2723 `GrB_PANIC` unknown internal error.

2724 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`
2725 `GrB_MATERIALIZE`.

2726 **Description**

2727 On successful return from `GrB_wait()`, the input object, `obj` is in one of two states depending on
2728 the mode of `GrB_wait`:

- 2729 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program
2730 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another
2731 thread. This result occurs when the mode parameter is set to `GrB_COMPLETE`.
- 2732 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on
2733 behalf of `obj` and error information is available. This result occurs when the mode parameter
2734 is set to `GrB_MATERIALIZE`.

2735 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,
2736 `GrB_wait(obj,mode)` has no effect when called in blocking mode.

2737 In non-blocking mode, the status of any pending method calls, other than those associated with pro-
2738 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to `GrB_wait(obj,mode)`.
2739 Methods in the sequence for `obj`, however, most likely would be impacted by a call to `GrB_wait(obj,mode)`;
2740 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be
2741 finished prior to the return from `GrB_wait(obj,mode)`.

2742 4.2.8 error: Retrieve an error string

2743 Retrieve an error-message about any errors encountered during the processing associated with an
2744 object.

2745 C Syntax

```
2746         GrB_Info GrB_error(const char          **error,
2747                           const GrB_Object    obj);
```

2748 Parameters

2749 `error` (OUT) A pointer to a null-terminated string. The contents of the string are im-
2750 plementation defined.

2751 `obj` (IN) An existing GraphBLAS object. The object must have been created by an
2752 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2753 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2754 or type.

2755 Return value

2756 `GrB_SUCCESS` operation completed successfully.

2757 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,
2758 or other constructor, method.

2759 `GrB_PANIC` unknown internal error.

2760 **Description**

2761 This method retrieves a message related to any errors that were encountered during the last Graph-
2762 BLAS method that had the opaque GraphBLAS object, `obj`, as an OUT or INOUT parameter.
2763 The function returns a pointer to a null-terminated string and the contents of that string are
2764 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error
2765 string. The string that is returned is owned by `obj` and will be valid until the next time `obj` is
2766 used as an OUT or INOUT parameter or the object is freed by a call to `GrB_free(obj)`. This is a
2767 thread-safe function. It can be safely called by multiple threads for the same object in a race-free
2768 program.

2769 **4.3 GraphBLAS operations**

2770 The GraphBLAS operations are defined in the GraphBLAS math specification and summarized in
2771 Table 4.1. In addition to methods that implement these fundamental GraphBLAS operations, we
2772 support a number of variants that have been found to be especially useful in algorithm development.
2773 A flowchart of the overall behavior of a GraphBLAS operation is shown in Figure 4.1.

2774 **Domains and Casting**

2775 A GraphBLAS operation is only valid when the domains of the GraphBLAS objects are mathemat-
2776 ically consistent. The C programming language defines implicit casts between built-in data types.
2777 For example, floats, doubles, and ints can be freely mixed according to the rules defined for implicit
2778 casts. It is the responsibility of the user to assure that these casts are appropriate for the algorithm
2779 in question. For example, a cast to int implies truncation of a floating point type. Depending on
2780 the operation, this truncation error could lead to erroneous results. Furthermore, casting a wider
2781 type onto a narrower type can lead to overflow errors. The GraphBLAS operations do not attempt
2782 to protect a user from these sorts of errors.

2783 When user-define types are involved, however, GraphBLAS requires strict equivalence between
2784 types and no casting is supported. If GraphBLAS detects these mismatches, it will return a
2785 domain mismatch error.

2786 **Dimensions and Transposes**

2787 GraphBLAS operations also make assumptions about the numbers of dimensions and the sizes of
2788 vectors and matrices in an operation. An operation will test these sizes and report an error if they
2789 are not *shape compatible*. For example, when multiplying two matrices, $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, the number
2790 of rows of \mathbf{C} must equal the number of rows of \mathbf{A} , the number of columns of \mathbf{A} must match the
2791 number of rows of \mathbf{B} , and the number of columns of \mathbf{C} must match the number of columns of \mathbf{B} .
2792 This is the behavior expected given the mathematical definition of the operations.

2793 For most of the GraphBLAS operations involving matrices, an optional descriptor can modify the
2794 matrix associated with an input GraphBLAS matrix object. For example, if an input matrix is an

Table 4.1: A mathematical notation for the fundamental GraphBLAS operations supported in this specification. Input matrices \mathbf{A} and \mathbf{B} may be optionally transposed (not shown). Use of an optional accumulate with existing values in the output object is indicated with \odot . Use of optional write masks and replace flags are indicated as $\mathbf{C}\langle\mathbf{M}, r\rangle$ when applied to the output matrix, \mathbf{C} . The mask controls which values resulting from the operation on the right-hand side are written into the output object (complement and structure flags are not shown). The "replace" option, indicated by specifying the r flag, means that all values in the output object are removed prior to assignment. If "replace" is not specified, only the values/locations computed on the right-hand side and allowed by the mask will be written to the output ("merge" mode).

Operation Name	Mathematical Notation	
mxm	$\mathbf{C}\langle\mathbf{M}, r\rangle$	$= \mathbf{C} \odot \mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{w}\langle\mathbf{m}, r\rangle$	$= \mathbf{w} \odot \mathbf{A} \oplus . \otimes \mathbf{u}$
vxm	$\mathbf{w}^T\langle\mathbf{m}^T, r\rangle$	$= \mathbf{w}^T \odot \mathbf{u}^T \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C}\langle\mathbf{M}, r\rangle$	$= \mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	$= \mathbf{w} \odot \mathbf{u} \otimes \mathbf{v}$
eWiseAdd	$\mathbf{C}\langle\mathbf{M}, r\rangle$	$= \mathbf{C} \odot \mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	$= \mathbf{w} \odot \mathbf{u} \oplus \mathbf{v}$
extract	$\mathbf{C}\langle\mathbf{M}, r\rangle$	$= \mathbf{C} \odot \mathbf{A}(i, j)$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	$= \mathbf{w} \odot \mathbf{u}(i)$
assign	$\mathbf{C}\langle\mathbf{M}, r\rangle(i, j)$	$= \mathbf{C}(i, j) \odot \mathbf{A}$
	$\mathbf{w}\langle\mathbf{m}, r\rangle(i)$	$= \mathbf{w}(i) \odot \mathbf{u}$
reduce (row)	$\mathbf{w}\langle\mathbf{m}, r\rangle$	$= \mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$
reduce (scalar)	s	$= s \odot [\oplus_{i,j} \mathbf{A}(i, j)]$
	s	$= s \odot [\oplus_i \mathbf{u}(i)]$
apply	$\mathbf{C}\langle\mathbf{M}, r\rangle$	$= \mathbf{C} \odot f_u(\mathbf{A})$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	$= \mathbf{w} \odot f_u(\mathbf{u})$
apply(indexop)	$\mathbf{C}\langle\mathbf{M}, r\rangle$	$= \mathbf{C} \odot f_i(\mathbf{A}, \text{ind}(\mathbf{A}), s)$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	$= \mathbf{w} \odot f_i(\mathbf{u}, \text{ind}(\mathbf{u}), s)$
select	$\mathbf{C}\langle\mathbf{M}, r\rangle$	$= \mathbf{C} \odot \mathbf{A}\langle f_i(\mathbf{A}, \text{ind}(\mathbf{A}), s) \rangle$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	$= \mathbf{w} \odot \mathbf{u}\langle f_i(\mathbf{u}, \text{ind}(\mathbf{u}), s) \rangle$
transpose	$\mathbf{C}\langle\mathbf{M}, r\rangle$	$= \mathbf{C} \odot \mathbf{A}^T$
kroncker	$\mathbf{C}\langle\mathbf{M}, r\rangle$	$= \mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

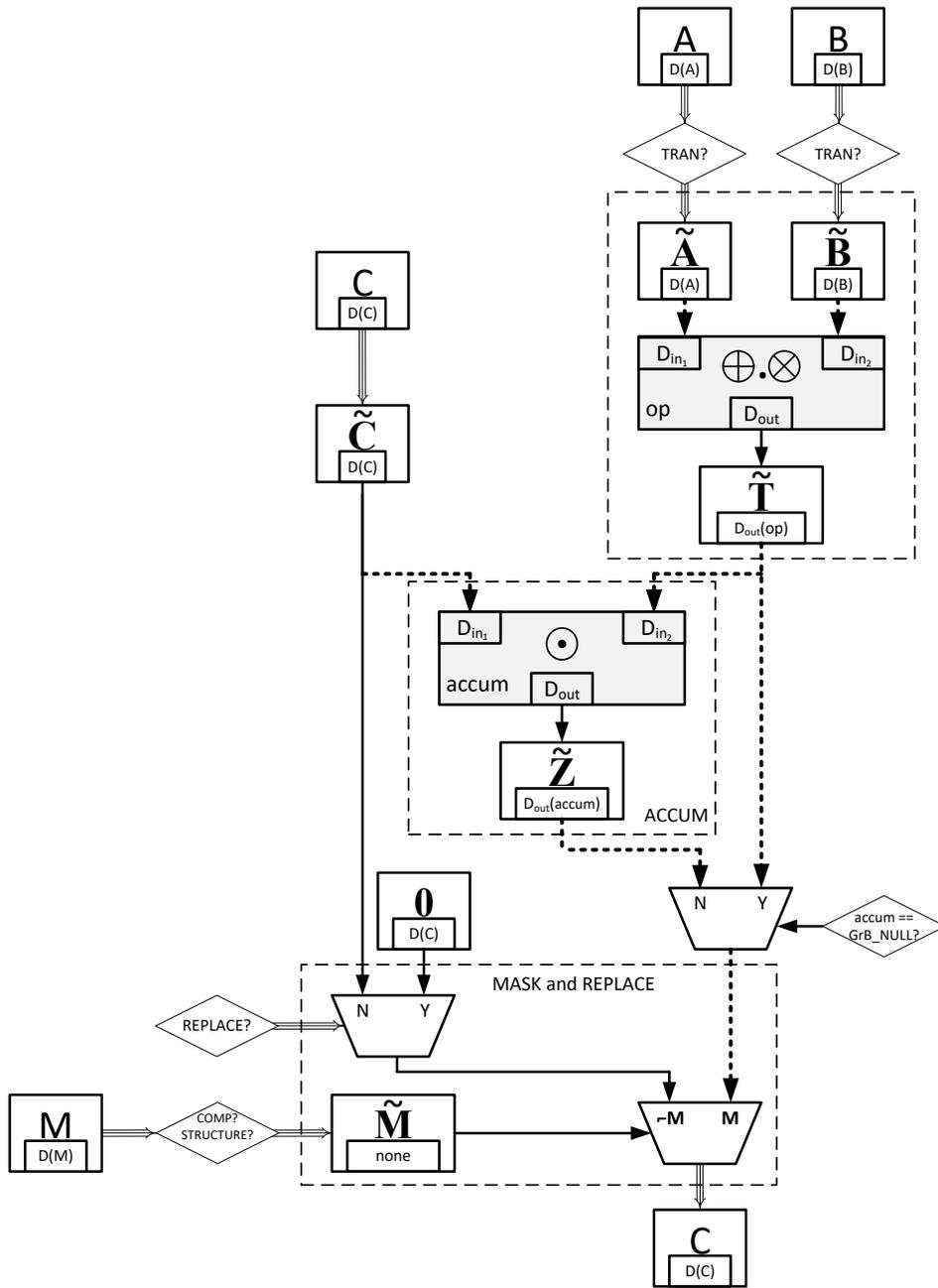


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows (\Rightarrow) denote where “as if copy” takes place (including both collections and descriptor settings). The bold, dotted arrows indicate where casting may occur between different domains.

2795 argument to a GraphBLAS operation and the associated descriptor indicates the transpose option,
 2796 then the operation occurs as if on the transposed matrix. In this case, the relationships between
 2797 the sizes in each dimension shift in the mathematically expected way.

2798 **Masks: Structure-only, Complement, and Replace**

2799 When a GraphBLAS operation supports the use of an optional mask, that mask is specified through
 2800 a GraphBLAS vector (for one-dimensional masks) or a GraphBLAS matrix (for two-dimensional
 2801 masks). When a mask is used and the `GrB_STRUCTURE` descriptor value is not set, it is applied
 2802 to the result from the operation wherever the stored values in the mask evaluate to true. If the
 2803 `GrB_STRUCTURE` descriptor is set, the mask is applied to the result from the operation wherever the
 2804 mask as a stored value (regardless of that value). Wherever the mask is applied, the result from
 2805 the operation is either assigned to the provided output matrix/vector or, if a binary accumulation
 2806 operation is provided, the result is accumulated into the corresponding elements of the provided
 2807 output matrix/vector.

2808 Given a GraphBLAS vector $\mathbf{v} = \langle D, N, \{(i, v_i)\} \rangle$, a one-dimensional mask is derived for use in the
 2809 operation as follows:

$$2810 \quad \mathbf{m} = \begin{cases} \langle N, \{\mathbf{ind}(\mathbf{v})\} \rangle, & \text{if } \text{GrB_STRUCTURE} \text{ is specified,} \\ \langle N, \{i : (\text{bool})v_i = \text{true}\} \rangle, & \text{otherwise} \end{cases}$$

2811 where $(\text{bool})v_i$ denotes casting the value v_i to a Boolean value (true or false). Likewise, given a
 2812 GraphBLAS matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$, a two-dimensional mask is derived for use in the
 2813 operation as follows:

$$2814 \quad \mathbf{M} = \begin{cases} \langle M, N, \{\mathbf{ind}(\mathbf{A})\} \rangle, & \text{if } \text{GrB_STRUCTURE} \text{ is specified,} \\ \langle M, N, \{(i, j) : (\text{bool})A_{ij} = \text{true}\} \rangle, & \text{otherwise} \end{cases}$$

2815 where $(\text{bool})A_{ij}$ denotes casting the value A_{ij} to a Boolean value. (true or false)

2816 In both the one- and two-dimensional cases, the mask may also have a subsequent complement
 2817 operation applied (*Section 3.5.4*) as specified in the descriptor, before a final mask is generated for
 2818 use in the operation.

2819 When the descriptor of an operation with a mask has specified that the `GrB_REPLACE` value is
 2820 to be applied to the output (`GrB_OUTP`), then anywhere the mask is not true, the corresponding
 2821 location in the output is cleared.

2822 **Invalid and uninitialized objects**

2823 Upon entering a GraphBLAS operation, the first step is a check that all objects are valid and ini-
 2824 tialized. (Optional parameters can be set to `GrB_NULL`, which always counts as a valid object.) An
 2825 invalid object is one that could not be computed due to a previous execution error. An uninitialized
 2826 object is one that has not yet been created by a corresponding `new` or `dup` method. Appropriate
 2827 error codes are returned if an object is not initialized (`GrB_UNINITIALIZED_OBJECT`) or invalid
 2828 (`GrB_INVALID_OBJECT`).

2829 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-
2830 ommended to initialize all GraphBLAS objects to the predefined value GrB_INVALID_HANDLE at
2831 the point of their declaration, as shown in the following examples:

```
2832         GrB_Type          type = GrB_INVALID_HANDLE;  
2833         GrB_Semiring      semiring = GrB_INVALID_HANDLE;  
2834         GrB_Matrix        matrix = GrB_INVALID_HANDLE;
```

2835 Compliance

2836 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.
2837 That is, for each operation we give a recipe for producing its outcome. Any implementation that
2838 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error
2839 model (Section 2.6) is a conforming implementation.

2840 4.3.1 mxm: Matrix-matrix multiply

2841 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

2842 C Syntax

```
2843         GrB_Info GrB_mxm(GrB_Matrix          C,  
2844                         const GrB_Matrix    Mask,  
2845                         const GrB_BinaryOp   accum,  
2846                         const GrB_Semiring   op,  
2847                         const GrB_Matrix     A,  
2848                         const GrB_Matrix     B,  
2849                         const GrB_Descriptor desc);
```

2850 Parameters

2851 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
2852 that may be accumulated with the result of the matrix product. On output, the
2853 matrix holds the results of the operation.

2854 **Mask** (IN) An optional “write” mask that controls which results from this operation are
2855 stored into the output matrix C. The mask dimensions must match those of the
2856 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
2857 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
2858 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
2859 dimensions of C), GrB_NULL should be specified.

2860 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
 2861 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 2862 specified.

2863 **op** (IN) The semiring used in the matrix-matrix multiply.

2864 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 2865 multiplication.

2866 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 2867 multiplication.

2868 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 2869 should be specified. Non-default field/value pairs are listed as follows:
 2870

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask .
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for the operation.

2871

2872 Return Values

2873 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 2874 blocking mode, this indicates that the compatibility tests on di-
 2875 mensions and domains for the input arguments passed successfully.
 2876 Either way, output matrix **C** is ready to be used in the next method
 2877 of the sequence.

2878 **GrB_PANIC** Unknown internal error.

2879 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 2880 GraphBLAS objects (input or output) is in an invalid state caused
 2881 by a previous execution error. Call **GrB_error()** to access any error
 2882 messages generated by the implementation.

2883 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

2884 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 2885 a call to **new** (or **Matrix_dup** for matrix parameters).

2886 **GrB_DIMENSION_MISMATCH** Mask and/or matrix dimensions are incompatible.

2887 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 2888 corresponding domains of the semiring or accumulation operator,
 2889 or the mask's domain is not compatible with bool (in the case where
 2890 desc[GrB_MASK].GrB_STRUCTURE is not set).

2891 **Description**

2892 GrB_mxm computes the matrix product $C = A \oplus . \otimes B$ or, if an optional binary accumulation operator
 2893 (\odot) is provided, $C = C \odot (A \oplus . \otimes B)$ (where matrices A and B can be optionally transposed).
 2894 Logically, this operation occurs in three steps:

2895 **Setup** The internal matrices and mask used in the computation are formed and their domains
 2896 and dimensions are tested for compatibility.

2897 **Compute** The indicated computations are carried out.

2898 **Output** The result is written into the output matrix, possibly under control of a mask.

2899 Up to four argument matrices are used in the GrB_mxm operation:

- 2900 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 2901 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 2902 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 2903 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

2904 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for
 2905 domain compatibility as follows:

- 2906 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
 2907 must be from one of the pre-defined types of Table 3.2.
- 2908 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 2909 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 2910 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 2911 5. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 2912 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 2913 of the accumulation operator.

2914 Two domains are compatible with each other if values from one domain can be cast to values in
 2915 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 2916 all compatible with each other. A domain from a user-defined type is only compatible with itself.

2917 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch
 2918 error listed above is returned.

2919 From the argument matrices, the internal matrices and mask used in the computation are formed
 2920 (\leftarrow denotes copy):

- 2921 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 2922 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 2923 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 2924 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 2925 (b) If `Mask \neq GrB_NULL`,
 - 2926 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 2927 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 2928 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 2929 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 2930 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 2931 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 2932 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

2933 The internal matrices and masks are checked for dimension compatibility. The following conditions
 2934 must hold:

- 2935 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 2936 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 2937 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 2938 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.
- 2939 5. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.

2940 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch
 2941 error listed above is returned.

2942 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 2943 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2944 We are now ready to carry out the matrix multiplication and any additional associated operations.
 2945 We describe this in terms of two intermediate matrices:

- 2946 • $\tilde{\mathbf{T}}$: The matrix holding the product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 2947 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

2948 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:$
 2949 $, j)) \neq \emptyset\} \rangle$ is created. The value of each of its elements is computed by

$$2950 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

2951 where \oplus and \otimes are the additive and multiplicative operators of semiring op , respectively.

2952 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 2953 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 2954 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$2955 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

2956 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 2957 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$2958 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$2959 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$2960 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

2963 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

2964 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 2965 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 2966 mask which acts as a “write mask”.

- 2967 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 2968 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$2969 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 2970 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 2971 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 2972 mask are unchanged:

$$2973 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

2974 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 2975 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 2976 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 2977 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 2978 sequence.

2979 4.3.2 vxm: Vector-matrix multiply

2980 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

2981 C Syntax

```
2982     GrB_Info GrB_vxm(GrB_Vector      w,  
2983                   const GrB_Vector  mask,  
2984                   const GrB_BinaryOp accum,  
2985                   const GrB_Semiring op,  
2986                   const GrB_Vector  u,  
2987                   const GrB_Matrix  A,  
2988                   const GrB_Descriptor desc);
```

2989 Parameters

2990 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
2991 that may be accumulated with the result of the vector-matrix product. On output,
2992 this vector holds the results of the operation.

2993 **mask** (IN) An optional “write” mask that controls which results from this operation are
2994 stored into the output vector **w**. The mask dimensions must match those of the
2995 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
2996 of the mask vector must be of type `bool` or any of the predefined “built-in” types
2997 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
2998 dimensions of **w**), `GrB_NULL` should be specified.

2999 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3000 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3001 specified.

3002 **op** (IN) Semiring used in the vector-matrix multiply.

3003 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3004 multiplication.

3005 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
3006 multiplication.

3007 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
3008 should be specified. Non-default field/value pairs are listed as follows:
3009

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation.

3010

3011 Return Values

3012 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3013 blocking mode, this indicates that the compatibility tests on di-
3014 mensions and domains for the input arguments passed successfully.
3015 Either way, output vector w is ready to be used in the next method
3016 of the sequence.

3017 GrB_PANIC Unknown internal error.

3018 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3019 GraphBLAS objects (input or output) is in an invalid state caused
3020 by a previous execution error. Call GrB_error() to access any error
3021 messages generated by the implementation.

3022 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3023 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3024 a call to new (or dup for matrix or vector parameters).

3025 GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3026 GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with
3027 the corresponding domains of the semiring or accumulation opera-
3028 tor, or the mask's domain is not compatible with bool (in the case
3029 where desc[GrB_MASK].GrB_STRUCTURE is not set).

3030 Description

3031 GrB_vxm computes the vector-matrix product $w^T = u^T \oplus . \otimes A$, or, if an optional binary accu-
3032 mulation operator (\odot) is provided, $w^T = w^T \odot (u^T \oplus . \otimes A)$ (where matrix A can be optionally
3033 transposed). Logically, this operation occurs in three steps:

3034 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
3035 domains/dimensions are tested for compatibility.

3036 **Compute** The indicated computations are carried out.

3037 **Output** The result is written into the output vector, possibly under control of a mask.

3038 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3039 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3040 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3041 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3042 4. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3043 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
3044 tested for domain compatibility as follows:

- 3045 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
3046 must be from one of the pre-defined types of Table 3.2.
- 3047 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 3048 3. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 3049 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.
- 3050 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
3051 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$
3052 of the accumulation operator.

3053 Two domains are compatible with each other if values from one domain can be cast to values in
3054 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
3055 all compatible with each other. A domain from a user-defined type is only compatible with itself.
3056 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch
3057 error listed above is returned.

3058 From the argument vectors and matrices, the internal matrices and mask used in the computation
3059 are formed (\leftarrow denotes copy):

- 3060 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3061 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3062 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3063 (b) If `mask` \neq `GrB_NULL`,
 - 3064 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3065 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 3066 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3067 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3068 4. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

3069 The internal matrices and masks are checked for shape compatibility. The following conditions
3070 must hold:

3071 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.

3072 2. $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$.

3073 3. $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$.

3074 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch
3075 error listed above is returned.

3076 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3077 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3078 We are now ready to carry out the vector-matrix multiplication and any additional associated
3079 operations. We describe this in terms of two intermediate vectors:

- 3080 • $\tilde{\mathbf{t}}$: The vector holding the product of vector $\tilde{\mathbf{u}}^T$ and matrix $\tilde{\mathbf{A}}$.
- 3081 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3082 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{ncols}(\tilde{\mathbf{A}}), \{(j, t_j) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j)) \neq \emptyset\} \rangle$ is created.
3083 The value of each of its elements is computed by

$$3084 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3085 where \oplus and \otimes are the additive and multiplicative operators of semiring `op`, respectively.

3086 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3087 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3088 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3089 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3090 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3091 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3092 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3093 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3094 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3095 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3096 \end{aligned}$$

3097 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3098 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3099 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3100 mask which acts as a “write mask”.

- 3101 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 3102 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3103 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3104 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3105 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3106 mask are unchanged:

$$3107 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3108 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 3109 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 3110 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 3111 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3112 sequence.

3113 4.3.3 mxv: Matrix-vector multiply

3114 Multiplies a matrix by a vector on a semiring. The result is a vector.

3115 C Syntax

```
3116     GrB_Info GrB_mxv(GrB_Vector      w,
3117                    const GrB_Vector mask,
3118                    const GrB_BinaryOp accum,
3119                    const GrB_Semiring op,
3120                    const GrB_Matrix  A,
3121                    const GrB_Vector  u,
3122                    const GrB_Descriptor desc);
```

3123 Parameters

3124 \mathbf{w} (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 3125 that may be accumulated with the result of the matrix-vector product. On output,
 3126 this vector holds the results of the operation.

3127 \mathbf{mask} (IN) An optional “write” mask that controls which results from this operation are
 3128 stored into the output vector \mathbf{w} . The mask dimensions must match those of the
 3129 vector \mathbf{w} . If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain

3130 of the mask vector must be of type `bool` or any of the predefined “built-in” types
 3131 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
 3132 dimensions of `w`), `GrB_NULL` should be specified.

3133 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`
 3134 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 3135 specified.

3136 `op` (IN) Semiring used in the vector-matrix multiply.

3137 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3138 multiplication.

3139 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3140 multiplication.

3141 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 3142 should be specified. Non-default field/value pairs are listed as follows:

3143

Param	Field	Value	Description
<code>w</code>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <code>w</code> is cleared (all elements removed) before the result is stored in it.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <code>mask</code> .
<code>A</code>	<code>GrB_INP0</code>	<code>GrB_TRAN</code>	Use transpose of <code>A</code> for the operation.

3144

3145 Return Values

3146 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 3147 blocking mode, this indicates that the compatibility tests on di-
 3148 mensions and domains for the input arguments passed successfully.
 3149 Either way, output vector `w` is ready to be used in the next method
 3150 of the sequence.

3151 `GrB_PANIC` Unknown internal error.

3152 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 3153 GraphBLAS objects (input or output) is in an invalid state caused
 3154 by a previous execution error. Call `GrB_error()` to access any error
 3155 messages generated by the implementation.

3156 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3157 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by
 3158 a call to `new` (or `dup` for matrix or vector parameters).

3159 GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3160 GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with
3161 the corresponding domains of the semiring or accumulation opera-
3162 tor, or the mask's domain is not compatible with `bool` (in the case
3163 where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3164 Description

3165 GrB_mvx computes the matrix-vector product $\mathbf{w} = \mathbf{A} \oplus . \otimes \mathbf{u}$, or, if an optional binary accumulation
3166 operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{A} \oplus . \otimes \mathbf{u})$ (where matrix \mathbf{A} can be optionally transposed).
3167 Logically, this operation occurs in three steps:

3168 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
3169 domains/dimensions are tested for compatibility.

3170 **Compute** The indicated computations are carried out.

3171 **Output** The result is written into the output vector, possibly under control of a mask.

3172 Up to four argument vectors or matrices are used in the GrB_mvx operation:

- 3173 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3174 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3175 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 3176 4. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

3177 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
3178 tested for domain compatibility as follows:

- 3179 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
3180 must be from one of the pre-defined types of Table 3.2.
- 3181 2. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 3182 3. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 3183 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.
- 3184 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
3185 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$
3186 of the accumulation operator.

3187 Two domains are compatible with each other if values from one domain can be cast to values in
3188 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
3189 all compatible with each other. A domain from a user-defined type is only compatible with itself.
3190 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the domain mismatch
3191 error listed above is returned.

3192 From the argument vectors and matrices, the internal matrices and mask used in the computation
3193 are formed (\leftarrow denotes copy):

- 3194 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3195 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3196 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3197 (b) If `mask \neq GrB_NULL`,
 - 3198 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3199 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 3200 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3201 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3202 4. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3203 The internal matrices and masks are checked for shape compatibility. The following conditions
3204 must hold:

- 3205 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$.
- 3206 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 3207 3. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

3208 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the dimension mismatch
3209 error listed above is returned.

3210 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3211 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3212 We are now ready to carry out the matrix-vector multiplication and any additional associated
3213 operations. We describe this in terms of two intermediate vectors:

- 3214 • $\tilde{\mathbf{t}}$: The vector holding the product of matrix $\tilde{\mathbf{A}}$ and vector $\tilde{\mathbf{u}}$.
- 3215 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3216 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \{(i, t_i) : \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{u}}) \neq \emptyset\} \rangle$ is created.
3217 The value of each of its elements is computed by

$$3218 \quad t_i = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3219 where \oplus and \otimes are the additive and multiplicative operators of semiring op , respectively.

3220 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3221 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3222 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3223 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3224 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3225 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3226 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$3227 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$3228 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3231 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3232 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3233 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3234 mask which acts as a “write mask”.

- 3235 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 3236 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3237 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3238 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3239 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3240 mask are unchanged:

$$3241 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3242 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3243 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3244 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
 3245 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3246 sequence.

3247 4.3.4 eWiseMult: Element-wise multiplication

3248 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation
 3249 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of
 3250 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”
 3251 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
 3252 the set of values from the resulting index set.

3253 4.3.4.1 eWiseMult: Vector variant

3254 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-
3255 ducing a third vector as result.

3256 C Syntax

```
3257     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3258                          const GrB_Vector  mask,  
3259                          const GrB_BinaryOp accum,  
3260                          const GrB_Semiring op,  
3261                          const GrB_Vector  u,  
3262                          const GrB_Vector  v,  
3263                          const GrB_Descriptor desc);  
3264  
3265     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3266                          const GrB_Vector  mask,  
3267                          const GrB_BinaryOp accum,  
3268                          const GrB_Monoid  op,  
3269                          const GrB_Vector  u,  
3270                          const GrB_Vector  v,  
3271                          const GrB_Descriptor desc);  
3272  
3273     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3274                          const GrB_Vector  mask,  
3275                          const GrB_BinaryOp accum,  
3276                          const GrB_BinaryOp op,  
3277                          const GrB_Vector  u,  
3278                          const GrB_Vector  v,  
3279                          const GrB_Descriptor desc);
```

3280 Parameters

3281 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3282 that may be accumulated with the result of the element-wise operation. On output,
3283 this vector holds the results of the operation.

3284 **mask** (IN) An optional “write” mask that controls which results from this operation are
3285 stored into the output vector **w**. The mask dimensions must match those of the
3286 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3287 of the mask vector must be of type `bool` or any of the predefined “built-in” types
3288 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
3289 dimensions of **w**), `GrB_NULL` should be specified.

3290 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3291 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 3292 specified.

3293 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
 3294 operation. Depending on which type is passed, the following defines the binary
 3295 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3296 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3297 Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ig-
 3298 nored.

3299 Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$; the additive monoid
 3300 is ignored.

3301 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 3302 operation.

3303 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3304 operation.

3305 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3306 should be specified. Non-default field/value pairs are listed as follows:

3307

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

3308

3309 Return Values

3310 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3311 blocking mode, this indicates that the compatibility tests on di-
 3312 mensions and domains for the input arguments passed successfully.
 3313 Either way, output vector w is ready to be used in the next method
 3314 of the sequence.

3315 GrB_PANIC Unknown internal error.

3316 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3317 GraphBLAS objects (input or output) is in an invalid state caused
 3318 by a previous execution error. Call GrB_error() to access any error
 3319 messages generated by the implementation.

3320 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3321 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3322 a call to new (or dup for vector parameters).

3323 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3324 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 3325 responding domains of the binary operator (op) or accumulation
 3326 operator, or the mask’s domain is not compatible with bool (in the
 3327 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

3328 **Description**

3329 This variant of GrB_eWiseMult computes the element-wise “product” of two GraphBLAS vectors:
 3330 $w = u \otimes v$, or, if an optional binary accumulation operator (\odot) is provided, $w = w \odot (u \otimes v)$.
 3331 Logically, this operation occurs in three steps:

3332 **Setup** The internal vectors and mask used in the computation are formed and their domains
 3333 and dimensions are tested for compatibility.

3334 **Compute** The indicated computations are carried out.

3335 **Output** The result is written into the output vector, possibly under control of a mask.

3336 Up to four argument vectors are used in the GrB_eWiseMult operation:

- 3337 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3338 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3339 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3340 4. $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3341 The argument vectors, the “product” operator (op), and the accumulation operator (if provided)
 3342 are tested for domain compatibility as follows:

- 3343 1. If mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
 3344 must be from one of the pre-defined types of Table 3.2.
- 3345 2. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3346 3. $\mathbf{D}(v)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3347 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3348 5. If accum is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3349 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3350 the accumulation operator.

3351 Two domains are compatible with each other if values from one domain can be cast to values in
 3352 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3353 compatible with each other. A domain from a user-defined type is only compatible with itself. If any
 3354 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3355 error listed above is returned.

3356 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3357 denotes copy):

- 3358 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3359 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3360 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3361 (b) If `mask \neq GrB_NULL`,
 - 3362 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3363 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 3364 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3365 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3366 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3367 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3368 must hold:

- 3369 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}}) = \mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{v}})$.

3370 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3371 mismatch error listed above is returned.

3372 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3373 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3374 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3375 We describe this in terms of two intermediate vectors:

- 3376 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “product” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3377 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3378 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$ is created. The
 3379 value of each of its elements is computed by:

$$3380 \quad t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}))$$

3381 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

3382 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3383 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3384 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3385 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3386 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3387 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

3388

$$3389 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3390

$$3391 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3392 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3393 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3394 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3395 mask which acts as a “write mask”.

3396 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 3397 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3398 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3399 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3400 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3401 mask are unchanged:

$$3402 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3403 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3404 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3405 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
 3406 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3407 sequence.

3408 4.3.4.2 eWiseMult: Matrix variant

3409 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-
 3410 ducing a third matrix as result.

3411 C Syntax

```
3412     GrB_Info GrB_eWiseMult(GrB_Matrix      C,  
3413                          const GrB_Matrix Mask,  
3414                          const GrB_BinaryOp accum,  
3415                          const GrB_Semiring op,  
3416                          const GrB_Matrix  A,  
3417                          const GrB_Matrix  B,  
3418                          const GrB_Descriptor desc);  
3419  
3420     GrB_Info GrB_eWiseMult(GrB_Matrix      C,  
3421                          const GrB_Matrix Mask,  
3422                          const GrB_BinaryOp accum,  
3423                          const GrB_Monoid  op,  
3424                          const GrB_Matrix  A,  
3425                          const GrB_Matrix  B,  
3426                          const GrB_Descriptor desc);  
3427  
3428     GrB_Info GrB_eWiseMult(GrB_Matrix      C,  
3429                          const GrB_Matrix Mask,  
3430                          const GrB_BinaryOp accum,  
3431                          const GrB_BinaryOp op,  
3432                          const GrB_Matrix  A,  
3433                          const GrB_Matrix  B,  
3434                          const GrB_Descriptor desc);
```

3435 Parameters

3436 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3437 that may be accumulated with the result of the element-wise operation. On output,
3438 the matrix holds the results of the operation.

3439 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3440 stored into the output matrix C. The mask dimensions must match those of the
3441 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
3442 of the Mask matrix must be of type bool or any of the predefined “built-in” types
3443 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
3444 dimensions of C), GrB_NULL should be specified.

3445 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3446 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3447 specified.

3448 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3449 operation. Depending on which type is passed, the following defines the binary
3450 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3451

BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3452

Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ignored.

3453

3454

Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$; the additive monoid is ignored.

3455

3456

A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the operation.

3457

3458

B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the operation.

3459

3460

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

3461

3462

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for the operation.

3463

3464 Return Values

3465

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

3466

3467

3468

3469

3470

GrB_PANIC Unknown internal error.

3471

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

3472

3473

3474

3475

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3476

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

3477

3478

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3479 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3480 corresponding domains of the binary operator (`op`) or accumulation
 3481 operator, or the mask's domain is not compatible with `bool` (in the
 3482 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3483 Description

3484 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:
 3485 $C = A \otimes B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \otimes B)$.
 3486 Logically, this operation occurs in three steps:

3487 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3488 and dimensions are tested for compatibility.

3489 **Compute** The indicated computations are carried out.

3490 **Output** The result is written into the output matrix, possibly under control of a mask.

3491 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3492 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3493 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 3494 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3495 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3496 The argument matrices, the “product” operator (`op`), and the accumulation operator (if provided)
 3497 are tested for domain compatibility as follows:

- 3498 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3499 must be from one of the pre-defined types of Table 3.2.
- 3500 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3501 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3502 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3503 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3504 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3505 the accumulation operator.

3506 Two domains are compatible with each other if values from one domain can be cast to values in
 3507 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3508 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3509 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3510 error listed above is returned.

3511 From the argument matrices, the internal matrices and mask used in the computation are formed
 3512 (\leftarrow denotes copy):

- 3513 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3514 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3515 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3516 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3517 (b) If `Mask \neq GrB_NULL`,
 - 3518 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3519 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3520 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3521 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 3522 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3523 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3524 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3525 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3526 must hold:

- 3527 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3528 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3529 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3530 mismatch error listed above is returned.

3531 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3532 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3533 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3534 We describe this in terms of two intermediate matrices:

- 3535 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise product of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3536 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3537 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$
 3538 is created. The value of each of its elements is computed by

$$3539 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3540 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3541 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3542 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

3543
$$\tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3544 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3545 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

3546
$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

3547
$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3548
$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3549

3550

3551 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3552 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 3553 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 3554 mask which acts as a “write mask”.

- 3555 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 3556 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

3557
$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3558 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 3559 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 3560 mask are unchanged:

3561
$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3562 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3563 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3564 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 3565 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3566 sequence.

3567 4.3.5 eWiseAdd: Element-wise addition

3568 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation
 3569 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of
 3570 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”
 3571 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
 3572 the set of values from the resulting index set.

3573 **4.3.5.1 eWiseAdd: Vector variant**

3574 Perform element-wise (general) addition on the elements of two vectors, producing a third vector
3575 as result.

3576 **C Syntax**

```
3577     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3578                          const GrB_Vector  mask,  
3579                          const GrB_BinaryOp accum,  
3580                          const GrB_Semiring op,  
3581                          const GrB_Vector  u,  
3582                          const GrB_Vector  v,  
3583                          const GrB_Descriptor desc);  
3584  
3585     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3586                          const GrB_Vector  mask,  
3587                          const GrB_BinaryOp accum,  
3588                          const GrB_Monoid  op,  
3589                          const GrB_Vector  u,  
3590                          const GrB_Vector  v,  
3591                          const GrB_Descriptor desc);  
3592  
3593     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3594                          const GrB_Vector  mask,  
3595                          const GrB_BinaryOp accum,  
3596                          const GrB_BinaryOp op,  
3597                          const GrB_Vector  u,  
3598                          const GrB_Vector  v,  
3599                          const GrB_Descriptor desc);
```

3600 **Parameters**

3601 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3602 that may be accumulated with the result of the element-wise operation. On output,
3603 this vector holds the results of the operation.

3604 **mask** (IN) An optional “write” mask that controls which results from this operation are
3605 stored into the output vector **w**. The mask dimensions must match those of the
3606 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3607 of the mask vector must be of type **bool** or any of the predefined “built-in” types
3608 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
3609 dimensions of **w**), **GrB_NULL** should be specified.

3610 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3611 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 3612 specified.

3613 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
 3614 operation. Depending on which type is passed, the following defines the binary
 3615 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3616 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3617 Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ig-
 3618 nored.

3619 Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus(\text{op}) \rangle$; the multiplicative bi-
 3620 nary op and additive identity are ignored.

3621 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 3622 operation.

3623 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3624 operation.

3625 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3626 should be specified. Non-default field/value pairs are listed as follows:

3627

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

3629 **Return Values**

3630 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3631 blocking mode, this indicates that the compatibility tests on di-
 3632 mensions and domains for the input arguments passed successfully.
 3633 Either way, output vector w is ready to be used in the next method
 3634 of the sequence.

3635 GrB_PANIC Unknown internal error.

3636 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3637 GraphBLAS objects (input or output) is in an invalid state caused
 3638 by a previous execution error. Call GrB_error() to access any error
 3639 messages generated by the implementation.

3640 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3641 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3642 a call to new (or dup for vector parameters).

3643 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3644 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
3645 responding domains of the binary operator (op) or accumulation
3646 operator, or the mask's domain is not compatible with bool (in the
3647 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

3648 Description

3649 This variant of GrB_eWiseAdd computes the element-wise “sum” of two GraphBLAS vectors: $w =$
3650 $u \oplus v$, or, if an optional binary accumulation operator (\odot) is provided, $w = w \odot (u \oplus v)$. Logically,
3651 this operation occurs in three steps:

3652 **Setup** The internal vectors and mask used in the computation are formed and their domains
3653 and dimensions are tested for compatibility.

3654 **Compute** The indicated computations are carried out.

3655 **Output** The result is written into the output vector, possibly under control of a mask.

3656 Up to four argument vectors are used in the GrB_eWiseAdd operation:

- 3657 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3658 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3659 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3660 4. $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3661 The argument vectors, the “sum” operator (op), and the accumulation operator (if provided) are
3662 tested for domain compatibility as follows:

- 3663 1. If mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
3664 must be from one of the pre-defined types of Table 3.2.
- 3665 2. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3666 3. $\mathbf{D}(v)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3667 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3668 5. $\mathbf{D}(u)$ and $\mathbf{D}(v)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3669 6. If accum is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3670 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
3671 the accumulation operator.

3672 Two domains are compatible with each other if values from one domain can be cast to values in
 3673 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3674 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3675 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch
 3676 error listed above is returned.

3677 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3678 denotes copy):

- 3679 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3680 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3681 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3682 (b) If `mask \neq GrB_NULL`,
 - 3683 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3684 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 3685 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3686 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3687 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3688 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3689 must hold:

- 3690 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}}) = \mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{v}})$.

3691 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 3692 mismatch error listed above is returned.

3693 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3694 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3695 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3696 We describe this in terms of two intermediate vectors:

- 3697 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “sum” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3698 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3699 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \mathbf{ind}(\tilde{\mathbf{u}}) \cup \mathbf{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$ is created. The
 3700 value of each of its elements is computed by:

$$3701 \quad t_i = (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}))$$

$$3702 \quad t_i = \tilde{\mathbf{u}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3704
3705

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3706 where the difference operator in the previous expressions refers to set difference.

3707 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3708 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3709 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3710 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3711 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3712 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3713 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$3714 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$3715 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3716
3717
3718 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3719 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
3720 using what is called a *standard vector mask and replace*. This is carried out under control of the
3721 mask which acts as a “write mask”.

- 3722 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
3723 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3724 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3725 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
3726 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
3727 mask are unchanged:

$$3728 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3729 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
3730 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
3731 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
3732 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3733 sequence.

3734 4.3.5.2 eWiseAdd: Matrix variant

3735 Perform element-wise (general) addition on the elements of two matrices, producing a third matrix
3736 as result.

3737 C Syntax

```
3738     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,  
3739                          const GrB_Matrix Mask,  
3740                          const GrB_BinaryOp accum,  
3741                          const GrB_Semiring op,  
3742                          const GrB_Matrix A,  
3743                          const GrB_Matrix B,  
3744                          const GrB_Descriptor desc);  
3745  
3746     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,  
3747                          const GrB_Matrix Mask,  
3748                          const GrB_BinaryOp accum,  
3749                          const GrB_Monoid op,  
3750                          const GrB_Matrix A,  
3751                          const GrB_Matrix B,  
3752                          const GrB_Descriptor desc);  
3753  
3754     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,  
3755                          const GrB_Matrix Mask,  
3756                          const GrB_BinaryOp accum,  
3757                          const GrB_BinaryOp op,  
3758                          const GrB_Matrix A,  
3759                          const GrB_Matrix B,  
3760                          const GrB_Descriptor desc);
```

3761 Parameters

3762 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3763 that may be accumulated with the result of the element-wise operation. On output,
3764 the matrix holds the results of the operation.

3765 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3766 stored into the output matrix C. The mask dimensions must match those of the
3767 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
3768 of the Mask matrix must be of type bool or any of the predefined “built-in” types
3769 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
3770 dimensions of C), GrB_NULL should be specified.

3771 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3772 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3773 specified.

3774 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
3775 operation. Depending on which type is passed, the following defines the binary
3776 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788

BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ignored.

Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus(\text{op}) \rangle$; the multiplicative binary op and additive identity are ignored.

A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the operation.

B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the operation.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for the operation.

3789

3790 Return Values

3791
3792
3793
3794
3795
3796
3797
3798
3799
3800
3801
3802
3803
3804

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3805 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3806 corresponding domains of the binary operator (\oplus) or accumulation
 3807 operator, or the mask's domain is not compatible with `bool` (in the
 3808 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3809 Description

3810 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:
 3811 $C = A \oplus B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \oplus B)$.
 3812 Logically, this operation occurs in three steps:

3813 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3814 and dimensions are tested for compatibility.

3815 **Compute** The indicated computations are carried out.

3816 **Output** The result is written into the output matrix, possibly under control of a mask.

3817 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3818 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3819 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 3820 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3821 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3822 The argument matrices, the “sum” operator (\oplus), and the accumulation operator (if provided) are
 3823 tested for domain compatibility as follows:

- 3824 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3825 must be from one of the pre-defined types of Table 3.2.
- 3826 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\oplus)$.
- 3827 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\oplus)$.
- 3828 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\oplus)$.
- 3829 5. $\mathbf{D}(A)$ and $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{out}(\oplus)$.
- 3830 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3831 of the accumulation operator and $\mathbf{D}_{out}(\oplus)$ of \oplus must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3832 the accumulation operator.

3833 Two domains are compatible with each other if values from one domain can be cast to values in
 3834 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3835 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3836 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch
 3837 error listed above is returned.

3838 From the argument matrices, the internal matrices and mask used in the computation are formed
 3839 (\leftarrow denotes copy):

- 3840 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3841 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3842 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3843 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3844 (b) If `Mask \neq GrB_NULL`,
 - 3845 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3846 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3847 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3848 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 3849 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3850 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3851 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3852 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3853 must hold:

- 3854 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3855 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3856 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 3857 mismatch error listed above is returned.

3858 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3859 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3860 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3861 We describe this in terms of two intermediate matrices:

- 3862 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise sum of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3863 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3864 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$
 3865 is created. The value of each of its elements is computed by

$$3866 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

$$3867 \quad T_{ij} = \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})))$$

$$3869 \quad T_{ij} = \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})))$$

3871 where the difference operator in the previous expressions refers to set difference.

3872 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3873 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3874 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3875 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3876 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3877 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3878 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3879 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3881 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3883 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3884 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 3885 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 3886 mask which acts as a “write mask”.

- 3887 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 3888 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3889 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3890 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 3891 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 3892 mask are unchanged:

$$3893 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3894 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3895 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3896 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 3897 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3898 sequence.

3899 **4.3.6 extract: Selecting sub-graphs**

3900 Extract a subset of a matrix or vector.

3901 **4.3.6.1 extract: Standard vector variant**

3902 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector
3903 whose size is equal to the number of indices.

3904 **C Syntax**

```
3905     GrB_Info GrB_extract(GrB_Vector      w,  
3906                        const GrB_Vector mask,  
3907                        const GrB_BinaryOp accum,  
3908                        const GrB_Vector  u,  
3909                        const GrB_Index   *indices,  
3910                        GrB_Index       nindices,  
3911                        const GrB_Descriptor desc);
```

3912 **Parameters**

3913 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3914 that may be accumulated with the result of the extract operation. On output, this
3915 vector holds the results of the operation.

3916 **mask** (IN) An optional “write” mask that controls which results from this operation are
3917 stored into the output vector **w**. The mask dimensions must match those of the
3918 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3919 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3920 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
3921 dimensions of **w**), **GrB_NULL** should be specified.

3922 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3923 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
3924 specified.

3925 **u** (IN) The GraphBLAS vector from which the subset is extracted.

3926 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of
3927 elements from **u** that are extracted. If all elements of **u** are to be extracted in order
3928 from 0 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution
3929 mode and return value, this array may be manipulated by the caller after this
3930 operation returns without affecting any deferred computations for this operation.

3931 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

3932 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3933 should be specified. Non-default field/value pairs are listed as follows:

3934	Param	Field	Value	Description
	w	GrB_OUTP	GrB_REPLACE	Output vector <i>w</i> is cleared (all elements removed) before the result is stored in it.
3935	mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <i>mask</i> vector. The stored values are not examined.
	mask	GrB_MASK	GrB_COMP	Use the complement of <i>mask</i> .

3936 Return Values

3937 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3938 blocking mode, this indicates that the compatibility tests on
 3939 dimensions and domains for the input arguments passed suc-
 3940 cessfully. Either way, output vector *w* is ready to be used in the
 3941 next method of the sequence.

3942 GrB_PANIC Unknown internal error.

3943 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 3944 opaque GraphBLAS objects (input or output) is in an invalid
 3945 state caused by a previous execution error. Call GrB_error() to
 3946 access any error messages generated by the implementation.

3947 GrB_OUT_OF_MEMORY Not enough memory available for operation.

3948 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
 3949 by a call to *new* (or *dup* for vector parameters).

3950 GrB_INDEX_OUT_OF_BOUNDS A value in *indices* is greater than or equal to **size(u)**. In non-
 3951 blocking mode, this error can be deferred.

3952 GrB_DIMENSION_MISMATCH *mask* and *w* dimensions are incompatible, or *nindices* \neq **size(w)**.

3953 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each
 3954 other or the corresponding domains of the accumulation oper-
 3955 ator, or the mask's domain is not compatible with *bool* (in the
 3956 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

3957 GrB_NULL_POINTER Argument *row_indices* is a NULL pointer.

3958 Description

3959 This variant of GrB_extract computes the result of extracting a subset of locations from a Graph-
 3960 BLAS vector in a specific order: $w = u(\text{indices})$; or, if an optional binary accumulation operator

3961 (\odot) is provided, $w = w \odot u(\text{indices})$. More explicitly:

$$\begin{aligned} 3962 \quad w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

3963 Logically, this operation occurs in three steps:

3964 **Setup** The internal vectors and mask used in the computation are formed and their domains
3965 and dimensions are tested for compatibility.

3966 **Compute** The indicated computations are carried out.

3967 **Output** The result is written into the output vector, possibly under control of a mask.

3968 Up to three argument vectors are used in this `GrB_extract` operation:

- 3969 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3970 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3971 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3972 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
3973 bility as follows:

- 3974 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3975 must be from one of the pre-defined types of Table 3.2.
- 3976 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(u)$.
- 3977 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3978 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
3979 mulation operator.

3980 Two domains are compatible with each other if values from one domain can be cast to values in
3981 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
3982 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3983 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
3984 error listed above is returned.

3985 From the arguments, the internal vectors, mask, and index array used in the computation are
3986 formed (\leftarrow denotes copy):

- 3987 1. Vector $\tilde{w} \leftarrow w$.
- 3988 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
 - 3989 (a) If `mask = GrB_NULL`, then $\tilde{m} = \langle \mathbf{size}(w), \{i, \forall i : 0 \leq i < \mathbf{size}(w)\} \rangle$.

- 3990 (b) If $\text{mask} \neq \text{GrB_NULL}$,
- 3991 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,
- 3992 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 3993 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
- 3994 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3995 4. The internal index array, $\widetilde{\mathbf{I}}$, is computed from argument indices as follows:
- 3996 (a) If $\text{indices} = \text{GrB_ALL}$, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.
- 3997 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

3998 The internal vectors and mask are checked for dimension compatibility. The following conditions
3999 must hold:

- 4000 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4001 2. $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$.

4002 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
4003 match error listed above is returned.

4004 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4005 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4006 We are now ready to carry out the extract and any additional associated operations. We describe
4007 this in terms of two intermediate vectors:

- 4008 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{w}}$.
- 4009 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4010 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4011 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \mathbf{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4012 At this point, if any value in $\widetilde{\mathbf{I}}$ is not in the valid range of indices for vector $\widetilde{\mathbf{u}}$, the execution of
4013 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`
4014 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4015 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4016 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 4017 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 4018 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$4019 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

4020 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4021 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned}
 4022 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\
 4023 \\
 4024 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 4025 \\
 4026 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),
 \end{aligned}$$

4027 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4028 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 4029 using what is called a *standard vector mask and replace*. This is carried out under control of the
 4030 mask which acts as a “write mask”.

- 4031 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 4032 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4033 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 4034 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 4035 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 4036 mask are unchanged:

$$4037 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4038 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 4039 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 4040 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 4041 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4042 sequence.

4043 4.3.6.2 extract: Standard matrix variant

4044 Extract a sub-matrix from a larger matrix as specified by a set of row indices and a set of column
 4045 indices. The result is a matrix whose size is equal to size of the sets of indices.

4046 C Syntax

```

4047     GrB_Info GrB_extract(GrB_Matrix      C,
4048                        const GrB_Matrix  Mask,
4049                        const GrB_BinaryOp accum,
4050                        const GrB_Matrix  A,
4051                        const GrB_Index   *row_indices,
4052                        GrB_Index         nrows,
4053                        const GrB_Index   *col_indices,
4054                        GrB_Index         ncols,
4055                        const GrB_Descriptor desc);

```

4056 **Parameters**

4057 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 4058 that may be accumulated with the result of the extract operation. On output, the
 4059 matrix holds the results of the operation.

4060 Mask (IN) An optional “write” mask that controls which results from this operation are
 4061 stored into the output matrix C. The mask dimensions must match those of the
 4062 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 4063 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
 4064 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
 4065 dimensions of C), GrB_NULL should be specified.

4066 accum (IN) An optional binary operator used for accumulating entries into existing C
 4067 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 4068 specified.

4069 A (IN) The GraphBLAS matrix from which the subset is extracted.

4070 row_indices (IN) Pointer to the ordered set (array) of indices corresponding to the rows of A
 4071 from which elements are extracted. If elements in all rows of A are to be extracted
 4072 in order, GrB_ALL should be specified. Regardless of execution mode and return
 4073 value, this array may be manipulated by the caller after this operation returns
 4074 without affecting any deferred computations for this operation.

4075 nrows (IN) The number of values in the row_indices array. Must be equal to `nrows(C)`.

4076 col_indices (IN) Pointer to the ordered set (array) of indices corresponding to the columns
 4077 of A from which elements are extracted. If elements in all columns of A are to
 4078 be extracted in order, then GrB_ALL should be specified. Regardless of execution
 4079 mode and return value, this array may be manipulated by the caller after this
 4080 operation returns without affecting any deferred computations for this operation.

4081 ncols (IN) The number of values in the col_indices array. Must be equal to `ncols(C)`.

4082 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4083 should be specified. Non-default field/value pairs are listed as follows:
 4084

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

4085

4086 Return Values

- 4087 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
4088 blocking mode, this indicates that the compatibility tests on
4089 dimensions and domains for the input arguments passed suc-
4090 cessfully. Either way, output matrix C is ready to be used in the
4091 next method of the sequence.
- 4092 GrB_PANIC Unknown internal error.
- 4093 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
4094 opaque GraphBLAS objects (input or output) is in an invalid
4095 state caused by a previous execution error. Call GrB_error() to
4096 access any error messages generated by the implementation.
- 4097 GrB_OUT_OF_MEMORY Not enough memory available for the operation.
- 4098 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
4099 by a call to new (or Matrix_dup for matrix parameters).
- 4100 GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to nrows(A), or
4101 a value in col_indices is greater than or equal to ncols(A). In
4102 non-blocking mode, this error can be deferred.
- 4103 GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrows \neq nrows(C), or
4104 ncols \neq ncols(C).
- 4105 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with each
4106 other or the corresponding domains of the accumulation oper-
4107 ator, or the mask's domain is not compatible with bool (in the
4108 case where desc[GrB_MASK].GrB_STRUCTURE is not set).
- 4109 GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices
4110 is a NULL pointer, or both.

4111 Description

4112 This variant of GrB_extract computes the result of extracting a subset of locations from specified
4113 rows and columns of a GraphBLAS matrix in a specific order: $C = A(\text{row_indices}, \text{col_indices})$; or,
4114 if an optional binary accumulation operator (\odot) is provided, $C = C \odot A(\text{row_indices}, \text{col_indices})$.
4115 More explicitly (not accounting for an optional transpose of A):

$$4116 \quad C(i, j) = \quad A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or}$$
$$\quad C(i, j) = C(i, j) \odot A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}$$

4117 Logically, this operation occurs in three steps:

4118 **Setup** The internal matrices and mask used in the computation are formed and their domains
4119 and dimensions are tested for compatibility.

4120 **Compute** The indicated computations are carried out.

4121 **Output** The result is written into the output matrix, possibly under control of a mask.

4122 Up to three argument matrices are used in the `GrB_extract` operation:

4123 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

4124 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)

4125 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4126 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
4127 ibility as follows:

4128 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
4129 must be from one of the pre-defined types of Table 3.2.

4130 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.

4131 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4132 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4133 mulation operator.

4134 Two domains are compatible with each other if values from one domain can be cast to values in
4135 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4136 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4137 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
4138 error listed above is returned.

4139 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are
4140 formed (\leftarrow denotes copy):

4141 1. Matrix $\tilde{C} \leftarrow C$.

4142 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:

4143 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
4144 $j < \mathbf{ncols}(C)\} \rangle$.

4145 (b) If `Mask` \neq `GrB_NULL`,

4146 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
4147 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,

4148 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
4149 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.

4150 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.

4151 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

- 4152 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4153 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 4154 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \text{nrows}$.
- 4155 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4156 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$.
- 4157 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \text{ncols}$.

4158 The internal matrices and mask are checked for dimension compatibility. The following conditions
4159 must hold:

- 4160 1. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$.
- 4161 2. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$.
- 4162 3. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$.
- 4163 4. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$.

4164 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
4165 match error listed above is returned.

4166 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4167 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4168 We are now ready to carry out the extract and any additional associated operations. We describe
4169 this in terms of two intermediate matrices:

- 4170 • $\tilde{\mathbf{T}}$: The matrix holding the extraction from $\tilde{\mathbf{A}}$.
- 4171 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4172 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4173 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \\ \{(i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4174 At this point, if any value in the $\tilde{\mathbf{I}}$ array is not in the range $[0, \text{nrows}(\tilde{\mathbf{A}}))$ or any value in the $\tilde{\mathbf{J}}$
4175 array is not in the range $[0, \text{ncols}(\tilde{\mathbf{A}}))$, the execution of `GrB_extract` ends and the index out-of-
4176 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
4177 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
4178 this point forward in the sequence.

4179 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4180 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

4181 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4182 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4183 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4184 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4185 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4186 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4188 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4190 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4191 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
4192 using what is called a *standard matrix mask and replace*. This is carried out under control of the
4193 mask which acts as a “write mask”.

4194 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
4195 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4196 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4197 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
4198 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
4199 mask are unchanged:

$$4200 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4201 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4202 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4203 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
4204 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4205 sequence.

4206 4.3.6.3 extract: Column (and row) variant

4207 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the
4208 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as
4209 well.

4210 C Syntax

```
4211     GrB_Info GrB_extract(GrB_Vector      w,  
4212                        const GrB_Vector  mask,  
4213                        const GrB_BinaryOp accum,  
4214                        const GrB_Matrix   A,  
4215                        const GrB_Index   *row_indices,  
4216                        GrB_Index        nrows,  
4217                        GrB_Index        col_index,  
4218                        const GrB_Descriptor desc);
```

4219 Parameters

4220 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4221 that may be accumulated with the result of the extract operation. On output, this
4222 vector holds the results of the operation.

4223 **mask** (IN) An optional “write” mask that controls which results from this operation are
4224 stored into the output vector **w**. The mask dimensions must match those of the
4225 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
4226 of the **mask** vector must be of type `bool` or any of the predefined “built-in” types
4227 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
4228 dimensions of **w**), `GrB_NULL` should be specified.

4229 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4230 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
4231 specified.

4232 **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4233 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations
4234 within the specified column of **A** from which elements are extracted. If elements in
4235 all rows of **A** are to be extracted in order, `GrB_ALL` should be specified. Regardless
4236 of execution mode and return value, this array may be manipulated by the caller
4237 after this operation returns without affecting any deferred computations for this
4238 operation.

4239 **nrows** (IN) The number of indices in the **row_indices** array. Must be equal to `size(w)`.

4240 **col_index** (IN) The index of the column of **A** from which to extract values. It must be in the
4241 range $[0, \mathbf{ncols}(A))$.

4242 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
4243 should be specified. Non-default field/value pairs are listed as follows:
4244

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

4245

4246 Return Values

4247

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

4248

4249

4250

4251

4252

GrB_PANIC Unknown internal error.

4253

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call `GrB_error()` to access any error messages generated by the implementation.

4254

4255

4256

4257

GrB_OUT_OF_MEMORY Not enough memory available for operation.

4258

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to `new` (or `dup` for vector or matrix parameters).

4259

4260

GrB_INVALID_INDEX `col_index` is outside the allowable range (i.e., greater than `ncols(A)`).

4261

GrB_INDEX_OUT_OF_BOUNDS A value in `row_indices` is greater than or equal to `nrows(A)`. In non-blocking mode, this error can be deferred.

4262

4263

GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or `nrows` \neq `size(w)`.

4264

GrB_DOMAIN_MISMATCH The domains of the vector or matrix are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4265

4266

4267

4268

GrB_NULL_POINTER Argument `row_indices` is a NULL pointer.

4269 Description

4270

This variant of `GrB_extract` computes the result of extracting a subset of locations (in a specific order) from a specified column of a GraphBLAS matrix: `w = A(:, col_index)(row_indices)`; or, if

4271

4272 an optional binary accumulation operator (\odot) is provided, $w = w \odot A(:, \text{col_index})(\text{row_indices})$.
 4273 More explicitly:

$$4274 \quad w(i) = A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or}$$

$$w(i) = w(i) \odot A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}$$

4275 Logically, this operation occurs in three steps:

4276 **Setup** The internal matrices, vectors, and mask used in the computation are formed and their
 4277 domains and dimensions are tested for compatibility.

4278 **Compute** The indicated computations are carried out.

4279 **Output** The result is written into the output vector, possibly under control of a mask.

4280 Up to three argument vectors and matrices are used in this `GrB_extract` operation:

- 4281 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4282 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4283 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4284 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain
 4285 compatibility as follows:

- 4286 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 4287 must be from one of the pre-defined types of Table 3.2.
- 4288 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(A)$.
- 4289 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4290 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4291 mulation operator.

4292 Two domains are compatible with each other if values from one domain can be cast to values in
 4293 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4294 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4295 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
 4296 error listed above is returned.

4297 From the arguments, the internal vector, matrix, mask, and index array used in the computation
 4298 are formed (\leftarrow denotes copy):

- 4299 1. Vector $\tilde{w} \leftarrow w$.
- 4300 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
 4301 (a) If `mask = GrB_NULL`, then $\tilde{m} = \langle \mathbf{size}(w), \{i, \forall i : 0 \leq i < \mathbf{size}(w)\} \rangle$.

- 4302 (b) If $\text{mask} \neq \text{GrB_NULL}$,
- 4303 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,
- 4304 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 4305 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 4306 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 4307 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument row_indices as follows:
- 4308 (a) If $\text{indices} = \text{GrB_ALL}$, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 4309 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$.

4310 The internal vector, $\tilde{\mathbf{w}}$, and index array are checked for dimension compatibility. The following
4311 conditions must hold:

- 4312 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 4313 2. $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}$.

4314 If any compatibility rule above is violated, execution of GrB_extract ends and the dimension mis-
4315 match error listed above is returned.

4316 The col_index parameter is checked for a valid value. The following condition must hold:

- 4317 1. $0 \leq \text{col_index} < \text{ncols}(\mathbf{A})$

4318 If the rule above is violated, execution of GrB_extract ends and the invalid index error listed above
4319 is returned.

4320 From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with
4321 GrB_SUCCESS return code and defer any computation and/or execution error codes.

4322 We are now ready to carry out the extract and any additional associated operations. We describe
4323 this in terms of two intermediate vectors:

- 4324 • $\tilde{\mathbf{t}}$: The vector holding the extraction from a column of $\tilde{\mathbf{A}}$.
- 4325 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4326 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

4327
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \text{col_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\tilde{\mathbf{I}}[i], \text{col_index}) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4328 At this point, if any value in $\tilde{\mathbf{I}}$ is not in the range $[0, \text{nrows}(\tilde{\mathbf{A}}))$, the execution of GrB_extract
4329 ends and the index-out-of-bounds error listed above is generated. In GrB_NONBLOCKING mode,
4330 the error can be deferred until a sequence-terminating $\text{GrB_wait}()$ is called. Regardless, the result
4331 vector, \mathbf{w} , is invalid from this point forward in the sequence.

4332 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 4333 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 4334 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

4335
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4336 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4337 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

4338
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

 4339
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

 4340
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

 4341
 4342

4343 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4344 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 4345 using what is called a *standard vector mask and replace*. This is carried out under control of the
 4346 mask which acts as a “write mask”.

- 4347 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 4348 deleted and the content of the new output vector, \mathbf{w} , is defined as,

4349
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 4350 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 4351 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 4352 mask are unchanged:

4353
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4354 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4355 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4356 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
 4357 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4358 sequence.

4359 4.3.7 assign: Modifying sub-graphs

4360 Assign the contents of a subset of a matrix or vector.

4361 4.3.7.1 assign: Standard vector variant

4362 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.
 4363 The size of the input vector is the same size as the index array provided.

4364 **C Syntax**

```
4365         GrB_Info GrB_assign(GrB_Vector          w,  
4366                             const GrB_Vector    mask,  
4367                             const GrB_BinaryOp   accum,  
4368                             const GrB_Vector    u,  
4369                             const GrB_Index     *indices,  
4370                             GrB_Index          nindices,  
4371                             const GrB_Descriptor desc);
```

4372 **Parameters**

4373 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4374 that may be accumulated with the result of the assign operation. On output, this
4375 vector holds the results of the operation.

4376 **mask** (IN) An optional “write” mask that controls which results from this operation are
4377 stored into the output vector **w**. The mask dimensions must match those of the
4378 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4379 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4380 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
4381 dimensions of **w**), **GrB_NULL** should be specified.

4382 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4383 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4384 specified.

4385 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4386 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4387 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0
4388 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4389 and return value, this array may be manipulated by the caller after this operation
4390 returns without affecting any deferred computations for this operation. If this
4391 array contains duplicate values, it implies in assignment of more than one value to
4392 the same location which leads to undefined results.

4393 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4394 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4395 should be specified. Non-default field/value pairs are listed as follows:
4396

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

4397

4398 Return Values

4399 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
4400 blocking mode, this indicates that the compatibility tests on
4401 dimensions and domains for the input arguments passed suc-
4402 cessfully. Either way, output vector w is ready to be used in the
4403 next method of the sequence.

4404 GrB_PANIC Unknown internal error.

4405 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
4406 opaque GraphBLAS objects (input or output) is in an invalid
4407 state caused by a previous execution error. Call GrB_error() to
4408 access any error messages generated by the implementation.

4409 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4410 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
4411 by a call to new (or dup for vector parameters).

4412 GrB_INDEX_OUT_OF_BOUNDS A value in indices is greater than or equal to size(w). In non-
4413 blocking mode, this can be reported as an execution error.

4414 GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or nindices \neq size(u).

4415 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each
4416 other or the corresponding domains of the accumulation oper-
4417 ator, or the mask's domain is not compatible with bool (in the
4418 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4419 GrB_NULL_POINTER Argument indices is a NULL pointer.

4420 Description

4421 This variant of GrB_assign computes the result of assigning elements from a source GraphBLAS
4422 vector to a destination GraphBLAS vector in a specific order: $w(\text{indices}) = u$; or, if an optional
4423 binary accumulation operator (\odot) is provided, $w(\text{indices}) = w(\text{indices}) \odot u$. More explicitly:

$$4424 \quad w(\text{indices}[i]) = \quad \quad \quad u(i), \forall i : 0 \leq i < n\text{indices}, \quad \text{or}$$

$$w(\text{indices}[i]) = w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < n\text{indices}.$$

4425 Logically, this operation occurs in three steps:

4426 **Setup** The internal vectors and mask used in the computation are formed and their domains
4427 and dimensions are tested for compatibility.

4428 **Compute** The indicated computations are carried out.

4429 **Output** The result is written into the output vector, possibly under control of a mask.

4430 Up to three argument vectors are used in the GrB_assign operation:

- 4431 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4432 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4433 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4434 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4435 bility as follows:

- 4436 1. If \mathbf{mask} is not GrB_NULL, and $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is not set, then $\mathbf{D}(\mathbf{mask})$
4437 must be from one of the pre-defined types of Table 3.2.
- 4438 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 4439 3. If \mathbf{accum} is not GrB_NULL, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
4440 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
4441 mulation operator.

4442 Two domains are compatible with each other if values from one domain can be cast to values in
4443 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4444 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4445 any compatibility rule above is violated, execution of GrB_assign ends and the domain mismatch
4446 error listed above is returned.

4447 From the arguments, the internal vectors, mask and index array used in the computation are formed
4448 (\leftarrow denotes copy):

- 4449 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 4450 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument \mathbf{mask} as follows:
 - 4451 (a) If $\mathbf{mask} = \text{GrB_NULL}$, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 4452 (b) If $\mathbf{mask} \neq \text{GrB_NULL}$,
 - 4453 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 4454 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\text{bool})\mathbf{mask}(i) = \text{true}\} \rangle$.
 - 4455 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

4456

3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4457

4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:

4458

(a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.

4459

(b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

4460

The internal vector and mask are checked for dimension compatibility. The following conditions must hold:

4461

4462

1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4463

2. $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$.

4464

If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch error listed above is returned.

4465

4466

From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4467

4468

We are now ready to carry out the assign and any additional associated operations. We describe this in terms of two intermediate vectors:

4469

4470

• $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.

4471

• $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4472

The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

4473

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4474

At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{w}}$, computation ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4475

4476

4477

4478

The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

4479

• If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

4480

$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4481

The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure of $\tilde{\mathbf{w}}$ ($\text{ind}(\tilde{\mathbf{w}})$) and remove from it all the indices of $\tilde{\mathbf{w}}$ that are in the set of indices being assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4482

4483

4484

The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

4485

4486

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4487

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4488

4489

where the difference operator refers to set difference.

4490 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4491 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4492 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4493 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4494 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$4495 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$4496 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

4498 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4500 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 4501 using what is called a *standard vector mask and replace*. This is carried out under control of the
 4502 mask which acts as a “write mask”.

4503 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 4504 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4505 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4506 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 4507 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 4508 mask are unchanged:

$$4509 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4510 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4511 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4512 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
 4513 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4514 sequence.

4515 4.3.7.2 `assign`: Standard matrix variant

4516 Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices.
 4517 The dimensions of the input matrix are the same size as the row and column index arrays provided.

4518 C Syntax

```
4519         GrB_Info GrB_assign(GrB_Matrix          C,
4520                             const GrB_Matrix   Mask,
4521                             const GrB_BinaryOp  accum,
4522                             const GrB_Matrix   A,
```

```

4523         const GrB_Index      *row_indices,
4524         GrB_Index            nrows,
4525         const GrB_Index      *col_indices,
4526         GrB_Index            ncols,
4527         const GrB_Descriptor desc);

```

4528 Parameters

4529 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
4530 that may be accumulated with the result of the assign operation. On output, the
4531 matrix holds the results of the operation.

4532 **Mask** (IN) An optional “write” mask that controls which results from this operation are
4533 stored into the output matrix **C**. The mask dimensions must match those of the
4534 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4535 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
4536 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4537 dimensions of **C**), **GrB_NULL** should be specified.

4538 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4539 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4540 specified.

4541 **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4542 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
4543 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
4544 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
4545 this array may be manipulated by the caller after this operation returns without
4546 affecting any deferred computations for this operation. If this array contains du-
4547 plicate values, it implies assignment of more than one value to the same location
4548 which leads to undefined results.

4549 **nrows** (IN) The number of values in the **row_indices** array. Must be equal to **nrows(A)**
4550 if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4551 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns
4552 of **C** that are assigned. If all columns of **C** are to be assigned in order from 0
4553 to **ncols** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4554 and return value, this array may be manipulated by the caller after this operation
4555 returns without affecting any deferred computations for this operation. If this
4556 array contains duplicate values, it implies assignment of more than one value to
4557 the same location which leads to undefined results.

4558 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **ncols(A)** if **A** is
4559 not transposed, or equal to **nrows(A)** if **A** is transposed.

4560
4561
4562

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

4563

4564 Return Values

4565
4566
4567
4568
4569

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

4570

GrB_PANIC Unknown internal error.

4571
4572
4573
4574

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

4575

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

4576
4577

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

4578
4579
4580

GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to nrows(C), or a value in col_indices is greater than or equal to ncols(C). In non-blocking mode, this can be reported as an execution error.

4581
4582

GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrows \neq nrows(A), or ncols \neq ncols(A).

4583
4584
4585
4586

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4587
4588

GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices is a NULL pointer, or both.

4589 **Description**

4590 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows
 4591 and columns in `C` in a specified order: $C(\text{row_indices}, \text{col_indices}) = A$; or, if an optional binary
 4592 accumulation operator (\odot) is provided, $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot$
 4593 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned}
 & C(\text{row_indices}[i], \text{col_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 4594 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot A(i, j), \\
 & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4595 Logically, this operation occurs in three steps:

4596 Setup The internal matrices and mask used in the computation are formed and their domains
 4597 and dimensions are tested for compatibility.

4598 Compute The indicated computations are carried out.

4599 Output The result is written into the output matrix, possibly under control of a mask.

4600 Up to three argument matrices are used in the `GrB_assign` operation:

- 4601 1. `C` = $\langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4602 2. `Mask` = $\langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 4603 3. `A` = $\langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4604 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
 4605 ibility as follows:

- 4606 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 4607 must be from one of the pre-defined types of Table 3.2.
- 4608 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 4609 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4610 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4611 mulation operator.

4612 Two domains are compatible with each other if values from one domain can be cast to values in
 4613 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4614 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4615 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4616 error listed above is returned.

4617 From the arguments, the internal matrices, mask, and index arrays used in the computation are
 4618 formed (\leftarrow denotes copy):

- 4619 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4620 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
- 4621 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
4622 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 4623 (b) If `Mask \neq GrB_NULL`,
- 4624 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
4625 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
- 4626 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
4627 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
- 4628 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4629 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 4630 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4631 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
- 4632 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 4633 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4634 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
- 4635 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4636 The internal matrices and mask are checked for dimension compatibility. The following conditions
4637 must hold:

- 4638 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 4639 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 4640 3. $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$.
- 4641 4. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$.

4642 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4643 match error listed above is returned.

4644 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4645 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4646 We are now ready to carry out the assign and any additional associated operations. We describe
4647 this in terms of two intermediate vectors:

- 4648 • $\tilde{\mathbf{T}}$: The matrix holding the contents from $\tilde{\mathbf{A}}$ in their destination locations relative to $\tilde{\mathbf{C}}$.
- 4649 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4650 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4651 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4652 At this point, if any value in the $\tilde{\mathbf{I}}$ array is not in the range $[0, \mathbf{nrows}(\tilde{\mathbf{C}}))$ or any value in the
 4653 $\tilde{\mathbf{J}}$ array is not in the range $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-
 4654 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
 4655 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
 4656 this point forward in the sequence.

4657 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 4658 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$4659 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4660 \quad \{(i, j, Z_{ij}) \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4661 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 4662 of $\tilde{\mathbf{C}}$ ($\mathbf{ind}(\tilde{\mathbf{C}})$) and remove from it all the indices of $\tilde{\mathbf{C}}$ that are in the set of indices being
 4663 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

4664 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4665 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4666 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4667 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \\ 4668$$

4669 where the difference operator refers to set difference.

- 4670 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4671 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4672 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4673 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4674 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4675 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4676 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4677 \\ 4678$$

4679 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

4680 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4681 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4682 mask which acts as a “write mask”.

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in C on input to this operation are deleted and the content of the new output matrix, C, is defined as,

$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are copied into the result matrix, C, and elements of C that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.7.3 assign: Column variant

Assign the contents a vector to a subset of elements in one column of a matrix. Note that since the output cannot be transposed, a different variant of assign is provided to assign to a row of a matrix.

C Syntax

```

4700     GrB_Info GrB_assign(GrB_Matrix      C,
4701                       const GrB_Vector  mask,
4702                       const GrB_BinaryOp accum,
4703                       const GrB_Vector  u,
4704                       const GrB_Index   *row_indices,
4705                       GrB_Index        nrows,
4706                       GrB_Index        col_index,
4707                       const GrB_Descriptor desc);

```

Parameters

C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the assign operation. On output, this matrix holds the results of the operation.

mask (IN) An optional “write” mask that controls which results from this operation are stored into the specified column of the output matrix C. The mask dimensions must match those of a single column of the matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain of the Mask matrix must be of type

4716 bool or any of the predefined “built-in” types in Table 3.2. If the default mask
 4717 is desired (i.e., a mask that is all true with the dimensions of a column of C),
 4718 GrB_NULL should be specified.

4719 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 4720 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 4721 specified.

4722 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column
 4723 of C.

4724 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
 4725 the specified column of C that are to be assigned. If all elements of the column
 4726 in C are to be assigned in order from index 0 to `nrows - 1`, then GrB_ALL should
 4727 be specified. Regardless of execution mode and return value, this array may be
 4728 manipulated by the caller after this operation returns without affecting any de-
 4729 ferred computations for this operation. If this array contains duplicate values, it
 4730 implies in assignment of more than one value to the same location which leads to
 4731 undefined results.

4732 **nrows** (IN) The number of values in `row_indices` array. Must be equal to `size(u)`.

4733 **col_index** (IN) The index of the column in C to assign. Must be in the range `[0, ncols(C))`.

4734 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4735 should be specified. Non-default field/value pairs are listed as follows:

4736

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output column in C is cleared (all elements removed) before result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

4738 **Return Values**

4739 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 4740 blocking mode, this indicates that the compatibility tests on
 4741 dimensions and domains for the input arguments passed suc-
 4742 cessfully. Either way, output matrix C is ready to be used in the
 4743 next method of the sequence.

4744 GrB_PANIC Unknown internal error.

4745 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
4746 opaque GraphBLAS objects (input or output) is in an invalid
4747 state caused by a previous execution error. Call GrB_error() to
4748 access any error messages generated by the implementation.

4749 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4750 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
4751 by a call to new (or dup for vector or matrix parameters).

4752 GrB_INVALID_INDEX col_index is outside the allowable range (i.e., greater than ncols(C)).

4753 GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to nrows(C). In
4754 non-blocking mode, this can be reported as an execution error.

4755 GrB_DIMENSION_MISMATCH mask size and number of rows in C are not the same, or nrows ≠
4756 size(u).

4757 GrB_DOMAIN_MISMATCH The domains of the matrix and vector are incompatible with
4758 each other or the corresponding domains of the accumulation
4759 operator, or the mask's domain is not compatible with bool (in
4760 the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4761 GrB_NULL_POINTER Argument row_indices is a NULL pointer.

4762 Description

4763 This variant of GrB_assign computes the result of assigning a subset of locations in a column of a
4764 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:
4765 $C(:, \text{col_index}) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(:, \text{col_index}) =$
4766 $C(:, \text{col_index}) \odot u$. Taking order of row_indices into account, it is more explicitly written as:

$$4767 \quad C(\text{row_indices}[i], \text{col_index}) = u(i), \quad \forall i : 0 \leq i < \text{nrows}, \text{ or}$$

$$4767 \quad C(\text{row_indices}[i], \text{col_index}) = C(\text{row_indices}[i], \text{col_index}) \odot u(i), \quad \forall i : 0 \leq i < \text{nrows}.$$

4768 Logically, this operation occurs in three steps:

4769 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
4770 domains and dimensions are tested for compatibility.

4771 **Compute** The indicated computations are carried out.

4772 **Output** The result is written into the output matrix, possibly under control of a mask.

4773 Up to three argument vectors and matrices are used in this GrB_assign operation:

- 4774 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4775 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

4776 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4777 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
4778 compatibility as follows:

4779 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4780 must be from one of the pre-defined types of Table 3.2.

4781 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}(\mathbf{u})$.

4782 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4783 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4784 mulation operator.

4785 Two domains are compatible with each other if values from one domain can be cast to values in
4786 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4787 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4788 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4789 error listed above is returned.

4790 The `col_index` parameter is checked for a valid value. The following condition must hold:

4791 1. $0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{C})$

4792 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
4793 is returned.

4794 From the arguments, the internal vectors, `mask`, and index array used in the computation are
4795 formed (\leftarrow denotes copy):

4796 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a column of \mathbf{C} as follows:

4797
$$\tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4798 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

4799 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$.

4800 (b) If `mask \neq GrB_NULL`,

4801 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

4802 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

4803 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

4804 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4805 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:

4806 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.

4807 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i]$, $\forall i : 0 \leq i < \text{nrows}$.

4808 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
 4809 conditions must hold:

4810 1. $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$

4811 2. $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$.

4812 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
 4813 match error listed above is returned.

4814 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4815 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4816 We are now ready to carry out the assign and any additional associated operations. We describe
 4817 this in terms of two intermediate vectors:

- 4818 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 4819 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4820 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4821 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4822 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
 4823 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
 4824 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
 4825 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4826 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4827 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4828 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4829 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 4830 of $\tilde{\mathbf{c}}$ ($\text{ind}(\tilde{\mathbf{c}})$) and remove from it all the indices of $\tilde{\mathbf{c}}$ that are in the set of indices being
 4831 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4832 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4833 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$4834 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4835 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4837 where the difference operator refers to set difference.

4838 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4839 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4840 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4841 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4842 \quad z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$4843 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$4844 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

4847 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4848 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
4849 result matrix, $\mathbf{C}(:, \text{col_index})$. This is carried out under control of the mask which acts as a “write
4850 mask”.

4851 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(:, \text{col_index})$ on input to this
4852 operation are deleted and the new contents of the column is given by:

$$4853 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4854 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4855 copied into the column of the final result matrix, $\mathbf{C}(:, \text{col_index})$, and elements of this column
4856 that fall outside the set indicated by the mask are unchanged:

$$4857 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup$$

$$4858 \quad \{(i, \text{col_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$4859 \quad \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4860 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4861 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4862 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may
4863 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

4864 4.3.7.4 `assign`: Row variant

4865 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the
4866 output cannot be transposed, a different variant of `assign` is provided to assign to a column of a
4867 matrix.

4868 C Syntax

```
4869         GrB_Info GrB_assign(GrB_Matrix      C,  
4870                             const GrB_Vector  mask,  
4871                             const GrB_BinaryOp accum,  
4872                             const GrB_Vector  u,  
4873                             GrB_Index        row_index,  
4874                             const GrB_Index  *col_indices,  
4875                             GrB_Index        ncols,  
4876                             const GrB_Descriptor desc);
```

4877 Parameters

4878 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values
4879 that may be accumulated with the result of the assign operation. On output, this
4880 matrix holds the results of the operation.

4881 **mask** (IN) An optional “write” mask that controls which results from this operation are
4882 stored into the specified row of the output matrix **C**. The mask dimensions must
4883 match those of a single row of the matrix **C**. If the **GrB_STRUCTURE** descriptor
4884 is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or
4885 any of the predefined “built-in” types in Table 3.2. If the default mask is desired
4886 (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB_NULL** should
4887 be specified.

4888 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4889 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4890 specified.

4891 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of
4892 **C**.

4893 **row_index** (IN) The index of the row in **C** to assign. Must be in the range $[0, \mathbf{nrows}(\mathbf{C})]$.

4894 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4895 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to
4896 be assigned in order from index 0 to $\mathbf{ncols} - 1$, then **GrB_ALL** should be specified.
4897 Regardless of execution mode and return value, this array may be manipulated by
4898 the caller after this operation returns without affecting any deferred computations
4899 for this operation. If this array contains duplicate values, it implies in assignment
4900 of more than one value to the same location which leads to undefined results.

4901 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **size(u)**.

4902 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4903 should be specified. Non-default field/value pairs are listed as follows:
4904

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output row in C is cleared (all elements removed) before result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <code>mask</code> .

4905

4906 Return Values

- 4907 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
4908 blocking mode, this indicates that the compatibility tests on
4909 dimensions and domains for the input arguments passed suc-
4910 cessfully. Either way, output matrix C is ready to be used in the
4911 next method of the sequence.
- 4912 GrB_PANIC Unknown internal error.
- 4913 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
4914 opaque GraphBLAS objects (input or output) is in an invalid
4915 state caused by a previous execution error. Call `GrB_error()` to
4916 access any error messages generated by the implementation.
- 4917 GrB_OUT_OF_MEMORY Not enough memory available for operation.
- 4918 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
4919 by a call to `new` (or `dup` for vector or matrix parameters).
- 4920 GrB_INVALID_INDEX `row_index` is outside the allowable range (i.e., greater than `nrows(C)`).
- 4921 GrB_INDEX_OUT_OF_BOUNDS A value in `col_indices` is greater than or equal to `ncols(C)`. In
4922 non-blocking mode, this can be reported as an execution error.
- 4923 GrB_DIMENSION_MISMATCH `mask` size and number of columns in C are not the same, or
4924 `ncols` \neq `size(u)`.
- 4925 GrB_DOMAIN_MISMATCH The domains of the matrix and vector are incompatible with
4926 each other or the corresponding domains of the accumulation
4927 operator, or the mask's domain is not compatible with `bool` (in
4928 the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).
- 4929 GrB_NULL_POINTER Argument `col_indices` is a NULL pointer.

4930 Description

4931 This variant of `GrB_assign` computes the result of assigning a subset of locations in a row of a
4932 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

4933 $C(\text{row_index}, :) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(\text{row_index}, :$
 4934 $) = C(\text{row_index}, :) \odot u$. Taking order of `col_indices` into account it is more explicitly written as:

$$4935 \quad C(\text{row_index}, \text{col_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$$

$$C(\text{row_index}, \text{col_indices}[j]) = C(\text{row_index}, \text{col_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$$

4936 Logically, this operation occurs in three steps:

4937 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
 4938 domains and dimensions are tested for compatibility.

4939 **Compute** The indicated computations are carried out.

4940 **Output** The result is written into the output matrix, possibly under control of a mask.

4941 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 4942 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4943 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4944 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4945 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
 4946 compatibility as follows:

- 4947 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 4948 must be from one of the pre-defined types of Table 3.2.
- 4949 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(u)$.
- 4950 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4951 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4952 mulation operator.

4953 Two domains are compatible with each other if values from one domain can be cast to values in
 4954 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4955 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4956 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4957 error listed above is returned.

4958 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 4959 1. $0 \leq \text{row_index} < \mathbf{nrows}(C)$

4960 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
 4961 is returned.

4962 From the arguments, the internal vectors, mask, and index array used in the computation are
 4963 formed (\leftarrow denotes copy):

4964 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a row of \mathbf{C} as follows:

$$4965 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \mathbf{row_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4966 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

4967 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$.

4968 (b) If `mask \neq GrB_NULL`,

4969 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,

4970 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.

4971 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

4972 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4973 4. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:

4974 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.

4975 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4976 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
4977 conditions must hold:

4978 1. $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

4979 2. $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$.

4980 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4981 match error listed above is returned.

4982 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4983 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4984 We are now ready to carry out the assign and any additional associated operations. We describe
4985 this in terms of two intermediate vectors:

4986 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.

4987 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4988 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4989 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle$$

4990 At this point, if any value of $\tilde{\mathbf{J}}[j]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
4991 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
4992 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4993 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4994 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

4995 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4996 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4997 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 4998 of $\tilde{\mathbf{c}}$ ($\mathbf{ind}(\tilde{\mathbf{c}})$) and remove from it all the indices of $\tilde{\mathbf{c}}$ that are in the set of indices being
 4999 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5000 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5001 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$5002 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5003 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5004 where the difference operator refers to set difference.

5005 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5006 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5007 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5008 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5009 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5010 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5011 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5012 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5013 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
 5014 result matrix, $\mathbf{C}(\text{row_index}, :)$. This is carried out under control of the mask which acts as a “write
 5015 mask”.

5016 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(\text{row_index}, :)$ on input to this
 5017 operation are deleted and the new contents of the column is given by:

$$5018 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5019 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 5020 copied into the column of the final result matrix, $\mathbf{C}(\text{row_index}, :)$, and elements of this column
 5021 that fall outside the set indicated by the mask are unchanged:

$$5022 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup$$

$$5023 \quad \{(\text{row_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5024 \quad \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5025 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 5026 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 5027 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may
 5028 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5032 **4.3.7.5 assign: Constant vector variant**

5033 Assign the same value to a specified subset of vector elements. With the use of GrB_ALL, the entire
5034 destination vector can be filled with the constant.

5035 **C Syntax**

```
5036     GrB_Info GrB_assign(GrB_Vector      w,  
5037                       const GrB_Vector mask,  
5038                       const GrB_BinaryOp accum,  
5039                       <type>         val,  
5040                       const GrB_Index  *indices,  
5041                       GrB_Index       nindices,  
5042                       const GrB_Descriptor desc);
```

```
5043     GrB_Info GrB_assign(GrB_Vector      w,  
5044                       const GrB_Vector mask,  
5045                       const GrB_BinaryOp accum,  
5046                       const GrB_Scalar  s,  
5047                       const GrB_Index  *indices,  
5048                       GrB_Index       nindices,  
5049                       const GrB_Descriptor desc);
```

5050 **Parameters**

5051 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5052 that may be accumulated with the result of the assign operation. On output, this
5053 vector holds the results of the operation.

5054 **mask** (IN) An optional “write” mask that controls which results from this operation are
5055 stored into the output vector *w*. The mask dimensions must match those of the
5056 vector *w*. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5057 of the mask vector must be of type `bool` or any of the predefined “built-in” types
5058 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5059 dimensions of *w*), GrB_NULL should be specified.

5060 **accum** (IN) An optional binary operator used for accumulating entries into existing *w*
5061 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5062 specified.

5063 **val** (IN) Scalar value to assign to (a subset of) *w*.

5064 **s** (IN) Scalar value to assign to (a subset of) *w*.

5065 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
5066 *w* that are to be assigned. If all elements of *w* are to be assigned in order from 0

5067 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode
 5068 and return value, this array may be manipulated by the caller after this operation
 5069 returns without affecting any deferred computations for this operation. In this
 5070 variant, the specific order of the values in the array has no effect on the result.
 5071 Unlike other variants, if there are duplicated values in this array the result is still
 5072 defined.

5073 `nindices` (IN) The number of values in `indices` array. Must be in the range: $[0, \text{size}(\mathbf{w})]$. If
 5074 `nindices` is zero, the operation becomes a NO-OP.

5075 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 5076 should be specified. Non-default field/value pairs are listed as follows:

5077

Param	Field	Value	Description
<code>w</code>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <code>w</code> is cleared (all elements removed) before the result is stored in it.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <code>mask</code> .

5078

5079 Return Values

5080 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 5081 blocking mode, this indicates that the compatibility tests on
 5082 dimensions and domains for the input arguments passed suc-
 5083 cessfully. Either way, output vector `w` is ready to be used in the
 5084 next method of the sequence.

5085 `GrB_PANIC` Unknown internal error.

5086 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the
 5087 opaque GraphBLAS objects (input or output) is in an invalid
 5088 state caused by a previous execution error. Call `GrB_error()` to
 5089 access any error messages generated by the implementation.

5090 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

5091 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized
 5092 by a call to `new` (or `dup` for vector parameters).

5093 `GrB_INDEX_OUT_OF_BOUNDS` A value in `indices` is greater than or equal to `size(w)`. In non-
 5094 blocking mode, this can be reported as an execution error.

5095 `GrB_DIMENSION_MISMATCH` `mask` and `w` dimensions are incompatible, or `nindices` is not less
 5096 than `size(w)`.

5127 4. If `accum` is not `GrB_NULL`, then either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5128 method, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5129 Two domains are compatible with each other if values from one domain can be cast to values in
 5130 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 5131 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5132 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 5133 error listed above is returned.

5134 From the arguments, the internal vectors, mask and index array used in the computation are formed
 5135 (\leftarrow denotes copy):

- 5136 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5137 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 5138 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 5139 (b) If `mask \neq GrB_NULL`,
 - 5140 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,
 - 5141 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 5142 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5143 3. Scalar $\tilde{s} \leftarrow s$ (`GrB_Scalar` version only).
- 5144 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument `indices` as follows:
 - 5145 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.
 - 5146 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

5147 The internal vector and mask are checked for dimension compatibility. The following conditions
 5148 must hold:

- 5149 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5150 2. $0 \leq \text{nindices} \leq \text{size}(\tilde{\mathbf{w}})$.

5151 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
 5152 match error listed above is returned.

5153 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5154 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5155 We are now ready to carry out the assign and any additional associated operations. We describe
 5156 this in terms of two intermediate vectors:

- 5157 • $\tilde{\mathbf{t}}$: The vector holding the copies of the scalar, either `val` or \tilde{s} , in their destination locations
 5158 relative to $\tilde{\mathbf{w}}$.

5159 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5160 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows. If a non-opaque scalar val is provided:

5161
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\text{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \text{val}) \mid \forall i, 0 \leq i < \text{nindices}\} \rangle.$$

5162 Correspondingly, if a non-empty GrB_Scalar \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 1$):

5163
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \text{nindices}\} \rangle.$$

5164 Finally, if an empty GrB_Scalar \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 0$):

5165
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5166 If $\tilde{\mathbf{I}}$ is empty, this operation results in an empty vector, $\tilde{\mathbf{t}}$. Otherwise, if any value in the $\tilde{\mathbf{I}}$ array
 5167 is not in the range $[0, \mathbf{size}(\tilde{\mathbf{w}}))$, the execution of GrB_assign ends and the index out-of-bounds
 5168 error listed above is generated. In GrB_NONBLOCKING mode, the error can be deferred until a
 5169 sequence-terminating GrB_wait() is called. Regardless, the result vector, \mathbf{w} , is invalid from this
 5170 point forward in the sequence.

5171 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5172 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}}$ is defined as

5173
$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5174 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 5175 of $\tilde{\mathbf{w}}$ ($\mathbf{ind}(\tilde{\mathbf{w}})$) and remove from it all the indices of $\tilde{\mathbf{w}}$ that are in the set of indices being
 5176 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5177 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5178 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

5179
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5180
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5182 where the difference operator refers to set difference. We note that in this case of assigning
 5183 a constant, $\{\tilde{\mathbf{I}}[k], \forall k\}$ and $\mathbf{ind}(\tilde{\mathbf{t}})$ are identical.

5184 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

5185
$$\langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5186 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5187 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

5188
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

5189
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5190
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5193 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5194 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 5195 using what is called a *standard vector mask and replace*. This is carried out under control of the
 5196 mask which acts as a “write mask”.

- 5197 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 5198 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$5199 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5200 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 5201 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 5202 mask are unchanged:

$$5203 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5204 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 5205 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 5206 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 5207 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 5208 sequence.

5209 4.3.7.6 assign: Constant matrix variant

5210 Assign the same value to a specified subset of matrix elements. With the use of GrB_ALL, the
 5211 entire destination matrix can be filled with the constant.

5212 C Syntax

```
5213     GrB_Info GrB_assign(GrB_Matrix      C,
5214                       const GrB_Matrix Mask,
5215                       const GrB_BinaryOp accum,
5216                       <type>         val,
5217                       const GrB_Index  *row_indices,
5218                       GrB_Index        nrows,
5219                       const GrB_Index  *col_indices,
5220                       GrB_Index        ncols,
5221                       const GrB_Descriptor desc);
```

```
5222     GrB_Info GrB_assign(GrB_Matrix      C,
5223                       const GrB_Matrix Mask,
5224                       const GrB_BinaryOp accum,
5225                       const GrB_Scalar  s,
5226                       const GrB_Index  *row_indices,
5227                       GrB_Index        nrows,
```

```

5228         const GrB_Index      *col_indices,
5229         GrB_Index            ncols,
5230         const GrB_Descriptor desc);

```

5231 Parameters

5232 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5233 that may be accumulated with the result of the assign operation. On output, the
5234 matrix holds the results of the operation.

5235 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5236 stored into the output matrix **C**. The mask dimensions must match those of the
5237 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5238 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
5239 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5240 dimensions of **C**), **GrB_NULL** should be specified.

5241 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5242 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5243 specified.

5244 **val** (IN) Scalar value to assign to (a subset of) **C**.

5245 **s** (IN) Scalar value to assign to (a subset of) **C**.

5246 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
5247 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
5248 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
5249 this array may be manipulated by the caller after this operation returns without
5250 affecting any deferred computations for this operation. Unlike other variants, if
5251 there are duplicated values in this array the result is still defined.

5252 **nrows** (IN) The number of values in **row_indices** array. Must be in the range: [0, **nrows**(**C**)].
5253 If **nrows** is zero, the operation becomes a NO-OP.

5254 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**
5255 that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** – 1,
5256 then **GrB_ALL** should be specified. Regardless of execution mode and return value,
5257 this array may be manipulated by the caller after this operation returns without
5258 affecting any deferred computations for this operation. Unlike other variants, if
5259 there are duplicated values in this array the result is still defined.

5260 **ncols** (IN) The number of values in **col_indices** array. Must be in the range: [0, **ncols**(**C**)].
5261 If **ncols** is zero, the operation becomes a NO-OP.

5262 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5263 should be specified. Non-default field/value pairs are listed as follows:
5264

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.

5265

5266 Return Values

5267 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
5268 blocking mode, this indicates that the compatibility tests on
5269 dimensions and domains for the input arguments passed suc-
5270 cessfully. Either way, output matrix C is ready to be used in the
5271 next method of the sequence.

5272 GrB_PANIC Unknown internal error.

5273 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
5274 opaque GraphBLAS objects (input or output) is in an invalid
5275 state caused by a previous execution error. Call GrB_error() to
5276 access any error messages generated by the implementation.

5277 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

5278 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
5279 by a call to new (or dup for vector parameters).

5280 GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to nrows(C), or
5281 a value in col_indices is greater than or equal to ncols(C). In
5282 non-blocking mode, this can be reported as an execution error.

5283 GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrows is not less than
5284 nrows(C), or ncols is not less than ncols(C).

5285 GrB_DOMAIN_MISMATCH The domains of the matrix and scalar are incompatible with
5286 each other or the corresponding domains of the accumulation
5287 operator, or the mask's domain is not compatible with bool (in
5288 the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

5289 GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices
5290 is a NULL pointer, or both.

5291 Description

5292 This variant of GrB_assign computes the result of assigning a constant scalar value – either val or
5293 s – to locations in a destination GraphBLAS matrix: Either $C(\text{row_indices}, \text{col_indices}) = \text{val}$

5294 or $C(\text{row_indices}, \text{col_indices}) = s$ is performed. If an optional binary accumulation operator
 5295 (\odot) is provided, then either $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot \text{val}$ or
 5296 $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot s$ is performed. More explicitly, if a
 5297 non-opaque value val is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = \text{val}, \text{ or} \\ 5298 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5299 Correspondingly, if a `GrB_Scalar` s is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = s, \text{ or} \\ 5300 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5301 Logically, this operation occurs in three steps:

5302 Setup The internal vectors and mask used in the computation are formed and their domains
 5303 and dimensions are tested for compatibility.

5304 Compute The indicated computations are carried out.

5305 Output The result is written into the output matrix, possibly under control of a mask.

5306 Up to two argument matrices are used in the `GrB_assign` operation:

- 5307 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5308 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)

5309 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain
 5310 compatibility as follows:

- 5311 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 5312 must be from one of the pre-defined types of Table 3.2.
- 5313 2. $\mathbf{D}(C)$ must be compatible with either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5314 method.
- 5315 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5316 of the accumulation operator.
- 5317 4. If `accum` is not `GrB_NULL`, then either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5318 method, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5319 Two domains are compatible with each other if values from one domain can be cast to values in
5320 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5321 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5322 any compatibility rule above is violated, execution of GrB_assign ends and the domain mismatch
5323 error listed above is returned.

5324 From the arguments, the internal matrices, index arrays, and mask used in the computation are
5325 formed (\leftarrow denotes copy):

- 5326 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 5327 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument Mask as follows:
 - 5328 (a) If Mask = GrB_NULL, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
5329 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 5330 (b) If Mask \neq GrB_NULL,
 - 5331 i. If desc[GrB_MASK].GrB_STRUCTURE is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
5332 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - 5333 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
5334 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - 5335 (c) If desc[GrB_MASK].GrB_COMP is set, then $\tilde{\mathbf{M}} \leftarrow \neg\tilde{\mathbf{M}}$.
- 5336 3. Scalar $\tilde{s} \leftarrow s$ (GrB_Scalar version only).
- 5337 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument row_indices as follows:
 - 5338 (a) If row_indices = GrB_ALL, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
 - 5339 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 5340 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument col_indices as follows:
 - 5341 (a) If col_indices = GrB_ALL, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
 - 5342 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

5343 The internal matrix and mask are checked for dimension compatibility. The following conditions
5344 must hold:

- 5345 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 5346 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 5347 3. $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$.
- 5348 4. $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$.

5349 If any compatibility rule above is violated, execution of GrB_assign ends and the dimension mis-
5350 match error listed above is returned.

5351 From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with
 5352 GrB_SUCCESS return code and defer any computation and/or execution error codes.

5353 We are now ready to carry out the assign and any additional associated operations. We describe
 5354 this in terms of two intermediate matrices:

- 5355 • $\tilde{\mathbf{T}}$: The matrix holding the copies of the scalar, either val or \tilde{s} , in their destination locations
 5356 relative to $\tilde{\mathbf{C}}$.
- 5357 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5358 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows. If a non-opaque scalar val is provided:

$$5359 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5360 Correspondingly, if a non-empty GrB_Scalar \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 1$):

$$5361 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5362 Finally, if an empty GrB_Scalar \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 0$):

$$5363 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5364 If either $\tilde{\mathbf{I}}$ or $\tilde{\mathbf{J}}$ is empty, this operation results in an empty matrix, $\tilde{\mathbf{T}}$. Otherwise, if any value
 5365 in the $\tilde{\mathbf{I}}$ array is not in the range $[0, \mathbf{nrows}(\tilde{\mathbf{C}}))$ or any value in the $\tilde{\mathbf{J}}$ array is not in the range
 5366 $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of GrB_assign ends and the index out-of-bounds error listed above is
 5367 generated. In GrB_NONBLOCKING mode, the error can be deferred until a sequence-terminating
 5368 GrB_wait() is called. Regardless, the result matrix \mathbf{C} is invalid from this point forward in the
 5369 sequence.

5370 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 5371 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{Z}}$ is defined as

$$5372 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5373 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5374 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 5375 of $\tilde{\mathbf{C}}$ ($\mathbf{ind}(\tilde{\mathbf{C}})$) and remove from it all the indices of $\tilde{\mathbf{C}}$ that are in the set of indices being
 5376 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

5377 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 5378 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5379 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5380 \\ 5381 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

5382 where the difference operator refers to set difference. We note that, in this particular case of
 5383 assigning a constant to a matrix, the sets $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\}$ and $\mathbf{ind}(\tilde{\mathbf{T}})$ are identical.

5384 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5385 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \text{ind}(\tilde{\mathbf{C}}) \cup \text{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5386 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5387 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5388 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}})),$$

5389

$$5390 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{C}}) - (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}}))),$$

5391

$$5392 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{T}}) - (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}}))),$$

5393 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5394 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5395 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5396 mask which acts as a “write mask”.

5397 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5398 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5399 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\text{ind}(\tilde{\mathbf{Z}}) \cap \text{ind}(\tilde{\mathbf{M}}))\}.$$

5400 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
5401 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
5402 mask are unchanged:

$$5403 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\text{ind}(\mathbf{C}) \cap \text{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\text{ind}(\tilde{\mathbf{Z}}) \cap \text{ind}(\tilde{\mathbf{M}}))\}.$$

5404 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5405 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5406 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
5407 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
5408 sequence.

5409 4.3.8 apply: Apply a function to the elements of an object

5410 Computes the transformation of the values of the elements of a vector or a matrix using a unary
5411 function, or a binary function where one argument is bound to a scalar.

5412 4.3.8.1 apply: Vector variant

5413 Computes the transformation of the values of the elements of a vector using a unary function.

5414 **C Syntax**

```

5415     GrB_Info GrB_apply(GrB_Vector      w,
5416                       const GrB_Vector mask,
5417                       const GrB_BinaryOp accum,
5418                       const GrB_UnaryOp op,
5419                       const GrB_Vector u,
5420                       const GrB_Descriptor desc);

```

5421 **Parameters**

5422 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5423 that may be accumulated with the result of the apply operation. On output, this
5424 vector holds the results of the operation.

5425 **mask** (IN) An optional “write” mask that controls which results from this operation are
5426 stored into the output vector *w*. The mask dimensions must match those of the
5427 vector *w*. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
5428 of the mask vector must be of type `bool` or any of the predefined “built-in” types
5429 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5430 dimensions of *w*), `GrB_NULL` should be specified.

5431 **accum** (IN) An optional binary operator used for accumulating entries into existing *w*
5432 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
5433 specified.

5434 **op** (IN) A unary operator applied to each element of input vector *u*.

5435 **u** (IN) The GraphBLAS vector to which the unary function is applied.

5436 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
5437 should be specified. Non-default field/value pairs are listed as follows:

5438

Param	Field	Value	Description
<i>w</i>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <i>w</i> is cleared (all elements removed) before the result is stored in it.
<i>mask</i>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <i>mask</i> vector. The stored values are not examined.
<i>mask</i>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <i>mask</i> .

5439

5440 **Return Values**

5441 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5442 blocking mode, this indicates that the compatibility tests on di-
5443 mensions and domains for the input arguments passed successfully.

5444 Either way, output vector w is ready to be used in the next method
5445 of the sequence.

5446 GrB_PANIC Unknown internal error.

5447 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
5448 GraphBLAS objects (input or output) is in an invalid state caused
5449 by a previous execution error. Call `GrB_error()` to access any error
5450 messages generated by the implementation.

5451 GrB_OUT_OF_MEMORY Not enough memory available for operation.

5452 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
5453 a call to `new` (or `dup` for vector parameters).

5454 GrB_DIMENSION_MISMATCH `mask`, `w` and/or `u` dimensions are incompatible.

5455 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the corre-
5456 sponding domains of the accumulation operator or unary function,
5457 or the mask's domain is not compatible with `bool` (in the case where
5458 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

5459 Description

5460 This variant of `GrB_apply` computes the result of applying a unary function to the elements of a
5461 GraphBLAS vector: $w = f(u)$; or, if an optional binary accumulation operator (\odot) is provided,
5462 $w = w \odot f(u)$.

5463 Logically, this operation occurs in three steps:

5464 **Setup** The internal vectors and mask used in the computation are formed and their domains
5465 and dimensions are tested for compatibility.

5466 **Compute** The indicated computations are carried out.

5467 **Output** The result is written into the output vector, possibly under control of a mask.

5468 Up to three argument vectors are used in this `GrB_apply` operation:

5469 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

5470 2. $\mathit{mask} = \langle \mathbf{D}(\mathit{mask}), \mathbf{size}(\mathit{mask}), \mathbf{L}(\mathit{mask}) = \{(i, m_i)\} \rangle$ (optional)

5471 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5472 The argument vectors, unary operator and the accumulation operator (if provided) are tested for
5473 domain compatibility as follows:

- 5474 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5475 must be from one of the pre-defined types of Table 3.2.
- 5476 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5477 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5478 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5479 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5480 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in}(\text{op})$.

5481 Two domains are compatible with each other if values from one domain can be cast to values in
5482 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5483 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5484 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
5485 error listed above is returned.

5486 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
5487 denotes copy):

- 5488 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5489 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 5490 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
- 5491 (b) If `mask \neq GrB_NULL`,
- 5492 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
- 5493 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 5494 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5495 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5496 The internal vectors and masks are checked for dimension compatibility. The following conditions
5497 must hold:

- 5498 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5499 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

5500 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5501 error listed above is returned.

5502 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5503 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5504 We are now ready to carry out the apply and any additional associated operations. We describe
5505 this in terms of two intermediate vectors:

- 5506 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the unary operator to the input vector $\tilde{\mathbf{u}}$.
- 5507 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5508 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$5509 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

5510 where $f = \mathbf{f}(\text{op})$.

5511 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 5512 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 5513 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5514 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5515 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5516 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5517 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$5518 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5519 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5520 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5521 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5522 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5523 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
5524 using what is called a *standard vector mask and replace*. This is carried out under control of the
5525 mask which acts as a “write mask”.

- 5526 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
5527 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$5528 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5529 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
5530 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
5531 mask are unchanged:

$$5532 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5533 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5534 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5535 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
5536 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
5537 sequence.

5538 **4.3.8.2 apply: Matrix variant**

5539 Computes the transformation of the values of the elements of a matrix using a unary function.

5540 **C Syntax**

```

5541     GrB_Info GrB_apply(GrB_Matrix      C,
5542                       const GrB_Matrix Mask,
5543                       const GrB_BinaryOp accum,
5544                       const GrB_UnaryOp op,
5545                       const GrB_Matrix  A,
5546                       const GrB_Descriptor desc);

```

5547 **Parameters**

5548 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5549 that may be accumulated with the result of the apply operation. On output, the
5550 matrix holds the results of the operation.

5551 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5552 stored into the output matrix C. The mask dimensions must match those of the
5553 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5554 of the Mask matrix must be of type bool or any of the predefined “built-in” types
5555 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5556 dimensions of C), GrB_NULL should be specified.

5557 **accum** (IN) An optional binary operator used for accumulating entries into existing C
5558 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5559 specified.

5560 **op** (IN) A unary operator applied to each element of input matrix A.

5561 **A** (IN) The GraphBLAS matrix to which the unary function is applied.

5562 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
5563 should be specified. Non-default field/value pairs are listed as follows:

5564

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

5565

5566 **Return Values**

5567 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
5568 blocking mode, this indicates that the compatibility tests on
5569 dimensions and domains for the input arguments passed suc-
5570 cessfully. Either way, output matrix C is ready to be used in the
5571 next method of the sequence.

5572 GrB_PANIC Unknown internal error.

5573 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
5574 opaque GraphBLAS objects (input or output) is in an invalid
5575 state caused by a previous execution error. Call GrB_error() to
5576 access any error messages generated by the implementation.

5577 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

5578 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
5579 by a call to new (or Matrix_dup for matrix parameters).

5580 GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrows \neq nrows(C), or
5581 ncols \neq ncols(C).

5582 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
5583 corresponding domains of the accumulation operator or unary
5584 function, or the mask's domain is not compatible with bool (in
5585 the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

5586 **Description**

5587 This variant of GrB_apply computes the result of applying a unary function to the elements of a
5588 GraphBLAS matrix: $C = f(A)$; or, if an optional binary accumulation operator (\odot) is provided,
5589 $C = C \odot f(A)$.

5590 Logically, this operation occurs in three steps:

5591 **Setup** The internal matrices and mask used in the computation are formed and their domains
5592 and dimensions are tested for compatibility.

5593 **Compute** The indicated computations are carried out.

5594 **Output** The result is written into the output matrix, possibly under control of a mask.

5595 Up to three argument matrices are used in the GrB_apply operation:

- 5596 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
5597 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)

5598 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5599 The argument matrices, unary operator and the accumulation operator (if provided) are tested for
5600 domain compatibility as follows:

5601 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
5602 must be from one of the pre-defined types of Table 3.2.

5603 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.

5604 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5605 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5606 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5607 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in}(\text{op})$ of the unary operator.

5608 Two domains are compatible with each other if values from one domain can be cast to values in
5609 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5610 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5611 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
5612 error listed above is returned.

5613 From the argument matrices, the internal matrices, `mask`, and index arrays used in the computation
5614 are formed (\leftarrow denotes copy):

5615 1. Matrix $\tilde{C} \leftarrow C$.

5616 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:

5617 (a) If `Mask = GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
5618 $j < \mathbf{ncols}(C)\} \rangle$.

5619 (b) If `Mask \neq GrB_NULL`,

5620 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
5621 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,

5622 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
5623 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.

5624 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.

5625 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

5626 The internal matrices and mask are checked for dimension compatibility. The following conditions
5627 must hold:

5628 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.

5629 2. $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$.

5630 3. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$.

5631 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

5632 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5633 error listed above is returned.

5634 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5635 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5636 We are now ready to carry out the apply and any additional associated operations. We describe
5637 this in terms of two intermediate matrices:

- 5638 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the unary operator to the input matrix $\tilde{\mathbf{A}}$.
- 5639 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5640 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$5641 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5642 where $f = \mathbf{f}(\mathbf{op})$.

5643 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 5644 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 5645 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5646 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5647 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5648 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5649 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5650 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5651 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5652 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5653 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5654 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5655 mask which acts as a “write mask”.

- 5656 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5657 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5658 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are copied into the result matrix, C, and elements of C that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.8.3 apply: Vector-BinaryOp variants

Computes the transformation of the values of the stored elements of a vector using a binary operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the first argument to the binary operator and stored elements of the vector are passed as the second argument. In the *bind-second* variant, the elements of the vector are passed as the first argument and the specified scalar value is passed as the second argument. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

C Syntax

```

5678 // bind-first + scalar value
5679 GrB_Info GrB_apply(GrB_Vector          w,
5680                   const GrB_Vector     mask,
5681                   const GrB_BinaryOp   accum,
5682                   const GrB_BinaryOp   op,
5683                   <type>               val,
5684                   const GrB_Vector     u,
5685                   const GrB_Descriptor  desc);

5686 // bind-first + GraphBLAS scalar
5687 GrB_Info GrB_apply(GrB_Vector          w,
5688                   const GrB_Vector     mask,
5689                   const GrB_BinaryOp   accum,
5690                   const GrB_BinaryOp   op,
5691                   const GrB_Scalar     s,
5692                   const GrB_Vector     u,
5693                   const GrB_Descriptor  desc);

5694 // bind-second + scalar value
5695 GrB_Info GrB_apply(GrB_Vector          w,
5696                   const GrB_Vector     mask,
```

```

5697         const GrB_BinaryOp    accum,
5698         const GrB_BinaryOp    op,
5699         const GrB_Vector      u,
5700         <type>                val,
5701         const GrB_Descriptor   desc);

5702 // bind-second + GraphBLAS scalar
5703 GrB_Info GrB_apply(GrB_Vector      w,
5704                  const GrB_Vector  mask,
5705                  const GrB_BinaryOp accum,
5706                  const GrB_BinaryOp op,
5707                  const GrB_Vector  u,
5708                  const GrB_Scalar   s,
5709                  const GrB_Descriptor desc);

```

5710 Parameters

5711 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5712 that may be accumulated with the result of the apply operation. On output, this
5713 vector holds the results of the operation.

5714 **mask** (IN) An optional “write” mask that controls which results from this operation are
5715 stored into the output vector *w*. The mask dimensions must match those of the
5716 vector *w*. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
5717 of the mask vector must be of type `bool` or any of the predefined “built-in” types
5718 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5719 dimensions of *w*), `GrB_NULL` should be specified.

5720 **accum** (IN) An optional binary operator used for accumulating entries into existing *w*
5721 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
5722 specified.

5723 **op** (IN) A binary operator applied to each element of input vector, *u*, and the scalar
5724 value, *val*.

5725 **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as
5726 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
5727 argument in the *bind-second* variant.

5728 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
5729 argument in the *bind-first* variant, or the right-hand (second) argument in the
5730 *bind-second* variant.

5731 **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand
5732 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
5733 the *bind-second* variant. It must not be empty.

5734
5735
5736

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

5737

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask .

5738 Return Values

5739
5740
5741
5742
5743

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector **w** is ready to be used in the next method of the sequence.

5744

GrB_PANIC Unknown internal error.

5745
5746
5747
5748

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

5749

GrB_OUT_OF_MEMORY Not enough memory available for operation.

5750
5751

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector parameters).

5752

GrB_DIMENSION_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

5753
5754
5755
5756

GrB_DOMAIN_MISMATCH The domains of the various vectors and scalar are incompatible with the corresponding domains of the binary operator or accumulation operator, or the **mask**'s domain is not compatible with **bool** (in the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

5757
5758

GrB_EMPTY_OBJECT The GrB_Scalar **s** used in the call is empty (**nvals(s) = 0**) and therefore a value cannot be passed to the binary operator.

5759 Description

5760
5761

This variant of GrB_apply computes the result of applying a binary operator to the elements of a GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5762 bind-first: $w = f(\text{val}, u)$ or $w = f(s, u)$

5763 bind-second: $w = f(u, \text{val})$ or $w = f(u, s)$,

5764 or if an optional binary accumulation operator (\odot) is provided:

5765 bind-first: $w = w \odot f(\text{val}, u)$ or $w = w \odot f(s, u)$

5766 bind-second: $w = w \odot f(u, \text{val})$ or $w = w \odot f(u, s)$.

5767 Logically, this operation occurs in three steps:

5768 **Setup** The internal vectors and mask used in the computation are formed and their domains
5769 and dimensions are tested for compatibility.

5770 **Compute** The indicated computations are carried out.

5771 **Output** The result is written into the output vector, possibly under control of a mask.

5772 Up to three argument vectors are used in this GrB_apply operation:

5773 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

5774 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

5775 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5776 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are
5777 tested for domain compatibility as follows:

5778 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
5779 must be from one of the pre-defined types of Table 3.2.

5780 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.

5781 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5782 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
5783 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5784 4. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

5785 5. If bind-first:

5786 (a) $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

5787 (b) If the non-opaque scalar **val** is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
5788 of the binary operator.

5789 (c) If the GrB_Scalar **s** is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
5790 binary operator.

- 5791 6. If bind-second:
- 5792 (a) $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the binary operator.
- 5793 (b) If the non-opaque scalar \mathbf{val} is provided, then $\mathbf{D}(\mathbf{val})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
- 5794 of the binary operator.
- 5795 (c) If the `GrB_Scalar` \mathbf{s} is provided, then $\mathbf{D}(\mathbf{s})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the
- 5796 binary operator.

5797 Two domains are compatible with each other if values from one domain can be cast to values in

5798 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5799 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5800 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5801 error listed above is returned.

5802 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow

5803 denotes copy):

- 5804 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5805 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 5806 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 5807 (b) If `mask \neq GrB_NULL`,
- 5808 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 5809 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 5810 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5811 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 5812 4. Scalar $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$ (GraphBLAS scalar case).

5813 The internal vectors and masks are checked for dimension compatibility. The following conditions

5814 must hold:

- 5815 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5816 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

5817 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5818 error listed above is returned.

5819 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5820 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5821 If an empty `GrB_Scalar` $\tilde{\mathbf{s}}$ is provided ($\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

5822 If a non-empty `GrB_Scalar`, $\tilde{\mathbf{s}}$, is provided (i.e., $\mathbf{nvals}(\tilde{\mathbf{s}}) = 1$), we then create an internal variable

5823 `val` with the same domain as $\tilde{\mathbf{s}}$ and set `val = val($\tilde{\mathbf{s}}$)`.

5824 We are now ready to carry out the apply and any additional associated operations. We describe

5825 this in terms of two intermediate vectors:

- 5826 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the binary operator to the input vector $\tilde{\mathbf{u}}$.
- 5827 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5828 The intermediate vector, $\tilde{\mathbf{t}}$, is created as one of the following:

$$5829 \quad \text{bind-first:} \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\text{val}, \tilde{\mathbf{u}}(i))) \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

$$5830 \quad \text{bind-second:} \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i), \text{val})) \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

5831 where $f = \mathbf{f}(\text{op})$.

5832 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 5833 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 5834 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5835 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5836 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5837 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5838 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$5839 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5840 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5841 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5842 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5843 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5844 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
5845 using what is called a *standard vector mask and replace*. This is carried out under control of the
5846 mask which acts as a “write mask”.

- 5847 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
5848 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$5849 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5850 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
5851 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
5852 mask are unchanged:

$$5853 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5854 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5855 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5856 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
5857 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
5858 sequence.

5859 4.3.8.4 apply: Matrix-BinaryOp variants

5860 Computes the transformation of the values of the stored elements of a matrix using a binary
5861 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the
5862 first argument to the binary operator and stored elements of the matrix are passed as the second
5863 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument
5864 and the specified scalar value is passed as the second argument. The scalar can be passed either as
5865 a non-opaque variable or as a GrB_Scalar object.

5866 C Syntax

```
5867 // bind-first + scalar value
5868 GrB_Info GrB_apply(GrB_Matrix      C,
5869                  const GrB_Matrix  Mask,
5870                  const GrB_BinaryOp accum,
5871                  const GrB_BinaryOp op,
5872                  <type>           val,
5873                  const GrB_Matrix  A,
5874                  const GrB_Descriptor desc);
```

```
5875 // bind-first + GraphBLAS scalar
5876 GrB_Info GrB_apply(GrB_Matrix      C,
5877                  const GrB_Matrix  Mask,
5878                  const GrB_BinaryOp accum,
5879                  const GrB_BinaryOp op,
5880                  const GrB_Scalar  s,
5881                  const GrB_Matrix  A,
5882                  const GrB_Descriptor desc);
```

```
5883 // bind-second + scalar value
5884 GrB_Info GrB_apply(GrB_Matrix      C,
5885                  const GrB_Matrix  Mask,
5886                  const GrB_BinaryOp accum,
5887                  const GrB_BinaryOp op,
5888                  const GrB_Matrix  A,
5889                  <type>           val,
5890                  const GrB_Descriptor desc);
```

```
5891 // bind-second + GraphBLAS scalar
5892 GrB_Info GrB_apply(GrB_Matrix      C,
5893                  const GrB_Matrix  Mask,
5894                  const GrB_BinaryOp accum,
5895                  const GrB_BinaryOp op,
5896                  const GrB_Matrix  A,
```

```
5897         const GrB_Scalar      s,  
5898         const GrB_Descriptor desc);
```

5899 Parameters

5900 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5901 that may be accumulated with the result of the apply operation. On output, the
5902 matrix holds the results of the operation.

5903 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5904 stored into the output matrix C. The mask dimensions must match those of the
5905 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5906 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
5907 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5908 dimensions of C), GrB_NULL should be specified.

5909 **accum** (IN) An optional binary operator used for accumulating entries into existing C
5910 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5911 specified.

5912 **op** (IN) A binary operator applied to each element of input matrix, A, with the element
5913 of the input matrix used as the left-hand argument, and the scalar value, val, used
5914 as the right-hand argument.

5915 **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as
5916 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
5917 argument in the *bind-second* variant.

5918 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
5919 argument in the *bind-first* variant, or the right-hand (second) argument in the
5920 *bind-second* variant.

5921 **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand
5922 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
5923 the *bind-second* variant. It must not be empty.

5924 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
5925 should be specified. Non-default field/value pairs are listed as follows:
5926

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation (<i>bind-second</i> variant only).
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation (<i>bind-first</i> variant only).

5927

5928 Return Values

5929 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
5930 blocking mode, this indicates that the compatibility tests on
5931 dimensions and domains for the input arguments passed suc-
5932 cessfully. Either way, output matrix C is ready to be used in the
5933 next method of the sequence.

5934 GrB_PANIC Unknown internal error.

5935 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
5936 opaque GraphBLAS objects (input or output) is in an invalid
5937 state caused by a previous execution error. Call GrB_error() to
5938 access any error messages generated by the implementation.

5939 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

5940 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
5941 by a call to new (or Matrix_dup for matrix parameters).

5942 GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to **nrows(A)**, or
5943 a value in col_indices is greater than or equal to **ncols(A)**. In
5944 non-blocking mode, this can be reported as an execution error.

5945 GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, **nrows** \neq **nrows(C)**, or
5946 **ncols** \neq **ncols(C)**.

5947 GrB_DOMAIN_MISMATCH The domains of the various matrices and scalar are incompatible
5948 with the corresponding domains of the binary operator or accu-
5949 mulation operator, or the mask's domain is not compatible with
5950 bool (in the case where desc[GrB_MASK].GrB_STRUCTURE is
5951 not set).

5952 GrB_EMPTY_OBJECT The GrB_Scalar s used in the call is empty (**nvals(s) = 0**) and
5953 therefore a value cannot be passed to the binary operator.

5954 **Description**

5955 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a
5956 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

5957 bind-first: $C = f(\text{val}, A)$ or $C = f(s, A)$

5958 bind-second: $C = f(A, \text{val})$ or $C = f(A, s)$,

5959 or if an optional binary accumulation operator (\odot) is provided:

5960 bind-first: $C = C \odot f(\text{val}, A)$ or $C = C \odot f(s, A)$

5961 bind-second: $C = C \odot f(A, \text{val})$ or $C = C \odot f(A, s)$.

5962 Logically, this operation occurs in three steps:

5963 **Setup** The internal matrices and mask used in the computation are formed and their domains
5964 and dimensions are tested for compatibility.

5965 **Compute** The indicated computations are carried out.

5966 **Output** The result is written into the output matrix, possibly under control of a mask.

5967 Up to three argument matrices are used in the `GrB_apply` operation:

5968 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

5969 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)

5970 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5971 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are
5972 tested for domain compatibility as follows:

5973 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
5974 must be from one of the pre-defined types of Table 3.2.

5975 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.

5976 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5977 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
5978 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5979 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

5980 5. If bind-first:

5981 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

5982 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
 5983 of the binary operator.

5984 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
 5985 binary operator.

5986 6. If `bind-second`:

5987 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

5988 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$
 5989 of the binary operator.

5990 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the
 5991 binary operator.

5992 Two domains are compatible with each other if values from one domain can be cast to values in
 5993 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 5994 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5995 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
 5996 error listed above is returned.

5997 From the argument matrices, the internal matrices, `mask`, and index arrays used in the computation
 5998 are formed (\leftarrow denotes copy):

5999 1. Matrix $\tilde{C} \leftarrow C$.

6000 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:

6001 (a) If `Mask = GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
 6002 $j < \mathbf{ncols}(C)\} \rangle$.

6003 (b) If `Mask \neq GrB_NULL`,

6004 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
 6005 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,

6006 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
 6007 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.

6008 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg\tilde{M}$.

6009 3. Matrix \tilde{A} is computed from argument `A` as follows:

6010 `bind-first`: $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? A^T : A$

6011 `bind-second`: $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$

6012 4. Scalar $\tilde{s} \leftarrow s$ (`GraphBLAS` scalar case).

6013 The internal matrices and `mask` are checked for dimension compatibility. The following conditions
 6014 must hold:

6015 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.

6016 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.

6017 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6018 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6019 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6020 error listed above is returned.

6021 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6022 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6023 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

6024 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6025 `val` with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6026 We are now ready to carry out the apply and any additional associated operations. We describe
6027 this in terms of two intermediate matrices:

6028 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the binary operator to the input matrix $\tilde{\mathbf{A}}$.

6029 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6030 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as one of the following:

6031 bind-first: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$,

6032 bind-second: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$,

6033 where $f = \mathbf{f}(\mathbf{op})$.

6034 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

6035 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

6036 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

6037
$$\tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle$$
.

6038 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6039 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

6040
$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

6041
$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6042
$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6043 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6046 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 6047 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 6048 mask which acts as a “write mask”.

- 6049 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 6050 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6051 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6052 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 6053 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 6054 mask are unchanged:

$$6055 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6056 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 6057 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 6058 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 6059 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 6060 sequence.

6061 4.3.8.5 apply: Vector index unary operator variant

6062 Computes the transformation of the values of the stored elements of a vector using an index unary
 6063 operator that is a function of the stored value, its location indices, and an user provided scalar
 6064 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6065 C Syntax

```
6066     GrB_Info GrB_apply(GrB_Vector      w,
6067                       const GrB_Vector mask,
6068                       const GrB_BinaryOp accum,
6069                       const GrB_IndexUnaryOp op,
6070                       const GrB_Vector u,
6071                       <type> val,
6072                       const GrB_Descriptor desc);
```

```
6073     GrB_Info GrB_apply(GrB_Vector      w,
6074                       const GrB_Vector mask,
6075                       const GrB_BinaryOp accum,
6076                       const GrB_IndexUnaryOp op,
6077                       const GrB_Vector u,
6078                       const GrB_Scalar s,
6079                       const GrB_Descriptor desc);
```

6080 **Parameters**

- 6081 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 6082 that may be accumulated with the result of the apply operation. On output, this
 6083 vector holds the results of the operation.
- 6084 **mask** (IN) An optional “write” mask that controls which results from this operation are
 6085 stored into the output vector **w**. The mask dimensions must match those of the
 6086 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
 6087 of the mask vector must be of type `bool` or any of the predefined “built-in” types
 6088 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
 6089 dimensions of **w**), `GrB_NULL` should be specified.
- 6090 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
 6091 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 6092 specified.
- 6093 **op** (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB_Index}), D_{in_2}, f_i \rangle$, applied
 6094 to each element stored in the input vector, **u**. It is a function of the stored element’s
 6095 value, its location index, and a user supplied scalar value (either **s** or **val**).
- 6096 **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-
 6097 ator.
- 6098 **val** (IN) An additional scalar value that is passed to the index unary operator.
- 6099 **s** (IN) An additional GraphBLAS scalar that is passed to the index unary operator.
 6100 It must not be empty.
- 6101 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 6102 should be specified. Non-default field/value pairs are listed as follows:

6103

Param	Field	Value	Description
w	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of mask .

6104

6105 **Return Values**

- 6106 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 6107 blocking mode, this indicates that the compatibility tests on di-
 6108 mensions and domains for the input arguments passed success-
 6109 fully. Either way, output vector **w** is ready to be used in the next
 6110 method of the sequence.

6111 GrB_PANIC Unknown internal error.

6112 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6113 opaque GraphBLAS objects (input or output) is in an invalid
6114 state caused by a previous execution error. Call GrB_error() to
6115 access any error messages generated by the implementation.

6116 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6117 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6118 by a call to new (or another constructor).

6119 GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

6120 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
6121 responding domains of the accumulation operator or index unary
6122 operator, or the mask's domain is not compatible with bool (in
6123 the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

6124 GrB_EMPTY_OBJECT The GrB_Scalar s used in the call is empty ($\mathbf{nvals}(s) = 0$) and
6125 therefore a value cannot be passed to the index unary operator.

6126 Description

6127 This variant of GrB_apply computes the result of applying an index unary operator to the elements
6128 of a GraphBLAS vector each composed with the element's index and a scalar constant, val or s:

$$6129 \quad \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \text{val}) \text{ or } \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}),$$

6130 or if an optional binary accumulation operator (\odot) is provided:

$$6131 \quad \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \text{val}) \text{ or } \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}).$$

6132 Logically, this operation occurs in three steps:

6133 **Setup** The internal vectors and mask used in the computation are formed and their domains
6134 and dimensions are tested for compatibility.

6135 **Compute** The indicated computations are carried out.

6136 **Output** The result is written into the output vector, possibly under control of a mask.

6137 Up to three argument vectors are used in this GrB_apply operation:

- 6138 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6139 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)

6140 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6141 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6142 are tested for domain compatibility as follows:

6143 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
6144 must be from one of the pre-defined types of Table 3.2.

6145 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the index unary operator.

6146 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6147 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be compatible
6148 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

6149 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.

6150 5. If the non-opaque scalar `val` is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of
6151 the index unary operator.

6152 6. If the `GrB_Scalar` `s` is provided, then $\mathbf{D}(\text{s})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the index
6153 unary operator.

6154 Two domains are compatible with each other if values from one domain can be cast to values in
6155 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6156 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6157 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
6158 error listed above is returned.

6159 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6160 denotes copy):

6161 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.

6162 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

6163 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.

6164 (b) If `mask \neq GrB_NULL`,

6165 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

6166 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

6167 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

6168 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

6169 4. Scalar $\tilde{s} \leftarrow \text{s}$ (GraphBLAS scalar case).

6170 The internal vectors and masks are checked for dimension compatibility. The following conditions
6171 must hold:

6172 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$

6173 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

6174 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6175 error listed above is returned.

6176 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6177 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6178 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

6179 If a non-empty `GrB_Scalar`, \tilde{s} , is provided ($\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable `val`
6180 with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6181 We are now ready to carry out the apply and any additional associated operations. We describe
6182 this in terms of two intermediate vectors:

- 6183 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6184 $\tilde{\mathbf{u}}$.
- 6185 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6186 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6187 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \mathbf{val})) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6188 where $f_i = \mathbf{f}(\mathbf{op})$.

6189 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6190 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 6191 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6192 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6193 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6194 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6195 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6196 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6197 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6198 \quad & \\ 6199 \end{aligned}$$

6200 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6201 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector `w`,
6202 using what is called a *standard vector mask and replace*. This is carried out under control of the
6203 mask which acts as a “write mask”.

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w , is defined as,

$$\mathbf{L}(w) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{z}) \cap \mathbf{ind}(\tilde{m}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of \tilde{z} indicated by the mask are copied into the result vector, w , and elements of w that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(w) = \{(i, w_i) : i \in (\mathbf{ind}(w) \cap \mathbf{ind}(\neg\tilde{m}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{z}) \cap \mathbf{ind}(\tilde{m}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.8.6 apply: Matrix index unary operator variant

Computes the transformation of the values of the stored elements of a matrix using an index unary operator that is a function of the stored value, its location indices, and an user provided scalar value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

C Syntax

```
GrB_Info GrB_apply(GrB_Matrix      C,
                  const GrB_Matrix  Mask,
                  const GrB_BinaryOp accum,
                  const GrB_IndexUnaryOp op,
                  const GrB_Matrix  A,
                  <type>           val,
                  const GrB_Descriptor desc);
```

```
GrB_Info GrB_apply(GrB_Matrix      C,
                  const GrB_Matrix  Mask,
                  const GrB_BinaryOp accum,
                  const GrB_IndexUnaryOp op,
                  const GrB_Matrix  A,
                  const GrB_Scalar  s,
                  const GrB_Descriptor desc);
```

Parameters

C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the apply operation. On output, the matrix holds the results of the operation.

6239 Mask (IN) An optional “write” mask that controls which results from this operation are
 6240 stored into the output matrix C. The mask dimensions must match those of the
 6241 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 6242 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
 6243 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
 6244 dimensions of C), GrB_NULL should be specified.

6245 accum (IN) An optional binary operator used for accumulating entries into existing C
 6246 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 6247 specified.

6248 op (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB_Index}), D_{in_2}, f_i \rangle$, applied
 6249 to each element stored in the input matrix, A. It is a function of the stored element’s
 6250 value, its row and column indices, and a user supplied scalar value (either `s` or `val`).

6251 A (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
 6252 ator.

6253 val (IN) An additional scalar value that is passed to the index unary operator.

6254 s (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6255 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 6256 should be specified. Non-default field/value pairs are listed as follows:

6257

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

6258

6259 Return Values

6260 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 6261 blocking mode, this indicates that the compatibility tests on di-
 6262 mensions and domains for the input arguments passed successfully.
 6263 Either way, output matrix C is ready to be used in the next method
 6264 of the sequence.

6265 GrB_PANIC Unknown internal error.

6266 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 6267 GraphBLAS objects (input or output) is in an invalid state caused

6268 by a previous execution error. Call `GrB_error()` to access any error
6269 messages generated by the implementation.

6270 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

6271 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
6272 a call to `new` (or another constructor).

6273 **GrB_DIMENSION_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6274 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
6275 corresponding domains of the accumulation operator or index unary
6276 operator, or the mask's domain is not compatible with `bool` (in the
6277 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6278 **GrB_EMPTY_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and
6279 therefore a value cannot be passed to the index unary operator.

6280 **Description**

6281 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements
6282 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar
6283 constant, `val` or `s`:

$$6284 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathit{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6285 or if an optional binary accumulation operator (\odot) is provided:

$$6286 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathit{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6287 Where the `row` and `col` functions extract the row and column indices from a list of two-dimensional
6288 indices, respectively.

6289 Logically, this operation occurs in three steps:

6290 **Setup** The internal matrices and mask used in the computation are formed and their domains
6291 and dimensions are tested for compatibility.

6292 **Compute** The indicated computations are carried out.

6293 **Output** The result is written into the output matrix, possibly under control of a mask.

6294 Up to three argument matrices are used in the `GrB_apply` operation:

- 6295 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6296 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)

6297 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6298 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6299 are tested for domain compatibility as follows:

- 6300 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
6301 must be from one of the pre-defined types of Table 3.2.
- 6302 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the index unary operator.
- 6303 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6304 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be compatible
6305 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6306 4. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6307 5. If the non-opaque scalar `val` is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of
6308 the index unary operator.
- 6309 6. If the `GrB_Scalar` `s` is provided, then $\mathbf{D}(\text{s})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the index
6310 unary operator.

6311 Two domains are compatible with each other if values from one domain can be cast to values in
6312 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6313 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6314 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
6315 error listed above is returned.

6316 From the argument matrices, the internal matrices, `mask`, and index arrays used in the computation
6317 are formed (\leftarrow denotes copy):

- 6318 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 6319 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
- 6320 (a) If `Mask` = `GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
6321 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 6322 (b) If `Mask` \neq `GrB_NULL`,
- 6323 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
6324 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
- 6325 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
6326 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
- 6327 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 6328 3. Matrix $\tilde{\mathbf{A}}$ is computed from argument `A` as follows:
- 6329
$$\tilde{\mathbf{A}} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$$
- 6330 4. Scalar $\tilde{s} \leftarrow \text{s}$ (GraphBLAS scalar case).

6331 The internal matrices and mask are checked for dimension compatibility. The following conditions
 6332 must hold:

- 6333 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 6334 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 6335 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 6336 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6337 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
 6338 error listed above is returned.

6339 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 6340 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6341 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
 6342 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
 6343 `val` with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6344 We are now ready to carry out the apply and any additional associated operations. We describe
 6345 this in terms of two intermediate matrices:

- 6346 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
 6347 $\tilde{\mathbf{A}}$.
- 6348 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6349 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6350 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6351 where $f_i = \mathbf{f}(\mathbf{op})$.

6352 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6353 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6354 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6355 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6356 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 6357 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6358 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6359 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6360 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6361 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6362 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6363 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6364 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 6365 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 6366 mask which acts as a “write mask”.

- 6367 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 6368 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6369 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6370 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 6371 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 6372 mask are unchanged:

$$6373 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6374 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 6375 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 6376 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 6377 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 6378 sequence.

6379 4.3.9 select:

6380 Apply a select operator to the stored elements of an object to determine whether or not to keep
 6381 them.

6382 4.3.9.1 select: Vector variant

6383 Apply a select operator (an index unary operator) to the elements of a vector.

6384 C Syntax

```
6385 // scalar value variant
6386 GrB_Info GrB_select(GrB_Vector          w,
6387                   const GrB_Vector     mask,
6388                   const GrB_BinaryOp   accum,
6389                   const GrB_IndexUnaryOp op,
6390                   const GrB_Vector     u,
6391                   <type>               val,
6392                   const GrB_Descriptor  desc);
6393
6394 // GraphBLAS scalar variant
6395 GrB_Info GrB_select(GrB_Vector          w,
6396                   const GrB_Vector     mask,
```

```

6397         const GrB_BinaryOp      accum,
6398         const GrB_IndexUnaryOp  op,
6399         const GrB_Vector        u,
6400         const GrB_Scalar        s,
6401         const GrB_Descriptor    desc);
6402

```

6403 Parameters

6404 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6405 that may be accumulated with the result of the select operation. On output, this
6406 vector holds the results of the operation.

6407 **mask** (IN) An optional “write” mask that controls which results from this operation are
6408 stored into the output vector **w**. The mask dimensions must match those of the
6409 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
6410 of the mask vector must be of type `bool` or any of the predefined “built-in” types
6411 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
6412 dimensions of **w**), `GrB_NULL` should be specified.

6413 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6414 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
6415 specified.

6416 **op** (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB_Index}), D_{in_2}, f_i \rangle$, applied
6417 to each element stored in the input vector, **u**. It is a function of the stored element’s
6418 value, its location index, and a user supplied scalar value (either **s** or **val**).

6419 **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-
6420 ator.

6421 **val** (IN) An additional scalar value that is passed to the index unary operator.

6422 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6423 not be empty.

6424 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
6425 should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of mask .

6427

6428 Return Values

6429 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
6430 blocking mode, this indicates that the compatibility tests on di-
6431 mensions and domains for the input arguments passed success-
6432 fully. Either way, output vector w is ready to be used in the next
6433 method of the sequence.

6434 GrB_PANIC Unknown internal error.

6435 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6436 opaque GraphBLAS objects (input or output) is in an invalid
6437 state caused by a previous execution error. Call `GrB_error()` to
6438 access any error messages generated by the implementation.

6439 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6440 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6441 by a call to one of its constructors.

6442 GrB_DIMENSION_MISMATCH $mask$, w and/or u dimensions are incompatible.

6443 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
6444 responding domains of the accumulation operator or index unary
6445 operator, or the mask's domain is not compatible with `bool` (in
6446 the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6447 GrB_EMPTY_OBJECT The `GrB_Scalar` s used in the call is empty ($nvals(s) = 0$) and
6448 therefore a value cannot be passed to the index unary operator.

6449 Description

6450 This variant of `GrB_select` computes the result of applying a index unary operator to select the
6451 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored
6452 element, along with the element's index and a scalar constant – either `val` or `s`. The corresponding
6453 element of the input vector is selected (kept) if the function evaluates to `true` when cast to `bool`.
6454 This acts like a functional mask on the input vector as follows:

$$6455 \quad w = u \langle f_i(u, \mathbf{ind}(u), 0, val) \rangle,$$

$$6456 \quad w = w \odot u \langle f_i(u, \mathbf{ind}(u), 0, val) \rangle.$$

6457 Correspondingly, if a `GrB_Scalar`, s , is provided:

$$6458 \quad w = u \langle f_i(u, \mathbf{ind}(u), 0, s) \rangle,$$

$$6459 \quad w = w \odot u \langle f_i(u, \mathbf{ind}(u), 0, s) \rangle.$$

6460 Logically, this operation occurs in three steps:

6461 **Setup** The internal vectors and mask used in the computation are formed and their domains
6462 and dimensions are tested for compatibility.

6463 **Compute** The indicated computations are carried out.

6464 **Output** The result is written into the output vector, possibly under control of a mask.

6465 Up to three argument vectors are used in this GrB_select operation:

- 6466 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6467 2. $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$ (optional)
- 6468 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

6469 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6470 are tested for domain compatibility as follows:

- 6471 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(mask)$
6472 must be from one of the pre-defined types of Table 3.2.
- 6473 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(u)$.
- 6474 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(accum)$ and $\mathbf{D}_{out}(accum)$
6475 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(accum)$ of the accu-
6476 mulation operator.
- 6477 4. $\mathbf{D}_{out}(op)$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
6478 i.e., castable to `bool`.
- 6479 5. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(op)$ of the index unary operator.
- 6480 6. $\mathbf{D}(val)$ or $\mathbf{D}(s)$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(op)$
6481 of the index unary operator.

6482 Two domains are compatible with each other if values from one domain can be cast to values in
6483 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6484 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6485 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch
6486 error listed above is returned.

6487 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6488 denotes copy):

- 6489 1. Vector $\tilde{w} \leftarrow w$.
- 6490 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:

- 6491 (a) If $\text{mask} = \text{GrB_NULL}$, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
- 6492 (b) If $\text{mask} \neq \text{GrB_NULL}$,
- 6493 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
- 6494 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 6495 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 6496 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 6497 4. Scalar $\tilde{s} \leftarrow s$ (GrB_Scalar version only).

6498 The internal vectors and masks are checked for dimension compatibility. The following conditions
6499 must hold:

- 6500 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 6501 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

6502 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch
6503 error listed above is returned.

6504 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6505 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6506 If an empty `GrB_Scalar` \tilde{s} is provided (i.e., $\text{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6507 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\text{nvals}(\tilde{s}) = 1$), we then create an internal variable
6508 `val` with the same domain as \tilde{s} and set $\text{val} = \text{val}(\tilde{s})$.

6509 We are now ready to carry out the `select` and any additional associated operations. We describe
6510 this in terms of two intermediate vectors:

- 6511 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6512 $\tilde{\mathbf{u}}$.
- 6513 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6514 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6515 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{u}}), \{(i, \tilde{\mathbf{u}}(i)), : i \in \text{ind}(\tilde{\mathbf{u}}) \wedge (\text{bool})f_i(\tilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true}\} \rangle,$$

6516 where $f_i = \mathbf{f}(\text{op})$.

6517 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6518 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 6519 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6520 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{\text{out}}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6521 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 6522 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned}
 6523 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\
 6524 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 6525 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 6526 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 6527 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),
 \end{aligned}$$

6528 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6529 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 6530 using what is called a *standard vector mask and replace*. This is carried out under control of the
 6531 mask which acts as a “write mask”.

- 6532 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 6533 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$6534 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 6535 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 6536 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 6537 mask are unchanged:

$$6538 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

6539 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 6540 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 6541 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 6542 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 6543 sequence.

6544 4.3.9.2 select: Matrix variant

6545 Apply a select operator (an index unary operator) to the elements of a matrix.

6546 C Syntax

```

6547 // scalar value variant
6548 GrB_Info GrB_select(GrB_Matrix          C,
6549                   const GrB_Matrix     Mask,
6550                   const GrB_BinaryOp   accum,
6551                   const GrB_IndexUnaryOp op,
6552                   const GrB_Matrix     A,
6553                   <type>               val,
6554                   const GrB_Descriptor desc);
6555

```

```

6556 // GraphBLAS scalar variant
6557 GrB_Info GrB_select(GrB_Matrix          C,
6558                   const GrB_Matrix     Mask,
6559                   const GrB_BinaryOp    accum,
6560                   const GrB_IndexUnaryOp op,
6561                   const GrB_Matrix      A,
6562                   const GrB_Scalar      s,
6563                   const GrB_Descriptor   desc);

```

6564 Parameters

6565 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6566 that may be accumulated with the result of the select operation. On output, the
6567 matrix holds the results of the operation.

6568 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6569 stored into the output matrix **C**. The mask dimensions must match those of the
6570 matrix **C**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
6571 of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types
6572 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
6573 dimensions of **C**), `GrB_NULL` should be specified.

6574 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
6575 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
6576 specified.

6577 **op** (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB_Index}), D_{in_2}, f_i \rangle$, applied
6578 to each element stored in the input matrix, **A**. It is a function of the stored element’s
6579 value, its row and column indices, and a user supplied scalar value (either `s` or `val`).

6580 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6581 ator.

6582 **val** (IN) An additional scalar value that is passed to the index unary operator.

6583 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6584 not be empty.

6585 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
6586 should be specified. Non-default field/value pairs are listed as follows:
6587

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

6588

6589 Return Values

6590 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
6591 blocking mode, this indicates that the compatibility tests on di-
6592 mensions and domains for the input arguments passed successfully.
6593 Either way, output matrix C is ready to be used in the next method
6594 of the sequence.

6595 GrB_PANIC Unknown internal error.

6596 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
6597 GraphBLAS objects (input or output) is in an invalid state caused
6598 by a previous execution error. Call GrB_error() to access any error
6599 messages generated by the implementation.

6600 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6601 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
6602 a call to one of its constructors.

6603 GrB_DIMENSION_MISMATCH Mask, C and/or A dimensions are incompatible.

6604 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
6605 corresponding domains of the accumulation operator or index unary
6606 operator, or the mask's domain is not compatible with bool (in the
6607 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

6608 GrB_EMPTY_OBJECT The GrB_Scalar s used in the call is empty (**nvals**(s) = 0) and
6609 therefore a value cannot be passed to the index unary operator.

6610 Description

6611 This variant of GrB_select computes the result of applying a index unary operator to select the
6612 elements of the input GraphBLAS matrix. The operator takes, as input, the value of each stored
6613 element, along with the element's row and column indices and a scalar constant – from either **val**
6614 or **s**. The corresponding element of the input matrix is selected (kept) if the function evaluates
6615 to true when cast to bool. This acts like a functional mask on the input matrix as follows when
6616 specifying a transparent scalar value:

6617 $C = A\langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \rangle$, or
 6618 $C = C \odot A\langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \rangle$.

6619 Correspondingly, if a GrB_Scalar, s , is provided:

6620 $C = A\langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s) \rangle$, or
 6621 $C = C \odot A\langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s) \rangle$.

6622 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
 6623 indices, respectively.

6624 Logically, this operation occurs in three steps:

6625 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6626 and dimensions are tested for compatibility.

6627 **Compute** The indicated computations are carried out.

6628 **Output** The result is written into the output matrix, possibly under control of a mask.

6629 Up to three argument matrices are used in the GrB_select operation:

- 6630 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6631 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 6632 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6633 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
 6634 are tested for domain compatibility as follows:

- 6635 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
 6636 must be from one of the pre-defined types of Table 3.2.
- 6637 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 6638 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6639 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 6640 mulation operator.
- 6641 4. $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
 6642 i.e., castable to bool.
- 6643 5. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6644 6. $\mathbf{D}(\mathbf{val})$ or $\mathbf{D}(s)$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\text{op})$
 6645 of the index unary operator.

6646 Two domains are compatible with each other if values from one domain can be cast to values in
6647 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6648 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6649 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch
6650 error listed above is returned.

6651 From the argument matrices, the internal matrices, `mask`, and index arrays used in the computation
6652 are formed (\leftarrow denotes copy):

- 6653 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 6654 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 6655 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
6656 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 6657 (b) If `Mask \neq GrB_NULL`,
 - 6658 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
6659 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 6660 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
6661 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 6662 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 6663 3. Matrix $\tilde{\mathbf{A}}$ is computed from argument `A` as follows: $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6664 4. Scalar $\tilde{s} \leftarrow s$ (`GrB_Scalar` version only).

6665 The internal matrices and mask are checked for dimension compatibility. The following conditions
6666 must hold:

- 6667 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 6668 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 6669 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 6670 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6671 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch
6672 error listed above is returned.

6673 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6674 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6675 If an empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6676 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6677 `val` with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6678 We are now ready to carry out the `select` and any additional associated operations. We describe
6679 this in terms of two intermediate matrices:

- 6680 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6681 $\tilde{\mathbf{A}}$.
- 6682 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6683 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6684 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\mathbf{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val}) = \mathbf{true})\},$$

6685 where $f_i = \mathbf{f}(\mathbf{op})$.

6686 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6687 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6688 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6689 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

6690 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6691 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6692 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6693 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6694 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6695 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6696 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6697 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6698 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6699 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6700 mask which acts as a “write mask”.

- 6701 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
6702 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6703 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6704 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6705 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6706 mask are unchanged:

$$6707 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6708 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6709 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6710 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
6711 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6712 sequence.

6713 **4.3.10 reduce: Perform a reduction across the elements of an object**

6714 Computes the reduction of the values of the elements of a vector or matrix.

6715 **4.3.10.1 reduce: Standard matrix to vector variant**

6716 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns
6717 is desired, the input matrix should be transposed using the descriptor.

6718 **C Syntax**

```
6719     GrB_Info GrB_reduce(GrB_Vector      w,  
6720                       const GrB_Vector mask,  
6721                       const GrB_BinaryOp accum,  
6722                       const GrB_Monoid op,  
6723                       const GrB_Matrix A,  
6724                       const GrB_Descriptor desc);  
6725  
6726     GrB_Info GrB_reduce(GrB_Vector      w,  
6727                       const GrB_Vector mask,  
6728                       const GrB_BinaryOp accum,  
6729                       const GrB_BinaryOp op,  
6730                       const GrB_Matrix A,  
6731                       const GrB_Descriptor desc);
```

6732 **Parameters**

6733 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6734 that may be accumulated with the result of the reduction operation. On output,
6735 this vector holds the results of the operation.

6736 **mask** (IN) An optional “write” mask that controls which results from this operation are
6737 stored into the output vector *w*. The mask dimensions must match those of the
6738 vector *w*. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
6739 of the mask vector must be of type `bool` or any of the predefined “built-in” types
6740 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
6741 dimensions of *w*), `GrB_NULL` should be specified.

6742 **accum** (IN) An optional binary operator used for accumulating entries into existing *w*
6743 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
6744 specified.

6745 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.
6746 Depending on which type is passed, the following defines the binary operator with
6747 one domain, $F_b = \langle D, D, D, \oplus \rangle$, that is used:

6748

BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

6749

Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$, the identity element of the monoid is ignored.

6750

6751

If `op` is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in both cases $\odot(\text{op})$ must be commutative and associative. Otherwise, the outcome of the operation is undefined.

6752

6753

6754

A (IN) The GraphBLAS matrix on which reduction will be performed.

6755

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL` should be specified. Non-default field/value pairs are listed as follows:

6756

6757

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

6758

6759 Return Values

6760

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

6761

6762

6763

6764

6765

GrB_PANIC Unknown internal error.

6766

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call `GrB_error()` to access any error messages generated by the implementation.

6767

6768

6769

6770

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

6771

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to `new` (or `dup` for vector parameters).

6772

6773

GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

6774

GrB_DOMAIN_MISMATCH Either the domains of the various vectors and matrices are incompatible with the corresponding domains of the accumulation operator or reduce function, or the domains of the GraphBLAS binary

6775

6776

6777 operator `op` are not all the same, or the mask's domain is not com-
 6778 patible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE`
 6779 is not set).

6780 Description

6781 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows
 6782 of an input matrix: $w(i) = \bigoplus A(i, :)\forall i$; or, if an optional binary accumulation operator is provided,
 6783 $w(i) = w(i) \odot (\bigoplus A(i, :))\forall i$, where $\bigoplus = \odot(F_b)$ and $\odot = \odot(\text{accum})$.

6784 Logically, this operation occurs in three steps:

6785 **Setup** The internal vector, matrix and mask used in the computation are formed and their
 6786 domains and dimensions are tested for compatibility.

6787 **Compute** The indicated computations are carried out.

6788 **Output** The result is written into the output vector, possibly under control of a mask.

6789 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6790 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6791 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 6792 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6793 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested
 6794 for domain compatibility as follows:

- 6795 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 6796 must be from one of the pre-defined types of Table 3.2.
- 6797 2. $\mathbf{D}(w)$ must be compatible with the domain of the reduction binary operator, $\mathbf{D}(F_b)$.
- 6798 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6799 of the accumulation operator and $\mathbf{D}(F_b)$, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 6800 mulation operator.
- 6801 4. $\mathbf{D}(A)$ must be compatible with the domain of the binary reduction operator, $\mathbf{D}(F_b)$.

6802 Two domains are compatible with each other if values from one domain can be cast to values in
 6803 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6804 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6805 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 6806 error listed above is returned.

6807 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 6808 denotes copy):

- 6809 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6810 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 6811 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 6812 (b) If `mask \neq GrB_NULL`,
- 6813 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 6814 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 6815 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 6816 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

6817 The internal vectors and masks are checked for dimension compatibility. The following conditions
6818 must hold:

- 6819 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6820 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6821 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-
6822 match error listed above is returned.

6823 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6824 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6825 We carry out the reduce and any additional associated operations. We describe this in terms of
6826 two intermediate vectors:

- 6827 • $\tilde{\mathbf{t}}$: The vector holding the result from reducing along the rows of input matrix $\tilde{\mathbf{A}}$.
- 6828 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6829 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6830 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6831 The value of each of its elements is computed by

$$6832 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6833 where $\bigoplus = \odot(F_b)$.

6834 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6835 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

6836 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6837 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6838 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6839 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$6840 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

$$6841$$

$$6842 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$6843$$

$$6844 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

6845 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6846 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
6847 using what is called a *standard vector mask and replace*. This is carried out under control of the
6848 mask which acts as a “write mask”.

6849 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
6850 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$6851 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6852 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
6853 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
6854 mask are unchanged:

$$6855 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6856 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6857 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6858 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
6859 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6860 sequence.

6861 4.3.10.2 reduce: Vector-scalar variant

6862 Reduce all stored values into a single scalar.

6863 C Syntax

```
6864 // scalar value + monoid (only)
6865 GrB_Info GrB_reduce(<type>          *val,
6866                      const GrB_BinaryOp accum,
6867                      const GrB_Monoid  op,
6868                      const GrB_Vector  u,
```

```

6869             const GrB_Descriptor desc);
6870
6871 // GraphBLAS Scalar + monoid
6872 GrB_Info GrB_reduce(GrB_Scalar      s,
6873                   const GrB_BinaryOp accum,
6874                   const GrB_Monoid  op,
6875                   const GrB_Vector  u,
6876                   const GrB_Descriptor desc);
6877
6878 // GraphBLAS Scalar + binary operator
6879 GrB_Info GrB_reduce(GrB_Scalar      s,
6880                   const GrB_BinaryOp accum,
6881                   const GrB_BinaryOp op,
6882                   const GrB_Vector  u,
6883                   const GrB_Descriptor desc);

```

6884 Parameters

6885 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
6886 a value that may be accumulated (optionally) with the result of the reduction
6887 operation. On output, this scalar holds the results of the operation.

6888 **accum** (IN) An optional binary operator used for accumulating entries into an exist-
6889 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,
6890 **GrB_NULL** should be specified.

6891 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
6892 the reduction operation. The \oplus operator must be commutative and associative;
6893 otherwise, the outcome of the operation is undefined.

6894 **u** (IN) The GraphBLAS vector on which reduction will be performed.

6895 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6896 should be specified. Non-default field/value pairs are listed as follows:

6898	Param	Field	Value	Description
------	-------	-------	-------	-------------

6899 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
6900 tions. There are currently no non-default field/value pairs that can be set for this
6901 operation.

6902 Return Values

6903 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
6904 cessfully, and the output scalar (**s** or **val**) is ready to be used in the
6905 next method of the sequence.

6906 GrB_PANIC Unknown internal error.

6907 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
6908 GraphBLAS objects (input or output) is in an invalid state caused
6909 by a previous execution error. Call GrB_error() to access any error
6910 messages generated by the implementation.

6911 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

6912 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
6913 a call to a respective constructor.

6914 GrB_NULL_POINTER val pointer is NULL.

6915 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
6916 the corresponding domains of the accumulation operator, or reduce
6917 operator.

6918 Description

This variant of GrB_reduce computes the result of performing a reduction across all of the stored elements of an input vector storing the result into either s or val. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[\bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

6919 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

6920 Logically, this operation occurs in three steps:

6921 **Setup** The internal vector used in the computation is formed and its domain is tested for
6922 compatibility.

6923 **Compute** The indicated computations are carried out.

6924 **Output** The result is written into the output scalar.

6925 One vector argument is used in this GrB_reduce operation:

6926 1. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6927 The output scalar, argument vector, reduction operator and accumulation operator (if provided)
6928 are tested for domain compatibility as follows:

6929 1. If accum is GrB_NULL, then $\mathbf{D}(\text{val})$ or $\mathbf{D}(\text{s})$ must be compatible with $\mathbf{D}(\text{op})$ from M (or with
6930 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

6931 2. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\text{val})$ or $\mathbf{D}(\text{s})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and
 6932 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 6933 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

6934 3. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}(\text{op})$ from M (or with $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

6935 Two domains are compatible with each other if values from one domain can be cast to values in
 6936 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6937 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6938 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 6939 error listed above is returned.

6940 The number of values stored in the input, \mathbf{u} , is checked. If there are no stored values in \mathbf{u} , then one
 6941 of the following occurs depending on the output variant:

$$6942 \quad \mathbf{L}(\mathbf{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \mathbf{L}(\mathbf{s}), & \text{(unchanged) otherwise,} \end{cases}$$

6943 OR

$$6944 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

6945 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

6946 For all other cases, the internal vector and scalar used in the computation is formed (\leftarrow denotes
 6947 copy):

6948 1. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

6949 2. Scalar $\tilde{s} \leftarrow \mathbf{s}$ (GraphBLAS scalar case).

6950 We are now ready to carry out the reduction and any additional associated operations. An inter-
 6951 mediate scalar result t is computed as follows:

$$6952 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

6953 where $\oplus = \odot(\text{op})$.

6954 The final reduction value is computed as follows:

$$6955 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

6956 OR

$$6957 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB_NULL}, \text{ or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

6958 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
6959 GrB_SUCCESS and the new contents of the output scalar is as defined above.

6960 4.3.10.3 reduce: Matrix-scalar variant

6961 Reduce all stored values into a single scalar.

6962 C Syntax

```
6963 // scalar value + monoid (only)
6964 GrB_Info GrB_reduce(<type>          *val,
6965                   const GrB_BinaryOp accum,
6966                   const GrB_Monoid   op,
6967                   const GrB_Matrix   A,
6968                   const GrB_Descriptor desc);
6969
6970 // GraphBLAS Scalar + monoid
6971 GrB_Info GrB_reduce(GrB_Scalar      s,
6972                   const GrB_BinaryOp accum,
6973                   const GrB_Monoid   op,
6974                   const GrB_Matrix   A,
6975                   const GrB_Descriptor desc);
6976
6977 // GraphBLAS Scalar + binary operator
6978 GrB_Info GrB_reduce(GrB_Scalar      s,
6979                   const GrB_BinaryOp accum,
6980                   const GrB_BinaryOp op,
6981                   const GrB_Matrix   A,
6982                   const GrB_Descriptor desc);
```

6983 Parameters

6984 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
6985 a value that may be accumulated (optionally) with the result of the reduction
6986 operation. On output, this scalar holds the results of the operation.

6987 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or
6988 **val**) value. If assignment rather than accumulation is desired, GrB_NULL should
6989 be specified.

6990 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
6991 the reduction operation. The \oplus operator must be commutative and associative;
6992 otherwise, the outcome of the operation is undefined.

6993 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

6994 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 6995 should be specified. Non-default field/value pairs are listed as follows:
 6996

6997	Param	Field	Value	Description
------	-------	-------	-------	-------------

6998 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
 6999 tions. There are currently no non-default field/value pairs that can be set for this
 7000 operation.

7001 Return Values

7002 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
 7003 cessfully, and the output scalar (s or val) is ready to be used in the
 7004 next method of the sequence.

7005 GrB_PANIC Unknown internal error.

7006 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 7007 GraphBLAS objects (input or output) is in an invalid state caused
 7008 by a previous execution error. Call GrB_error() to access any error
 7009 messages generated by the implementation.

7010 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7011 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 7012 a call to a respective constructor.

7013 GrB_NULL_POINTER val pointer is NULL.

7014 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
 7015 the corresponding domains of the accumulation operator, or reduce
 7016 operator.

7017 Description

This variant of GrB_reduce computes the result of performing a reduction across all of the stored elements of an input matrix storing the result into either s or val. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[\bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the the optional accumulator is specified.} \end{cases}$$

7018 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7019 Logically, this operation occurs in three steps:

7020 **Setup** The internal matrix used in the computation is formed and its domain is tested for
 7021 compatibility.

7022 **Compute** The indicated computations are carried out.

7023 **Output** The result is written into the output scalar.

7024 One matrix argument is used in this GrB_reduce operation:

7025 1. $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7026 The output scalar, argument matrix, reduction operator and accumulation operator (if provided)
 7027 are tested for domain compatibility as follows:

7028 1. If `accum` is `GrB_NULL`, then $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$ must be compatible with $\mathbf{D}(\text{op})$ from M (or with
 7029 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

7030 2. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and
 7031 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 7032 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

7033 3. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}(\text{op})$ from M (or with $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

7034 Two domains are compatible with each other if values from one domain can be cast to values in
 7035 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7036 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7037 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 7038 error listed above is returned.

7039 The number of values stored in the input, A , is checked. If there are no stored values in A , then
 7040 one of the following occurs depending on the output variant:

$$7041 \quad \mathbf{L}(s) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \mathbf{L}(s), & \text{(unchanged) otherwise,} \end{cases}$$

7042 or

$$7043 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7044 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7045 For all other cases, the internal matrix and scalar used in the computation is formed (\leftarrow denotes
 7046 copy):

7047 1. Matrix $\tilde{A} \leftarrow A$.

7048 2. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

7049 We are now ready to carry out the reduce and any additional associated operations. An intermediate
 7050 scalar result t is computed as follows:

$$7051 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7052 where $\oplus = \odot(\text{op})$.

7053 The final reduction value is computed as follows:

$$7054 \quad \mathbf{L}(s) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7055 or

$$7056 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7057 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
 7058 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7059 4.3.11 transpose: Transpose rows and columns of a matrix

7060 This version computes a new matrix that is the transpose of the source matrix.

7061 C Syntax

```
7062     GrB_Info GrB_transpose(GrB_Matrix      C,
7063                          const GrB_Matrix Mask,
7064                          const GrB_BinaryOp accum,
7065                          const GrB_Matrix  A,
7066                          const GrB_Descriptor desc);
```

7067 Parameters

7068 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 7069 that may be accumulated with the result of the transpose operation. On output,
 7070 the matrix holds the results of the operation.

7071 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 7072 stored into the output matrix C. The mask dimensions must match those of the
 7073 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 7074 of the Mask matrix must be of type bool or any of the predefined “built-in” types
 7075 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
 7076 dimensions of C), GrB_NULL should be specified.

7077 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
 7078 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 7079 specified.

7080 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7081 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 7082 should be specified. Non-default field/value pairs are listed as follows:
 7083

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask .
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

7085 **Return Values**

7086 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 7087 blocking mode, this indicates that the compatibility tests on di-
 7088 mensions and domains for the input arguments passed successfully.
 7089 Either way, output matrix **C** is ready to be used in the next method
 7090 of the sequence.

7091 **GrB_PANIC** Unknown internal error.

7092 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 7093 GraphBLAS objects (input or output) is in an invalid state caused
 7094 by a previous execution error. Call **GrB_error()** to access any error
 7095 messages generated by the implementation.

7096 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

7097 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 7098 a call to **new** (or **Matrix_dup** for matrix parameters).

7099 **GrB_DIMENSION_MISMATCH** **mask**, **C** and/or **A** dimensions are incompatible.

7100 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the cor-
 7101 responding domains of the accumulation operator, or the mask's do-
 7102 main is not compatible with **bool** (in the case where **desc[GrB_MASK].GrB_STRUCT**
 7103 is not set).

7104 **Description**

7105 GrB_transpose computes the result of performing a transpose of the input matrix: $C = A^T$; or, if an
7106 optional binary accumulation operator (\odot) is provided, $C = C \odot A^T$. We note that the input matrix
7107 A can itself be optionally transposed before the operation, which would cause either an assignment
7108 from A to C or an accumulation of A into C.

7109 Logically, this operation occurs in three steps:

7110 **Setup** The internal matrix and mask used in the computation are formed and their domains
7111 and dimensions are tested for compatibility.

7112 **Compute** The indicated computations are carried out.

7113 **Output** The result is written into the output matrix, possibly under control of a mask.

7114 Up to three matrix arguments are used in this GrB_transpose operation:

- 7115 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7116 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 7117 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7118 The argument matrices and accumulation operator (if provided) are tested for domain compatibility
7119 as follows:

- 7120 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
7121 must be from one of the pre-defined types of Table 3.2.
- 7122 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$ of the input matrix.
- 7123 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
7124 of the accumulation operator and $\mathbf{D}(A)$ of the input matrix must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
7125 of the accumulation operator.

7126 Two domains are compatible with each other if values from one domain can be cast to values in
7127 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
7128 compatible with each other. A domain from a user-defined type is only compatible with itself. If
7129 any compatibility rule above is violated, execution of GrB_transpose ends and the domain mismatch
7130 error listed above is returned.

7131 From the argument matrices, the internal matrices and mask used in the computation are formed
7132 (\leftarrow denotes copy):

- 7133 1. Matrix $\tilde{C} \leftarrow C$.
- 7134 2. Two-dimensional mask, \tilde{M} , is computed from argument Mask as follows:

- 7135 (a) If $\text{Mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
7136 $j < \mathbf{ncols}(\mathbf{C})\}$.
- 7137 (b) If $\text{Mask} \neq \text{GrB_NULL}$,
- 7138 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
7139 $(i, j) \in \mathbf{ind}(\text{Mask})\}$,
- 7140 ii. Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
7141 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\}$.
- 7142 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}$.
- 7143 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

7144 The internal matrices and masks are checked for dimension compatibility. The following conditions
7145 must hold:

- 7146 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
- 7147 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
- 7148 3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.
- 7149 4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

7150 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension
7151 mismatch error listed above is returned.

7152 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
7153 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7154 We are now ready to carry out the matrix transposition and any additional associated operations.
7155 We describe this in terms of two intermediate matrices:

- 7156 • $\widetilde{\mathbf{T}}$: The matrix holding the transpose of $\widetilde{\mathbf{A}}$.
- 7157 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

7158 The intermediate matrix

$$7159 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7160 is created.

7161 The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7162 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- 7163 • If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$7164 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7165 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
7166 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned}
7167 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
7168 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
7169 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
7170 \\
7171
\end{aligned}$$

7172 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7173 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
7174 using what is called a *standard matrix mask and replace*. This is carried out under control of the
7175 mask which acts as a “write mask”.

- 7176 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
7177 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7178 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7179 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
7180 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
7181 mask are unchanged:

$$7182 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7183 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
7184 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
7185 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
7186 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7187 sequence.

7188 4.3.12 kronecker: Kronecker product of two matrices

7189 Computes the Kronecker product of two matrices. The result is a matrix.

7190 C Syntax

```

7191      GrB_Info GrB_kronecker(GrB_Matrix      C,
7192                          const GrB_Matrix  Mask,
7193                          const GrB_BinaryOp accum,
7194                          const GrB_Semiring op,
7195                          const GrB_Matrix  A,
7196                          const GrB_Matrix  B,
7197                          const GrB_Descriptor desc);
7198

```

```

7199     GrB_Info GrB_kronecker(GrB_Matrix      C,
7200                          const GrB_Matrix  Mask,
7201                          const GrB_BinaryOp accum,
7202                          const GrB_Monoid   op,
7203                          const GrB_Matrix  A,
7204                          const GrB_Matrix  B,
7205                          const GrB_Descriptor desc);
7206
7207     GrB_Info GrB_kronecker(GrB_Matrix      C,
7208                          const GrB_Matrix  Mask,
7209                          const GrB_BinaryOp accum,
7210                          const GrB_BinaryOp op,
7211                          const GrB_Matrix  A,
7212                          const GrB_Matrix  B,
7213                          const GrB_Descriptor desc);

```

7214 Parameters

7215 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
7216 that may be accumulated with the result of the Kronecker product. On output,
7217 the matrix holds the results of the operation.

7218 **Mask** (IN) An optional “write” mask that controls which results from this operation are
7219 stored into the output matrix C. The mask dimensions must match those of the
7220 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
7221 of the Mask matrix must be of type bool or any of the predefined “built-in” types
7222 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
7223 dimensions of C), GrB_NULL should be specified.

7224 **accum** (IN) An optional binary operator used for accumulating entries into existing C
7225 entries. If assignment rather than accumulation is desired, GrB_NULL should be
7226 specified.

7227 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
7228 operation. Depending on which type is passed, the following defines the binary
7229 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

7230 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

7231 Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ig-
7232 nored.

7233 Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$; the additive monoid
7234 is ignored.

7235 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
7236 product.

7237
7238
7239
7240
7241
7242

B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the product.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for the operation.

7243 Return Values

7244
7245
7246
7247
7248
7249
7250
7251
7252
7253
7254
7255
7256
7257
7258
7259
7260
7261

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the corresponding domains of the binary operator (op) or accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

7262 Description

7263 GrB_kronecker computes the Kronecker product $C = A \otimes B$ or, if an optional binary accumulation
7264 operator (\odot) is provided, $C = C \odot (A \otimes B)$ (where matrices A and B can be optionally transposed).

7265 The Kronecker product is defined as follows:

7266

$$7267 \quad \mathbf{C} = \mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{0,0} \otimes \mathbf{B} & A_{0,1} \otimes \mathbf{B} & \dots & A_{0,n_A-1} \otimes \mathbf{B} \\ A_{1,0} \otimes \mathbf{B} & A_{1,1} \otimes \mathbf{B} & \dots & A_{1,n_A-1} \otimes \mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes \mathbf{B} & A_{m_A-1,1} \otimes \mathbf{B} & \dots & A_{m_A-1,n_A-1} \otimes \mathbf{B} \end{bmatrix}$$

7268 where $\mathbf{A} : \mathbb{S}^{m_A \times n_A}$, $\mathbf{B} : \mathbb{S}^{m_B \times n_B}$, and $\mathbf{C} : \mathbb{S}^{m_A m_B \times n_A n_B}$. More explicitly, the elements of the
7269 Kronecker product are defined as

$$7270 \quad \mathbf{C}(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7271 where \otimes is the multiplicative operator specified by the `op` parameter.

7272 Logically, this operation occurs in three steps:

7273 **Setup** The internal matrices and mask used in the computation are formed and their domains
7274 and dimensions are tested for compatibility.

7275 **Compute** The indicated computations are carried out.

7276 **Output** The result is written into the output matrix, possibly under control of a mask.

7277 Up to four argument matrices are used in the `GrB_kronecker` operation:

- 7278 1. $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 7279 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 7280 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 7281 4. $\mathbf{B} = \langle \mathbf{D}(\mathbf{B}), \mathbf{nrows}(\mathbf{B}), \mathbf{ncols}(\mathbf{B}), \mathbf{L}(\mathbf{B}) = \{(i, j, B_{ij})\} \rangle$

7282 The argument matrices, the "product" operator (`op`), and the accumulation operator (if provided)
7283 are tested for domain compatibility as follows:

- 7284 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
7285 must be from one of the pre-defined types of Table 3.2.
- 7286 2. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 7287 3. $\mathbf{D}(\mathbf{B})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 7288 4. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 7289 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
7290 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
7291 the accumulation operator.

7292 Two domains are compatible with each other if values from one domain can be cast to values in
7293 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
7294 compatible with each other. A domain from a user-defined type is only compatible with itself. If
7295 any compatibility rule above is violated, execution of `GrB_kronecker` ends and the domain mismatch
7296 error listed above is returned.

7297 From the argument matrices, the internal matrices and mask used in the computation are formed
7298 (\leftarrow denotes copy):

- 7299 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 7300 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 7301 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
7302 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 7303 (b) If `Mask \neq GrB_NULL`,
 - 7304 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
7305 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 7306 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
7307 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 7308 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 7309 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 7310 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

7311 The internal matrices and masks are checked for dimension compatibility. The following conditions
7312 must hold:

- 7313 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 7314 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 7315 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) \cdot \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 7316 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) \cdot \mathbf{ncols}(\tilde{\mathbf{B}})$.

7317 If any compatibility rule above is violated, execution of `GrB_kronecker` ends and the dimension
7318 mismatch error listed above is returned.

7319 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
7320 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7321 We are now ready to carry out the Kronecker product and any additional associated operations.
7322 We describe this in terms of two intermediate matrices:

- 7323 • $\tilde{\mathbf{T}}$: The matrix holding the Kronecker product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 7324 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

7325 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$
7326 $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$ is created. The value of
7327 each of its elements is computed by

$$7328 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7329 where \otimes is the multiplicative operator specified by the `op` parameter.

7330 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7331 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 7332 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$7333 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

7334 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
7335 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$7336 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7337 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7338 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7341 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7342 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix `C`,
7343 using what is called a *standard matrix mask and replace*. This is carried out under control of the
7344 mask which acts as a “write mask”.

- 7345 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in `C` on input to this operation are
7346 deleted and the content of the new output matrix, `C`, is defined as,

$$7347 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7348 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
7349 copied into the result matrix, `C`, and elements of `C` that fall outside the set indicated by the
7350 mask are unchanged:

$$7351 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7352 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
7353 of matrix `C` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
7354 exits with return value `GrB_SUCCESS` and the new content of matrix `C` is as defined above but
7355 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7356 sequence. `s`

7357 **Chapter 5**

7358 **Nonpolymorphic interface**

7359 Each polymorphic GraphBLAS method (those with multiple parameter signatures under the same
 7360 name) has a corresponding set of long-name forms that are specific to each parameter signature.
 7361 That is show in Tables 5.1 through 5.11.

Table 5.1: Long-name, nonpolymorphic form of GraphBLAS methods.

Polymorphic signature	Nonpolymorphic signature
GrB_Monoid_new(GrB_Monoid*,...,bool)	GrB_Monoid_new_BOOL(GrB_Monoid*,GrB_BinaryOp,bool)
GrB_Monoid_new(GrB_Monoid*,...,int8_t)	GrB_Monoid_new_INT8(GrB_Monoid*,GrB_BinaryOp,int8_t)
GrB_Monoid_new(GrB_Monoid*,...,uint8_t)	GrB_Monoid_new_UINT8(GrB_Monoid*,GrB_BinaryOp,uint8_t)
GrB_Monoid_new(GrB_Monoid*,...,int16_t)	GrB_Monoid_new_INT16(GrB_Monoid*,GrB_BinaryOp,int16_t)
GrB_Monoid_new(GrB_Monoid*,...,uint16_t)	GrB_Monoid_new_UINT16(GrB_Monoid*,GrB_BinaryOp,uint16_t)
GrB_Monoid_new(GrB_Monoid*,...,int32_t)	GrB_Monoid_new_INT32(GrB_Monoid*,GrB_BinaryOp,int32_t)
GrB_Monoid_new(GrB_Monoid*,...,uint32_t)	GrB_Monoid_new_UINT32(GrB_Monoid*,GrB_BinaryOp,uint32_t)
GrB_Monoid_new(GrB_Monoid*,...,int64_t)	GrB_Monoid_new_INT64(GrB_Monoid*,GrB_BinaryOp,int64_t)
GrB_Monoid_new(GrB_Monoid*,...,uint64_t)	GrB_Monoid_new_UINT64(GrB_Monoid*,GrB_BinaryOp,uint64_t)
GrB_Monoid_new(GrB_Monoid*,...,float)	GrB_Monoid_new_FP32(GrB_Monoid*,GrB_BinaryOp,float)
GrB_Monoid_new(GrB_Monoid*,...,double)	GrB_Monoid_new_FP64(GrB_Monoid*,GrB_BinaryOp,double)
GrB_Monoid_new(GrB_Monoid*,...,other)	GrB_Monoid_new_UDT(GrB_Monoid*,GrB_BinaryOp,void*)

Table 5.2: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_setElement(..., bool,...)	GrB_Scalar_setElement_BOOL(..., bool,...)
GrB_Scalar_setElement(..., int8_t,...)	GrB_Scalar_setElement_INT8(..., int8_t,...)
GrB_Scalar_setElement(..., uint8_t,...)	GrB_Scalar_setElement_UINT8(..., uint8_t,...)
GrB_Scalar_setElement(..., int16_t,...)	GrB_Scalar_setElement_INT16(..., int16_t,...)
GrB_Scalar_setElement(..., uint16_t,...)	GrB_Scalar_setElement_UINT16(..., uint16_t,...)
GrB_Scalar_setElement(..., int32_t,...)	GrB_Scalar_setElement_INT32(..., int32_t,...)
GrB_Scalar_setElement(..., uint32_t,...)	GrB_Scalar_setElement_UINT32(..., uint32_t,...)
GrB_Scalar_setElement(..., int64_t,...)	GrB_Scalar_setElement_INT64(..., int64_t,...)
GrB_Scalar_setElement(..., uint64_t,...)	GrB_Scalar_setElement_UINT64(..., uint64_t,...)
GrB_Scalar_setElement(..., float,...)	GrB_Scalar_setElement_FP32(..., float,...)
GrB_Scalar_setElement(..., double,...)	GrB_Scalar_setElement_FP64(..., double,...)
GrB_Scalar_setElement(..., <i>other</i> ,...)	GrB_Scalar_setElement_UDT(..., const void*,...)
GrB_Scalar_extractElement(bool*,...)	GrB_Scalar_extractElement_BOOL(bool*,...)
GrB_Scalar_extractElement(int8_t*,...)	GrB_Scalar_extractElement_INT8(int8_t*,...)
GrB_Scalar_extractElement(uint8_t*,...)	GrB_Scalar_extractElement_UINT8(uint8_t*,...)
GrB_Scalar_extractElement(int16_t*,...)	GrB_Scalar_extractElement_INT16(int16_t*,...)
GrB_Scalar_extractElement(uint16_t*,...)	GrB_Scalar_extractElement_UINT16(uint16_t*,...)
GrB_Scalar_extractElement(int32_t*,...)	GrB_Scalar_extractElement_INT32(int32_t*,...)
GrB_Scalar_extractElement(uint32_t*,...)	GrB_Scalar_extractElement_UINT32(uint32_t*,...)
GrB_Scalar_extractElement(int64_t*,...)	GrB_Scalar_extractElement_INT64(int64_t*,...)
GrB_Scalar_extractElement(uint64_t*,...)	GrB_Scalar_extractElement_UINT64(uint64_t*,...)
GrB_Scalar_extractElement(float*,...)	GrB_Scalar_extractElement_FP32(float*,...)
GrB_Scalar_extractElement(double*,...)	GrB_Scalar_extractElement_FP64(double*,...)
GrB_Scalar_extractElement(<i>other</i> *,...)	GrB_Scalar_extractElement_UDT(void*,...)

Table 5.3: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Vector_build(...,const bool*,...)	GrB_Vector_build_BOOL(...,const bool*,...)
GrB_Vector_build(...,const int8_t*,...)	GrB_Vector_build_INT8(...,const int8_t*,...)
GrB_Vector_build(...,const uint8_t*,...)	GrB_Vector_build_UINT8(...,const uint8_t*,...)
GrB_Vector_build(...,const int16_t*,...)	GrB_Vector_build_INT16(...,const int16_t*,...)
GrB_Vector_build(...,const uint16_t*,...)	GrB_Vector_build_UINT16(...,const uint16_t*,...)
GrB_Vector_build(...,const int32_t*,...)	GrB_Vector_build_INT32(...,const int32_t*,...)
GrB_Vector_build(...,const uint32_t*,...)	GrB_Vector_build_UINT32(...,const uint32_t*,...)
GrB_Vector_build(...,const int64_t*,...)	GrB_Vector_build_INT64(...,const int64_t*,...)
GrB_Vector_build(...,const uint64_t*,...)	GrB_Vector_build_UINT64(...,const uint64_t*,...)
GrB_Vector_build(...,const float*,...)	GrB_Vector_build_FP32(...,const float*,...)
GrB_Vector_build(...,const double*,...)	GrB_Vector_build_FP64(...,const double*,...)
GrB_Vector_build(...,const <i>other</i> *,...)	GrB_Vector_build_UDT(...,const void*,...)
GrB_Vector_setElement(...,GrB_Scalar,...)	GrB_Vector_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Vector_setElement(...,bool,...)	GrB_Vector_setElement_BOOL(..., bool,...)
GrB_Vector_setElement(...,int8_t,...)	GrB_Vector_setElement_INT8(..., int8_t,...)
GrB_Vector_setElement(...,uint8_t,...)	GrB_Vector_setElement_UINT8(..., uint8_t,...)
GrB_Vector_setElement(...,int16_t,...)	GrB_Vector_setElement_INT16(..., int16_t,...)
GrB_Vector_setElement(...,uint16_t,...)	GrB_Vector_setElement_UINT16(..., uint16_t,...)
GrB_Vector_setElement(...,int32_t,...)	GrB_Vector_setElement_INT32(..., int32_t,...)
GrB_Vector_setElement(...,uint32_t,...)	GrB_Vector_setElement_UINT32(..., uint32_t,...)
GrB_Vector_setElement(...,int64_t,...)	GrB_Vector_setElement_INT64(..., int64_t,...)
GrB_Vector_setElement(...,uint64_t,...)	GrB_Vector_setElement_UINT64(..., uint64_t,...)
GrB_Vector_setElement(...,float,...)	GrB_Vector_setElement_FP32(..., float,...)
GrB_Vector_setElement(...,double,...)	GrB_Vector_setElement_FP64(..., double,...)
GrB_Vector_setElement(..., <i>other</i> ,...)	GrB_Vector_setElement_UDT(...,const void*,...)
GrB_Vector_extractElement(GrB_Scalar,...)	GrB_Vector_extractElement_Scalar(GrB_Scalar,...)
GrB_Vector_extractElement(bool*,...)	GrB_Vector_extractElement_BOOL(bool*,...)
GrB_Vector_extractElement(int8_t*,...)	GrB_Vector_extractElement_INT8(int8_t*,...)
GrB_Vector_extractElement(uint8_t*,...)	GrB_Vector_extractElement_UINT8(uint8_t*,...)
GrB_Vector_extractElement(int16_t*,...)	GrB_Vector_extractElement_INT16(int16_t*,...)
GrB_Vector_extractElement(uint16_t*,...)	GrB_Vector_extractElement_UINT16(uint16_t*,...)
GrB_Vector_extractElement(int32_t*,...)	GrB_Vector_extractElement_INT32(int32_t*,...)
GrB_Vector_extractElement(uint32_t*,...)	GrB_Vector_extractElement_UINT32(uint32_t*,...)
GrB_Vector_extractElement(int64_t*,...)	GrB_Vector_extractElement_INT64(int64_t*,...)
GrB_Vector_extractElement(uint64_t*,...)	GrB_Vector_extractElement_UINT64(uint64_t*,...)
GrB_Vector_extractElement(float*,...)	GrB_Vector_extractElement_FP32(float*,...)
GrB_Vector_extractElement(double*,...)	GrB_Vector_extractElement_FP64(double*,...)
GrB_Vector_extractElement(<i>other</i> *,...)	GrB_Vector_extractElement_UDT(void*,...)
GrB_Vector_extractTuples(...,bool*,...)	GrB_Vector_extractTuples_BOOL(..., bool*,...)
GrB_Vector_extractTuples(...,int8_t*,...)	GrB_Vector_extractTuples_INT8(..., int8_t*,...)
GrB_Vector_extractTuples(...,uint8_t*,...)	GrB_Vector_extractTuples_UINT8(..., uint8_t*,...)
GrB_Vector_extractTuples(...,int16_t*,...)	GrB_Vector_extractTuples_INT16(..., int16_t*,...)
GrB_Vector_extractTuples(...,uint16_t*,...)	GrB_Vector_extractTuples_UINT16(..., uint16_t*,...)
GrB_Vector_extractTuples(...,int32_t*,...)	GrB_Vector_extractTuples_INT32(..., int32_t*,...)
GrB_Vector_extractTuples(...,uint32_t*,...)	GrB_Vector_extractTuples_UINT32(..., uint32_t*,...)
GrB_Vector_extractTuples(...,int64_t*,...)	GrB_Vector_extractTuples_INT64(..., int64_t*,...)
GrB_Vector_extractTuples(...,uint64_t*,...)	GrB_Vector_extractTuples_UINT64(..., uint64_t*,...)
GrB_Vector_extractTuples(...,float*,...)	GrB_Vector_extractTuples_FP32(..., float*,...)
GrB_Vector_extractTuples(...,double*,...)	GrB_Vector_extractTuples_FP64(..., double*,...)
GrB_Vector_extractTuples(..., <i>other</i> *,...)	GrB_Vector_extractTuples_UDT(..., void*,...)

Table 5.4: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_build(...,const bool*,...)	GrB_Matrix_build_BOOL(...,const bool*,...)
GrB_Matrix_build(...,const int8_t*,...)	GrB_Matrix_build_INT8(...,const int8_t*,...)
GrB_Matrix_build(...,const uint8_t*,...)	GrB_Matrix_build_UINT8(...,const uint8_t*,...)
GrB_Matrix_build(...,const int16_t*,...)	GrB_Matrix_build_INT16(...,const int16_t*,...)
GrB_Matrix_build(...,const uint16_t*,...)	GrB_Matrix_build_UINT16(...,const uint16_t*,...)
GrB_Matrix_build(...,const int32_t*,...)	GrB_Matrix_build_INT32(...,const int32_t*,...)
GrB_Matrix_build(...,const uint32_t*,...)	GrB_Matrix_build_UINT32(...,const uint32_t*,...)
GrB_Matrix_build(...,const int64_t*,...)	GrB_Matrix_build_INT64(...,const int64_t*,...)
GrB_Matrix_build(...,const uint64_t*,...)	GrB_Matrix_build_UINT64(...,const uint64_t*,...)
GrB_Matrix_build(...,const float*,...)	GrB_Matrix_build_FP32(...,const float*,...)
GrB_Matrix_build(...,const double*,...)	GrB_Matrix_build_FP64(...,const double*,...)
GrB_Matrix_build(...,const <i>other</i> *,...)	GrB_Matrix_build_UDT(...,const void*,...)
GrB_Matrix_setElement(...,GrB_Scalar,...)	GrB_Matrix_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Matrix_setElement(...,bool,...)	GrB_Matrix_setElement_BOOL(..., bool,...)
GrB_Matrix_setElement(...,int8_t,...)	GrB_Matrix_setElement_INT8(..., int8_t,...)
GrB_Matrix_setElement(...,uint8_t,...)	GrB_Matrix_setElement_UINT8(..., uint8_t,...)
GrB_Matrix_setElement(...,int16_t,...)	GrB_Matrix_setElement_INT16(..., int16_t,...)
GrB_Matrix_setElement(...,uint16_t,...)	GrB_Matrix_setElement_UINT16(..., uint16_t,...)
GrB_Matrix_setElement(...,int32_t,...)	GrB_Matrix_setElement_INT32(..., int32_t,...)
GrB_Matrix_setElement(...,uint32_t,...)	GrB_Matrix_setElement_UINT32(..., uint32_t,...)
GrB_Matrix_setElement(...,int64_t,...)	GrB_Matrix_setElement_INT64(..., int64_t,...)
GrB_Matrix_setElement(...,uint64_t,...)	GrB_Matrix_setElement_UINT64(..., uint64_t,...)
GrB_Matrix_setElement(...,float,...)	GrB_Matrix_setElement_FP32(..., float,...)
GrB_Matrix_setElement(...,double,...)	GrB_Matrix_setElement_FP64(..., double,...)
GrB_Matrix_setElement(..., <i>other</i> ,...)	GrB_Matrix_setElement_UDT(...,const void*,...)
GrB_Matrix_extractElement(GrB_Scalar,...)	GrB_Matrix_extractElement_Scalar(GrB_Scalar,...)
GrB_Matrix_extractElement(bool*,...)	GrB_Matrix_extractElement_BOOL(bool*,...)
GrB_Matrix_extractElement(int8_t*,...)	GrB_Matrix_extractElement_INT8(int8_t*,...)
GrB_Matrix_extractElement(uint8_t*,...)	GrB_Matrix_extractElement_UINT8(uint8_t*,...)
GrB_Matrix_extractElement(int16_t*,...)	GrB_Matrix_extractElement_INT16(int16_t*,...)
GrB_Matrix_extractElement(uint16_t*,...)	GrB_Matrix_extractElement_UINT16(uint16_t*,...)
GrB_Matrix_extractElement(int32_t*,...)	GrB_Matrix_extractElement_INT32(int32_t*,...)
GrB_Matrix_extractElement(uint32_t*,...)	GrB_Matrix_extractElement_UINT32(uint32_t*,...)
GrB_Matrix_extractElement(int64_t*,...)	GrB_Matrix_extractElement_INT64(int64_t*,...)
GrB_Matrix_extractElement(uint64_t*,...)	GrB_Matrix_extractElement_UINT64(uint64_t*,...)
GrB_Matrix_extractElement(float*,...)	GrB_Matrix_extractElement_FP32(float*,...)
GrB_Matrix_extractElement(double*,...)	GrB_Matrix_extractElement_FP64(double*,...)
GrB_Matrix_extractElement(<i>other</i> ,...)	GrB_Matrix_extractElement_UDT(void*,...)
GrB_Matrix_extractTuples(..., bool*,...)	GrB_Matrix_extractTuples_BOOL(..., bool*,...)
GrB_Matrix_extractTuples(..., int8_t*,...)	GrB_Matrix_extractTuples_INT8(..., int8_t*,...)
GrB_Matrix_extractTuples(..., uint8_t*,...)	GrB_Matrix_extractTuples_UINT8(..., uint8_t*,...)
GrB_Matrix_extractTuples(..., int16_t*,...)	GrB_Matrix_extractTuples_INT16(..., int16_t*,...)
GrB_Matrix_extractTuples(..., uint16_t*,...)	GrB_Matrix_extractTuples_UINT16(..., uint16_t*,...)
GrB_Matrix_extractTuples(..., int32_t*,...)	GrB_Matrix_extractTuples_INT32(..., int32_t*,...)
GrB_Matrix_extractTuples(..., uint32_t*,...)	GrB_Matrix_extractTuples_UINT32(..., uint32_t*,...)
GrB_Matrix_extractTuples(..., int64_t*,...)	GrB_Matrix_extractTuples_INT64(..., int64_t*,...)
GrB_Matrix_extractTuples(..., uint64_t*,...)	GrB_Matrix_extractTuples_UINT64(..., uint64_t*,...)
GrB_Matrix_extractTuples(..., float*,...)	GrB_Matrix_extractTuples_FP32(..., float*,...)
GrB_Matrix_extractTuples(..., double*,...)	GrB_Matrix_extractTuples_FP64(..., double*,...)
GrB_Matrix_extractTuples(..., <i>other</i> *,...)	GrB_Matrix_extractTuples_UDT(..., void*,...)

Table 5.5: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_import(...,const bool*,...)	GrB_Matrix_import_BOOL(...,const bool*,...)
GrB_Matrix_import(...,const int8_t*,...)	GrB_Matrix_import_INT8(...,const int8_t*,...)
GrB_Matrix_import(...,const uint8_t*,...)	GrB_Matrix_import_UINT8(...,const uint8_t*,...)
GrB_Matrix_import(...,const int16_t*,...)	GrB_Matrix_import_INT16(...,const int16_t*,...)
GrB_Matrix_import(...,const uint16_t*,...)	GrB_Matrix_import_UINT16(...,const uint16_t*,...)
GrB_Matrix_import(...,const int32_t*,...)	GrB_Matrix_import_INT32(...,const int32_t*,...)
GrB_Matrix_import(...,const uint32_t*,...)	GrB_Matrix_import_UINT32(...,const uint32_t*,...)
GrB_Matrix_import(...,const int64_t*,...)	GrB_Matrix_import_INT64(...,const int64_t*,...)
GrB_Matrix_import(...,const uint64_t*,...)	GrB_Matrix_import_UINT64(...,const uint64_t*,...)
GrB_Matrix_import(...,const float*,...)	GrB_Matrix_import_FP32(...,const float*,...)
GrB_Matrix_import(...,const double*,...)	GrB_Matrix_import_FP64(...,const double*,...)
GrB_Matrix_import(...,const other,...)	GrB_Matrix_import_UDT(...,const void*,...)
GrB_Matrix_export(...,bool*,...)	GrB_Matrix_export_BOOL(...,bool*,...)
GrB_Matrix_export(...,int8_t*,...)	GrB_Matrix_export_INT8(...,int8_t*,...)
GrB_Matrix_export(...,uint8_t*,...)	GrB_Matrix_export_UINT8(...,uint8_t*,...)
GrB_Matrix_export(...,int16_t*,...)	GrB_Matrix_export_INT16(...,int16_t*,...)
GrB_Matrix_export(...,uint16_t*,...)	GrB_Matrix_export_UINT16(...,uint16_t*,...)
GrB_Matrix_export(...,int32_t*,...)	GrB_Matrix_export_INT32(...,int32_t*,...)
GrB_Matrix_export(...,uint32_t*,...)	GrB_Matrix_export_UINT32(...,uint32_t*,...)
GrB_Matrix_export(...,int64_t*,...)	GrB_Matrix_export_INT64(...,int64_t*,...)
GrB_Matrix_export(...,uint64_t*,...)	GrB_Matrix_export_UINT64(...,uint64_t*,...)
GrB_Matrix_export(...,float*,...)	GrB_Matrix_export_FP32(...,float*,...)
GrB_Matrix_export(...,double*,...)	GrB_Matrix_export_FP64(...,double*,...)
GrB_Matrix_export(...,other,...)	GrB_Matrix_export_UDT(...,void*,...)
GrB_free(GrB_Type*)	GrB_Type_free(GrB_Type*)
GrB_free(GrB_UnaryOp*)	GrB_UnaryOp_free(GrB_UnaryOp*)
GrB_free(GrB_IndexUnaryOp*)	GrB_IndexUnaryOp_free(GrB_IndexUnaryOp*)
GrB_free(GrB_BinaryOp*)	GrB_BinaryOp_free(GrB_BinaryOp*)
GrB_free(GrB_Monoid*)	GrB_Monoid_free(GrB_Monoid*)
GrB_free(GrB_Semiring*)	GrB_Semiring_free(GrB_Semiring*)
GrB_free(GrB_Scalar*)	GrB_Scalar_free(GrB_Scalar*)
GrB_free(GrB_Vector*)	GrB_Vector_free(GrB_Vector*)
GrB_free(GrB_Matrix*)	GrB_Matrix_free(GrB_Matrix*)
GrB_free(GrB_Descriptor*)	GrB_Descriptor_free(GrB_Descriptor*)
GrB_wait(GrB_Type, GrB_WaitMode)	GrB_Type_wait(GrB_Type, GrB_WaitMode)
GrB_wait(GrB_UnaryOp, GrB_WaitMode)	GrB_UnaryOp_wait(GrB_UnaryOp, GrB_WaitMode)
GrB_wait(GrB_IndexUnaryOp, GrB_WaitMode)	GrB_IndexUnaryOp_wait(GrB_IndexUnaryOp, GrB_WaitMode)
GrB_wait(GrB_BinaryOp, GrB_WaitMode)	GrB_BinaryOp_wait(GrB_BinaryOp, GrB_WaitMode)
GrB_wait(GrB_Monoid, GrB_WaitMode)	GrB_Monoid_wait(GrB_Monoid, GrB_WaitMode)
GrB_wait(GrB_Semiring, GrB_WaitMode)	GrB_Semiring_wait(GrB_Semiring, GrB_WaitMode)
GrB_wait(GrB_Scalar, GrB_WaitMode)	GrB_Scalar_wait(GrB_Scalar, GrB_WaitMode)
GrB_wait(GrB_Vector, GrB_WaitMode)	GrB_Vector_wait(GrB_Vector, GrB_WaitMode)
GrB_wait(GrB_Matrix, GrB_WaitMode)	GrB_Matrix_wait(GrB_Matrix, GrB_WaitMode)
GrB_wait(GrB_Descriptor, GrB_WaitMode)	GrB_Descriptor_wait(GrB_Descriptor, GrB_WaitMode)
GrB_error(const char**, const GrB_Type)	GrB_Type_error(const char**, const GrB_Type)
GrB_error(const char**, const GrB_UnaryOp)	GrB_UnaryOp_error(const char**, const GrB_UnaryOp)
GrB_error(const char**, const GrB_IndexUnaryOp)	GrB_IndexUnaryOp_error(const char**, const GrB_IndexUnaryOp)
GrB_error(const char**, const GrB_BinaryOp)	GrB_BinaryOp_error(const char**, const GrB_BinaryOp)
GrB_error(const char**, const GrB_Monoid)	GrB_Monoid_error(const char**, const GrB_Monoid)
GrB_error(const char**, const GrB_Semiring)	GrB_Semiring_error(const char**, const GrB_Semiring)
GrB_error(const char**, const GrB_Scalar)	GrB_Scalar_error(const char**, const GrB_Scalar)
GrB_error(const char**, const GrB_Vector)	GrB_Vector_error(const char**, const GrB_Vector)
GrB_error(const char**, const GrB_Matrix)	GrB_Matrix_error(const char**, const GrB_Matrix)
GrB_error(const char**, const GrB_Descriptor)	GrB_Descriptor_error(const char**, const GrB_Descriptor)

Table 5.6: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_eWiseMult(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseMult_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseMult_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseMult_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseMult_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseMult_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseMult_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseAdd_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseAdd_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseAdd_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseAdd_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseAdd_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseAdd_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_extract(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_extract(GrB_Vector,...,GrB_Vector,...)
GrB_extract(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_extract(GrB_Matrix,...,GrB_Matrix,...)
GrB_extract(GrB_Vector,...,GrB_Matrix,...)	GrB_Col_extract(GrB_Vector,...,GrB_Matrix,...)
GrB_assign(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_assign(GrB_Vector,...,GrB_Vector,...)
GrB_assign(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_assign(GrB_Matrix,...,GrB_Matrix,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)	GrB_Col_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)	GrB_Row_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)
GrB_assign(GrB_Vector,...,GrB_Scalar,...)	GrB_Vector_assign_Scalar(GrB_Vector,...,const GrB_Scalar,...)
GrB_assign(GrB_Vector,...,bool,...)	GrB_Vector_assign_BOOL(GrB_Vector,..., bool,...)
GrB_assign(GrB_Vector,...,int8_t,...)	GrB_Vector_assign_INT8(GrB_Vector,..., int8_t,...)
GrB_assign(GrB_Vector,...,uint8_t,...)	GrB_Vector_assign_UINT8(GrB_Vector,..., uint8_t,...)
GrB_assign(GrB_Vector,...,int16_t,...)	GrB_Vector_assign_INT16(GrB_Vector,..., int16_t,...)
GrB_assign(GrB_Vector,...,uint16_t,...)	GrB_Vector_assign_UINT16(GrB_Vector,..., uint16_t,...)
GrB_assign(GrB_Vector,...,int32_t,...)	GrB_Vector_assign_INT32(GrB_Vector,..., int32_t,...)
GrB_assign(GrB_Vector,...,uint32_t,...)	GrB_Vector_assign_UINT32(GrB_Vector,..., uint32_t,...)
GrB_assign(GrB_Vector,...,int64_t,...)	GrB_Vector_assign_INT64(GrB_Vector,..., int64_t,...)
GrB_assign(GrB_Vector,...,uint64_t,...)	GrB_Vector_assign_UINT64(GrB_Vector,..., uint64_t,...)
GrB_assign(GrB_Vector,...,float,...)	GrB_Vector_assign_FP32(GrB_Vector,..., float,...)
GrB_assign(GrB_Vector,...,double,...)	GrB_Vector_assign_FP64(GrB_Vector,..., double,...)
GrB_assign(GrB_Vector,...,other,...)	GrB_Vector_assign_UDT(GrB_Vector,...,const void*,...)
GrB_assign(GrB_Matrix,...,GrB_Scalar,...)	GrB_Matrix_assign_Scalar(GrB_Matrix,...,const GrB_Scalar,...)
GrB_assign(GrB_Matrix,...,bool,...)	GrB_Matrix_assign_BOOL(GrB_Matrix,..., bool,...)
GrB_assign(GrB_Matrix,...,int8_t,...)	GrB_Matrix_assign_INT8(GrB_Matrix,..., int8_t,...)
GrB_assign(GrB_Matrix,...,uint8_t,...)	GrB_Matrix_assign_UINT8(GrB_Matrix,..., uint8_t,...)
GrB_assign(GrB_Matrix,...,int16_t,...)	GrB_Matrix_assign_INT16(GrB_Matrix,..., int16_t,...)
GrB_assign(GrB_Matrix,...,uint16_t,...)	GrB_Matrix_assign_UINT16(GrB_Matrix,..., uint16_t,...)
GrB_assign(GrB_Matrix,...,int32_t,...)	GrB_Matrix_assign_INT32(GrB_Matrix,..., int32_t,...)
GrB_assign(GrB_Matrix,...,uint32_t,...)	GrB_Matrix_assign_UINT32(GrB_Matrix,..., uint32_t,...)
GrB_assign(GrB_Matrix,...,int64_t,...)	GrB_Matrix_assign_INT64(GrB_Matrix,..., int64_t,...)
GrB_assign(GrB_Matrix,...,uint64_t,...)	GrB_Matrix_assign_UINT64(GrB_Matrix,..., uint64_t,...)
GrB_assign(GrB_Matrix,...,float,...)	GrB_Matrix_assign_FP32(GrB_Matrix,..., float,...)
GrB_assign(GrB_Matrix,...,double,...)	GrB_Matrix_assign_FP64(GrB_Matrix,..., double,...)
GrB_assign(GrB_Matrix,...,other,...)	GrB_Matrix_assign_UDT(GrB_Matrix,...,const void*,...)

Table 5.7: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)	GrB_Vector_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)
GrB_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)	GrB_Matrix_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_BOOL(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT8(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT8(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT16(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT16(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT32(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT32(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT64(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT64(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP32(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP64(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp, <i>other</i> ,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UDT(GrB_Vector,...,GrB_BinaryOp,const void*,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_BinaryOp2nd_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_BinaryOp2nd_BOOL(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_BinaryOp2nd_INT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_BinaryOp2nd_INT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_BinaryOp2nd_INT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_BinaryOp2nd_INT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)	GrB_Vector_apply_BinaryOp2nd_FP32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)	GrB_Vector_apply_BinaryOp2nd_FP64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_BinaryOp2nd_UDT(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,const void*,...)

Table 5.8: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_BOOL(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT8(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT8(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT16(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT16(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT32(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT32(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT64(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT64(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP32(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP64(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp, <i>other</i> ,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UDT(GrB_Matrix,...,GrB_BinaryOp,const void*,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_BinaryOp2nd_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_BinaryOp2nd_BOOL(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_BinaryOp2nd_FP32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_BinaryOp2nd_FP64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_BinaryOp2nd_UDT(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,const void*,...)

Table 5.9: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_IndexOp_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_IndexOp_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_IndexOp_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_IndexOp_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_IndexOp_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_IndexOp_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_IndexOp_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_IndexOp_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_IndexOp_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_IndexOp_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)	GrB_Vector_apply_IndexOp_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)	GrB_Vector_apply_IndexOp_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_IndexOp_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_IndexOp_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_IndexOp_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_IndexOp_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_IndexOp_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_IndexOp_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_IndexOp_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_IndexOp_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_IndexOp_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_IndexOp_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_IndexOp_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_IndexOp_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_IndexOp_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_IndexOp_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)

Table 5.10: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)	GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)	GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)	GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)	GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)	GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)	GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)	GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)	GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)	GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)	GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)	GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)	GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)

Table 5.11: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_reduce(GrB_Vector,...,GrB_Monoid,...)	GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Vector,...)	GrB_Vector_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Vector,...)	GrB_Vector_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(bool*,...,GrB_Vector,...)	GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)
GrB_reduce(int8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)
GrB_reduce(uint8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)
GrB_reduce(int16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)
GrB_reduce(uint16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)
GrB_reduce(int32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)
GrB_reduce(uint32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)
GrB_reduce(int64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)
GrB_reduce(uint64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)
GrB_reduce(float*,...,GrB_Vector,...)	GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)
GrB_reduce(double*,...,GrB_Vector,...)	GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)
GrB_reduce(<i>other</i> ,...,GrB_Vector,...)	GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)	GrB_Matrix_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)	GrB_Matrix_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)
GrB_reduce(bool*,...,GrB_Matrix,...)	GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)
GrB_reduce(int8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)
GrB_reduce(uint8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)
GrB_reduce(int16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)
GrB_reduce(uint16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)
GrB_reduce(int32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)
GrB_reduce(uint32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)
GrB_reduce(int64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)
GrB_reduce(uint64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)
GrB_reduce(float*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)
GrB_reduce(double*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)
GrB_reduce(<i>other</i> ,...,GrB_Matrix,...)	GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)
GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)

7362 Appendix A

7363 Revision history

7364 Changes in 2.0.0 (Released: ##-Xxxxx-2021:

- 7365 • Reorganized Chapters 2 and 3: Chapter 2 contains prose regarding the basic concepts cap-
7366 tured in the API; Chapter 3 presents all of the enumerations, literals, data types, and prede-
7367 fined objects required by the API. Made short captions for the List of Tables.
- 7368 • (Issue BB-49, BB-50) Updated and corrected language regarding multithreading and comple-
7369 tion, and requirements regarding acquire-release memory orders. Methods that used to force
7370 complete no longer do.
- 7371 • (Issue BB-74, BB-9) Assigned integer values to all return codes as well as all enumerations
7372 in the API to ensure run-time compatibility between libraries.
- 7373 • (Issues BB-70, BB-67) Changed semantics and signature of `GrB_wait(obj, mode)`. Added wait
7374 modes for 'complete' or 'materialize' and removed `GrB_wait(void)`. **This breaks backward**
7375 **compatibility.**
- 7376 • (Issue GH-51) Removed deprecated `GrB_SCMP` literal from descriptor values. **This breaks**
7377 **backward compatibility.**
- 7378 • (Issues BB-8, BB-36) Added sparse `GrB_Scalar` object and its use in additional variants of
7379 `extract/setElement` methods, and `reduce`, `apply`, `assign` and `select` operations.
- 7380 • (Issues BB-34, GH-33, GH-45) Added new `select` operation that uses an index unary operator.
7381 Added new variants of `apply` that take an index unary operator (matrix and vector variants).
- 7382 • (Issues BB-68, BB-51) Added `serialize` and `deserialize` methods for matrices to/from imple-
7383 mentation defined formats.
- 7384 • (Issues BB-25, GH-42) Added `import` and `export` methods for matrices to/from API specified
7385 formats. Three formats have been specified: `CSC`, `CSR`, `COO`. Dense row and column formats
7386 have been deferred.
- 7387 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.

- 7388 • (Issue BB-73) Allow GrB_NULL for dup operator in matrix and vector build methods. Return
7389 error if duplicate locations encountered.
- 7390 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7391 • (Issue BB-17) Added GrB_ABS_T (absolute value) unary operator.
- 7392 • (Issue GH-46) Adding GrB_ONEB_T binary operator that returns 1 cast to type T (not to
7393 be confused with the proposed unary operator).
- 7394 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added
7395 GrB_NOT_IMPLEMENTED return value (API error) for API any combinations of inputs to
7396 a method that is not supported by the implementation.
- 7397 • Added GrB_EMPTY_OBJECT return value (execution error) that is used when an opaque
7398 object (currently only GrB_Scalar) is passed as an input that cannot be empty.
- 7399 • (Issue BB-45) Removed language about annihilators.
- 7400 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7401 • Updated a number algorithms in the appendix to use new operations and methods.
- 7402 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the
7403 specification.
- 7404 • Typographical error in version macros was corrected. They are all caps: GRB_VERSION and
7405 GRB_SUBVERSION.
- 7406 • Typographical change to eWiseAdd Description to be consistent in order of set intersections.
- 7407 • Typographical errors in eWiseAdd: cut-and-paste errors from eWiseMult/set intersection
7408 fixed to read eWiseAdd/set union.
- 7409 • Typographical error (NEQ → NE) in Description of Table 3.8.
- 7410 Changes in 1.3.0 (Released: 25 September 2019):
- 7411 • (Issue BB-50) Changed definition of completion and added GrB_wait() that takes an opaque
7412 GraphBLAS object as an argument.
- 7413 • (Issue BB-39) Added GrB_kronecker operation.
- 7414 • (Issue BB-40) Added variants of the GrB_apply operation that take a binary function and a
7415 scalar.
- 7416 • (Issue BB-59) Changed specification about how reductions to scalar (GrB_reduce) are to be
7417 performed (to minimize dependence on monoid identity).
- 7418 • (Issue BB-24) Added methods to resize matrices and vectors (GrB_Matrix_resize and GrB_Vector_resize).

7419
7420
7421
7422
7423
7424
7425
7426
7427
7428
7429
7430
7431
7432
7433
7434
7435
7436
7437
7438
7439
7440
7441
7442
7443
7444
7445
7446
7447
7448
7449

- (Issue BB-47) Added methods to remove single elements from matrices and vectors (`GrB_Matrix_removeElement` and `GrB_Vector_removeElement`).
- (Issue BB-41) Added `GrB_STRUCTURE` descriptor flag for masks (consider only the structure of the mask and not the values).
- (Issue BB-64) Deprecated `GrB_SCMP` in favor of new `GrB_COMP` for descriptor values.
- (Issue BB-46) Added predefined descriptors covering all possible combinations of field, value pairs.
- Added unary operators: absolute value (`GrB_ABS_T`) and bitwise complement of integers (`GrB_BNOT_I`).
- (Issues BB-42, BB-62) Added binary operators: Added boolean exclusive-nor (`GrB_LXNOR`) and bitwise logical operators on integers (`GrB_BOR_I`, `GrB_BAND_I`, `GrB_BXOR_I`, `GrB_BXNOR_I`).
- (Issue BB-11) Added a set of predefined monoids and semirings.
- (Issue BB-57) Updated all examples in the appendix to take advantage of new capabilities and predefined objects.
- (Issue BB-43) Added parent-BFS example.
- (Issue BB-1) Fixed bug in the non-batch betweenness centrality algorithm in Appendix C.4 where source nodes were incorrectly assigned path counts.
- (Issue BB-3) Added compile-time preprocessor defines and runtime method for querying the GraphBLAS API version being used.
- (Issue BB-10) Clarified `GrB_init()` and `GrB_finalize()` errors.
- (Issue BB-16) Clarified behavior of boolean and integer division. **Note that `GrB_MINV` for integer and boolean types was removed from this version of the spec.**
- (Issue BB-19) Clarified aliasing in user-defined operators.
- (Issue BB-20) Clarified language about behavior of `GrB_free()` with predefined objects (implementation defined)
- (Issue BB-55) Clarified that multiplication does not have to distribute over addition in a GraphBLAS semiring.
- (Issue BB-45) Removed unnecessary language about annihilators.
- (Issue BB-61) Removed unnecessary language about implied zeros.
- (Issue BB-60) Added disclaimer against overspecification.
- Fixed miscellaneous typographical errors (such as \otimes , \oplus).

7450 Changes in 1.2.0:

7451 • Removed "provisional" clause.

7452 Changes in 1.1.0:

7453 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
7454 `assign` operations.

7455 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.

7456 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.

7457 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
7458 from the domains of the binary operator provided.

7459 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
7460 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
7461 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
7462 which is returned when the capacities of the output arrays are insufficient to hold all of the
7463 tuples.

7464 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
7465 face.

7466 • Added replace flag (z) notation to Table 4.1.

7467 • Updated the "Mathematical Description" of the `assign` operation in Table 4.1.

7468 • Added triangle counting example.

7469 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
7470 sections of GraphBLAS operations when the respective text was the "standard" text (i.e.,
7471 identical in a majority of the operations).

7472 • Fixed typographical errors.

7473 Changes in 1.0.2:

7474 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
7475 matrices and avoid casting issues in certain implementations.

7476 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
7477 erased outside the assigned area.

7478 • Changes non-polymorphic interface:

7479 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

7480 – Renamed `GrB_Vector_reduce_BinaryOp` to `GrB_Matrix_reduce_BinaryOp`.

7481 – Renamed `GrB_Vector_reduce_Monoid` to `GrB_Matrix_reduce_Monoid`.

7482 • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.

7483 • Fixed numerous typographical errors.

7484 **Appendix B**

7485 **Non-opaque data format definitions**

7486 **B.1 GrB_Format: Specify the format for input/output of a Graph-**
7487 **BLAS matrix.**

7488 In this section, the non-opaque matrix formats specified by GrB_Format and used in matrix import
7489 and export methods are defined.

7490 **B.1.1 GrB_CSR_FORMAT**

7491 The GrB_CSR_FORMAT format indicates that a matrix will be imported or exported using the
7492 compressed sparse row (CSR) format. `indptr` is a pointer to an array of GrB_Index of size `nrows+1`
7493 elements, where the `i`'th index will contain the starting index in the `values` and `indices` arrays
7494 corresponding to the `i`'th row of the matrix. `indices` is a pointer to an array of number of stored
7495 elements (each a GrB_Index), where each element contains the corresponding element's column
7496 index within a row of the matrix. `values` is a pointer to an array of number of stored elements (each
7497 the size of the scalar stored in the matrix) containing the corresponding value. The elements of
7498 each row are not required to be sorted by column index.

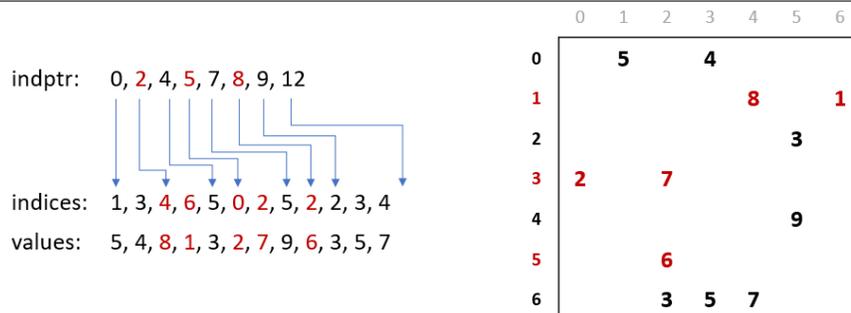


Figure B.1: Data layout for CSR format.

7499 **B.1.2 GrB_CSC_FORMAT**

7500 The GrB_CSC_FORMAT format indicates that a matrix will be imported or exported using the
 7501 compressed sparse column (CSC) format. `indptr` is a pointer to an array of `GrB_Index` of size
 7502 `ncols+1` elements, where the *i*'th index will contain the starting index in the `values` and `indices`
 7503 arrays corresponding to the *i*'th column of the matrix. `indices` is a pointer to an array of number of
 7504 stored elements (each a `GrB_Index`), where each element contains the corresponding element's row
 7505 index within a column of the matrix. `values` is a pointer to an array of number of stored elements
 7506 (each the size of the scalar stored in the matrix) containing the corresponding value. The elements
 7507 of each column are not required to be sorted by row index.

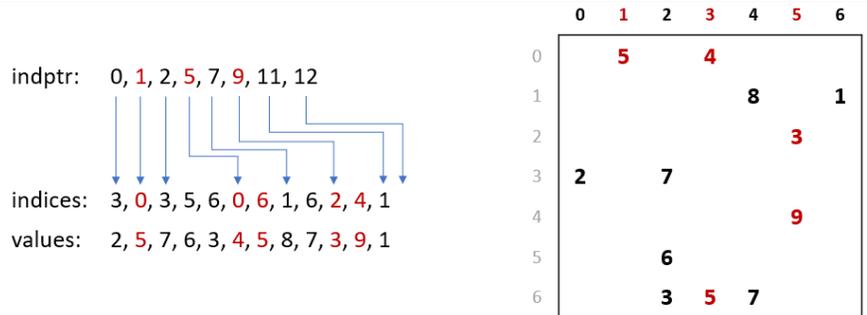


Figure B.2: Data layout for CSC format.

7508 **B.1.3 GrB_COO_FORMAT**

7509 The GrB_COO_FORMAT format indicates that a matrix will be imported or exported using the
 7510 coordinate list (COO) format. `indptr` is a pointer to an array of `GrB_Index` of size number of stored
 7511 elements, where each element contains the corresponding element's column index. `indices` will be a
 7512 pointer to an array of `GrB_Index` of size number of stored elements, where each element contains
 7513 the corresponding element's row index. `values` will be a pointer to an array of size number of stored
 7514 elements (each the size of the scalar stored in the matrix) containing the corresponding value.
 7515 Elements are not required to be sorted in any order.

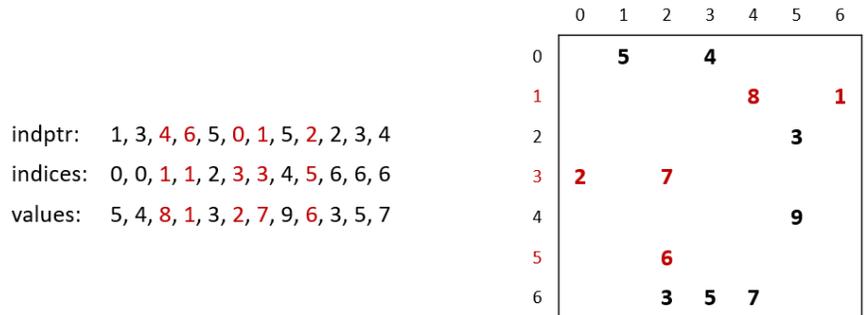


Figure B.3: Data layout for COO format.

7516 **Appendix C**

7517 **Examples**

C.1 Example: Level breadth-first search (BFS) in GraphBLAS

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9  * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10 * If  $i$  is not reachable from  $s$ , then  $v[i] = 0$ . (Vector  $v$  should be empty on input.)
11 */
12 GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
13 {
14     GrB_Index n;
15     GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
16
17     GrB_Vector_new(v,GrB_INT32,n);    // Vector<int32_t>  $v(n)$ 
18
19     GrB_Vector q;                     // vertices visited in each level
20     GrB_Vector_new(&q,GrB_BOOL,n);    // Vector<bool>  $q(n)$ 
21     GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
22
23     /*
24      * BFS traversal and label the vertices.
25      */
26     int32_t d = 0;                    //  $d =$  level in BFS traversal
27     bool succ = false;                //  $\text{succ} == \text{true}$  when some successor found
28     do {
29         ++d;                           // next level (start with 1)
30         GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL); //  $v[q] = d$ 
31         GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
32             q,A,GrB_DESC_RC);          //  $q[!v] = q \parallel A$ ; finds all the
33                                         // unvisited successors from current  $q$ 
34         GrB_reduce(&succ,GrB_NULL,GrB_LOR_MONOID_BOOL,
35             q,GrB_NULL);                //  $\text{succ} = \parallel(q)$ 
36     } while (succ);                    // if there is no successor in  $q$ , we are done.
37
38     GrB_free(&q);                       //  $q$  vector no longer needed
39
40     return GrB_SUCCESS;
41 }

```

C.2 Example: Level BFS in GraphBLAS using apply

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9  * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10 * If  $i$  is not reachable from  $s$ , then  $v[i]$  does not have a stored element.
11 * Vector  $v$  should be uninitialized on input.
12 */
13 GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
14 {
15     GrB_Index n;
16     GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
17
18     GrB_Vector_new(v,GrB_INT32,n);    // Vector<int32_t>  $v(n) = 0$ 
19
20     GrB_Vector q;                    // vertices visited in each level
21     GrB_Vector_new(&q,GrB_BOOL,n);    // Vector<bool>  $q(n) = false$ 
22     GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = true$ , false everywhere else
23
24     /*
25      * BFS traversal and label the vertices.
26      */
27     int32_t level = 0;                // level = depth in BFS traversal
28     GrB_Index nvals;
29     do {
30         ++level;                      // next level (start with 1)
31         GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,
32                 GrB_SECOND_INT32,q,level,GrB_NULL); //  $v[q] = level$ 
33         GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
34                q,A,GrB_DESC_RC);      //  $q[!v] = q \|\&\& A$ ; finds all the
35                                         // unvisited successors from current  $q$ 
36         GrB_Vector_nvals(&nvals, q);
37     } while (nvals);                  // if there is no successor in  $q$ , we are done.
38
39     GrB_free(&q);                      //  $q$  vector no longer needed
40
41     return GrB_SUCCESS;
42 }

```

C.3 Example: Parent BFS in GraphBLAS

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given a binary  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS
9  * traversal of the graph and sets  $parents[i]$  to the index of vertex  $i$ 's parent.
10 * The parent of the root vertex,  $s$ , will be set to itself ( $parents[s] = s$ ). If
11 * vertex  $i$  is not reachable from  $s$ ,  $parents[i]$  will not contain a stored value.
12 */
13 GrB_Info BFS(GrB_Vector *parents, const GrB_Matrix A, GrB_Index s)
14 {
15     GrB_Index N;
16     GrB_Matrix_nrows(&N, A);           //  $N = \#$  vertices
17
18     GrB_Vector_new(parents, GrB_UINT64, N);
19     GrB_Vector_setElement(*parents, s, s); //  $parents[s] = s$ 
20
21     GrB_Vector wavefront;
22     GrB_Vector_new(&wavefront, GrB_UINT64, N);
23     GrB_Vector_setElement(wavefront, 1UL, s); //  $wavefront[s] = 1$ 
24
25     /*
26      * BFS traversal and label the vertices.
27      */
28     GrB_Index nvals;
29     GrB_Vector_nvals(&nvals, wavefront);
30
31     while (nvals > 0)
32     {
33         // convert all stored values in wavefront to their 0-based index
34         GrB_apply(new(&wavefront), GrB_NULL, GrB_NULL, GrB_ROWINDEX_INT64,
35                 wavefront, 0UL, GrB_NULL);
36
37         // "FIRST" because left-multiplying wavefront rows. Masking out the parent
38         // list ensures wavefront values do not overwrite parents already stored.
39         GrB_vxm(wavefront, *parents, GrB_NULL, GrB_MIN_FIRST_SEMIRING_UINT64,
40                wavefront, A, GrB_DESC_RSC);
41
42         // Don't need to mask here since we did it in mxm. Merges new parents in
43         // current wavefront with existing parents:  $parents += wavefront$ 
44         GrB_apply(*parents, GrB_NULL, GrB_PLUS_UINT64,
45                 GrB_IDENTITY_UINT64, wavefront, GrB_NULL);
46
47         GrB_Vector_nvals(&nvals, wavefront);
48     }
49
50     GrB_free(&wavefront);
51
52     return GrB_SUCCESS;
53 }
```

C.4 Example: Betweenness centrality (BC) in GraphBLAS

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ ,
9  * compute the BC-metric vector  $\delta$ , which should be empty on input.
10 */
11 GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n,A); //  $n = \#$  of vertices in graph
15
16     GrB_Vector_new(delta,GrB_FP32,n); // Vector<float>  $\delta(n)$ 
17
18     GrB_Matrix sigma; // Matrix<int32_t>  $\sigma(n,n)$ 
19     GrB_Matrix_new(&sigma,GrB_INT32,n,n); //  $\sigma[d,k] = \#$  shortest paths to node  $k$  at level  $d$ 
20
21     GrB_Vector q; // Vector<int32_t>  $q(n)$  of path counts
22     GrB_Vector_new(&q,GrB_INT32,n); //  $q[s] = 1$ 
23     GrB_Vector_setElement(q,1,s);
24
25     GrB_Vector p; // Vector<int32_t>  $p(n)$  shortest path counts so far
26     GrB_Vector_dup(&p,q); //  $p = q$ 
27
28     GrB_vxm(q,p,GrB_NULL,GrB_PLUS_TIMES_SEMIRING_INT32,
29             q,A,GrB_DESC_RC); // get the first set of out neighbors
30
31     /*
32     * BFS phase
33     */
34     GrB_Index d = 0; // BFS level number
35     int32_t sum = 0; //  $\text{sum} == 0$  when BFS phase is complete
36
37     do {
38         GrB_assign(sigma,GrB_NULL,GrB_NULL,q,d,GrB_ALL,n,GrB_NULL); //  $\sigma[d,:] = q$ 
39         GrB_eWiseAdd(p,GrB_NULL,GrB_NULL,GrB_PLUS_INT32,p,q,GrB_NULL); // accum path counts on this level
40         GrB_vxm(q,p,GrB_NULL,GrB_PLUS_TIMES_SEMIRING_INT32,
41                q,A,GrB_DESC_RC); //  $q = \#$  paths to nodes reachable
42                                     // from current level
43         GrB_reduce(&sum,GrB_NULL,GrB_PLUS_MONOID_INT32,q,GrB_NULL); // sum path counts at this level
44         ++d;
45     } while (sum);
46
47     /*
48     * BC computation phase
49     * ( $t_1, t_2, t_3, t_4$ ) are temporary vectors
50     */
51     GrB_Vector t1; GrB_Vector_new(&t1,GrB_FP32,n);
52     GrB_Vector t2; GrB_Vector_new(&t2,GrB_FP32,n);
53     GrB_Vector t3; GrB_Vector_new(&t3,GrB_FP32,n);
54     GrB_Vector t4; GrB_Vector_new(&t4,GrB_FP32,n);
55
56     for(int i=d-1; i>0; i--)
57     {
58         GrB_assign(t1,GrB_NULL,GrB_NULL,1.0f,GrB_ALL,n,GrB_NULL); //  $t_1 = 1 + \delta$ 
59         GrB_eWiseAdd(t1,GrB_NULL,GrB_NULL,GrB_PLUS_FP32,t1,*delta,GrB_NULL);
60         GrB_extract(t2,GrB_NULL,GrB_NULL,sigma,GrB_ALL,n,i,GrB_DESC_T0); //  $t_2 = \sigma[i,:]$ 
61         GrB_eWiseMult(t2,GrB_NULL,GrB_NULL,GrB_DIV_FP32,t1,t2,GrB_NULL); //  $t_2 = (1 + \delta) / \sigma[i,:]$ 
62         GrB_mxv(t3,GrB_NULL,GrB_NULL,GrB_PLUS_TIMES_SEMIRING_FP32,

```

```

63     A, t2, GrB_NULL);
64     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, GrB_DESC_T0); // t4 = sigma[i-1,:]
65     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, GrB_TIMES_FP32, t4, t3, GrB_NULL); // t4 = sigma[i-1,:]*t3
66     GrB_eWiseAdd(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, *delta, t4, GrB_NULL); // accumulate into delta
67 }
68
69 GrB_free(&sigma);
70 GrB_free(&q); GrB_free(&p);
71 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
72
73 return GrB_SUCCESS;
74 }

```

C.5 Example: Batched BC in GraphBLAS

```

1 #include <stdlib.h>
2 #include "GraphBLAS.h" // in addition to other required C headers
3
4 // Compute partial BC metric for a subset of source vertices, s, in graph A
5 GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
6 {
7     GrB_Index n;
8     GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
9     GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
10
11 // index and value arrays needed to build numsp
12 GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
13 int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
14 for(int i=0; i<nsver; ++i) {
15     i_nsver[i] = i;
16     ones[i] = 1;
17 }
18
19 // numsp: structure holds the number of shortest paths for each node and starting vertex
20 // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
21 GrB_Matrix numsp;
22 GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
23 GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
24 free(i_nsver); free(ones);
25
26 // frontier: Holds the current frontier where values are path counts.
27 // Initialized to out vertices of each source node in s.
28 GrB_Matrix frontier;
29 GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
30 GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, GrB_DESC_RCT0);
31
32 // sigma: stores frontier information for each level of BFS phase. The memory
33 // for an entry in sigmas is only allocated within the do-while loop if needed.
34 // n is an upper bound on diameter.
35 GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n);
36
37 int32_t d = 0; // BFS level number
38 GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
39
40 // ----- The BFS phase (forward sweep) -----
41 do {
42     // sigmas[d](:,s) = dth level frontier from source vertex s
43     GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
44
45     GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
46              GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
47     GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL, GrB_PLUS_INT32,
48                numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
49     GrB_mxnm(frontier, numsp, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
50              A, frontier, GrB_DESC_RCT0); // f<!numsp> = A' +.* f (update frontier)
51     GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level
52     d++;
53 } while (nvals);
54
55 // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
56 GrB_Matrix nspinv;
57 GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
58 GrB_apply(nspinv, GrB_NULL, GrB_NULL,
59           GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
60
61 // bcu: BC updates for each vertex for each starting vertex in s
62 GrB_Matrix bcu;

```

```

63 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
64 GrB_assign(bcu, GrB_NULL, GrB_NULL,
65           1.0f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
66
67 GrB_Matrix w; // temporary workspace matrix
68 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
69
70 // ----- Tally phase (backward sweep) -----
71 for (int i=d-1; i>0; i--) {
72     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
73                 GrB_TIMES_FP32, bcu, nspinv, GrB_DESC_R); // w<sigmas[i]>=(1 ./ nsp).*bcu
74
75     // add contributions by successors and mask with that BFS level's frontier
76     GrB_mxm(w, sigmas[i-1], GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
77            A, w, GrB_DESC_R); // w<sigmas[i-1]> = (A +.* w)
78     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32, GrB_TIMES_FP32,
79                 w, numsp, GrB_NULL); // bcu += w .* numsp
80 }
81
82 // row reduce bcu and subtract "nsver" from every entry to account
83 // for 1 extra value per bcu row element.
84 GrB_reduce(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, bcu, GrB_NULL);
85 GrB_apply(*delta, GrB_NULL, GrB_NULL, GrB_MINUS_FP32, *delta, (float)nsver, GrB_NULL);
86
87 // Release resources
88 for (int i=0; i<d; i++) {
89     GrB_free(&(sigmas[i]));
90 }
91 free(sigmas);
92
93 GrB_free(&frontier); GrB_free(&numsp);
94 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
95
96 return GrB_SUCCESS;
97 }

```

C.6 Example: Maximal independent set (MIS) in GraphBLAS

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 // Assign a random number to each element scaled by the inverse of the node's degree.
8 // This will increase the probability that low degree nodes are selected and larger
9 // sets are selected.
10 void setRandom(void *out, const void *in)
11 {
12     uint32_t degree = *(uint32_t*)in;
13     *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
14 }
15
16 /*
17 * A variant of Luby's randomized algorithm [Luby 1985].
18 *
19 * Given a numeric n x n adjacency matrix A of an unweighted and undirected graph (where
20 * the value true represents an edge), compute a maximal set of independent vertices and
21 * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
22 * of the set (the iset vector should be uninitialized on input.)
23 */
24 GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
25 {
26     GrB_Index n;
27     GrB_Matrix_nrows(&n,A); // n = # of rows of A
28
29     GrB_Vector prob; // holds random probabilities for each node
30     GrB_Vector neighbor_max; // holds value of max neighbor probability
31     GrB_Vector new_members; // holds set of new members to iset
32     GrB_Vector new_neighbors; // holds set of new neighbors to new iset mbrs.
33     GrB_Vector candidates; // candidate members to iset
34
35     GrB_Vector_new(&prob,GrB_FP32,n);
36     GrB_Vector_new(&neighbor_max,GrB_FP32,n);
37     GrB_Vector_new(&new_members,GrB_BOOL,n);
38     GrB_Vector_new(&new_neighbors,GrB_BOOL,n);
39     GrB_Vector_new(&candidates,GrB_BOOL,n);
40     GrB_Vector_new(iset,GrB_BOOL,n); // Initialize independent set vector, bool
41
42     GrB_UnaryOp set_random;
43     GrB_UnaryOp_new(&set_random,setRandom,GrB_FP32,GrB_UINT32);
44
45     // compute the degree of each vertex.
46     GrB_Vector degrees;
47     GrB_Vector_new(&degrees,GrB_FP64,n);
48     GrB_reduce(degrees,GrB_NULL,GrB_NULL,GrB_PLUS_FP64,A,GrB_NULL);
49
50     // Isolated vertices are not candidates: candidates[degrees != 0] = true
51     GrB_assign(candidates,degrees,GrB_NULL,true,GrB_ALL,n,GrB_NULL);
52
53     // add all singletons to iset: iset[degree == 0] = 1
54     GrB_assign(*iset,degrees,GrB_NULL,true,GrB_ALL,n,GrB_DESC_RC);
55
56     // Iterate while there are candidates to check.
57     GrB_Index nvals;
58     GrB_Vector_nvals(&nvals,candidates);
59     while (nvals > 0) {
60         // compute a random probability scaled by inverse of degree
61         GrB_apply(prob,candidates,GrB_NULL,set_random,degrees,GrB_DESC_R);
62

```

```

63 // compute the max probability of all neighbors
64 GrB_m xv(neighbor_max , candidates , GrB_NULL, GrB_MAX_SECOND_SEMIRING_FP32, A, prob , GrB_DESC_R);
65
66 // select vertex if its probability is larger than all its active neighbors ,
67 // and apply a "masked no-op" to remove stored falses
68 GrB_eWiseAdd( new_members , GrB_NULL, GrB_NULL, GrB_GT_FP64, prob , neighbor_max , GrB_NULL);
69 GrB_apply( new_members , new_members , GrB_NULL, GrB_IDENTITY_BOOL, new_members , GrB_DESC_R);
70
71 // add new members to independent set .
72 GrB_eWiseAdd( *iset , GrB_NULL, GrB_NULL, GrB_LOR, *iset , new_members , GrB_NULL);
73
74 // remove new members from set of candidates  $c = c \& \text{!new}$ 
75 GrB_eWiseMult( candidates , new_members , GrB_NULL,
76               GrB_LAND, candidates , candidates , GrB_DESC_RC);
77
78 GrB_Vector_nvals(&nvals , candidates);
79 if ( nvals == 0 ) { break; } // early exit condition
80
81 // Neighbors of new members can also be removed from candidates
82 GrB_m xv( new_neighbors , candidates , GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL,
83          A, new_members , GrB_NULL);
84 GrB_eWiseMult( candidates , new_neighbors , GrB_NULL, GrB_LAND,
85               candidates , candidates , GrB_DESC_RC);
86
87 GrB_Vector_nvals(&nvals , candidates);
88 }
89
90 GrB_free(&neighbor_max); // free all objects "new'ed"
91 GrB_free(&new_members);
92 GrB_free(&new_neighbors);
93 GrB_free(&prob);
94 GrB_free(&candidates);
95 GrB_free(&set_random);
96 GrB_free(&degrees);
97
98 return GrB_SUCCESS;
99 }

```

C.7 Example: Counting triangles in GraphBLAS

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given an  $n \times n$  boolean adjacency matrix,  $A$ , of an undirected graph, computes
9  * the number of triangles in the graph.
10 */
11 uint64_t triangle_count(GrB_Matrix A)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);           //  $n = \#$  of vertices
15
16     //  $L: N \times N$ , lower-triangular, bool
17     GrB_Matrix L;
18     GrB_Matrix_new(&L, GrB_BOOL, n, n);
19     GrB_select(L, GrB_NULL, GrB_NULL, GrB_TRIL, A, 0UL, GrB_NULL);
20
21     GrB_Matrix C;
22     GrB_Matrix_new(&C, GrB_UINT64, n, n);
23
24     GrB_mxm(C, L, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_UINT64, L, L, GrB_NULL); //  $\langle C \rangle = L + * L$ 
25
26     uint64_t count;
27     GrB_reduce(&count, GrB_NULL, GrB_PLUS_MONOID_UINT64, C, GrB_NULL); // 1-norm of  $C$ 
28
29     GrB_free(&C);
30     GrB_free(&L);
31
32     return count;
33 }
```