

The GraphBLAS C API Specification [†]:

Version 2.0.1

[Scott: THIS IS A DRAFT VERION. Update acks and remove DRAFT before release.]

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

Generated on 2023/05/02 at 11:12:26 EDT

[†]Based on *GraphBLAS Mathematics* by Jeremy Kepner

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

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

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

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

Contents

25		
26	List of Tables	9
27	List of Figures	11
28	Acknowledgments	12
29	1 Introduction	13
30	2 Basic concepts	15
31	2.1 Glossary	15
32	2.1.1 GraphBLAS API basic definitions	15
33	2.1.2 GraphBLAS objects and their structure	16
34	2.1.3 Algebraic structures used in the GraphBLAS	17
35	2.1.4 The execution of an application using the GraphBLAS C API	18
36	2.1.5 GraphBLAS methods: behaviors and error conditions	19
37	2.2 Notation	21
38	2.3 Mathematical foundations	22
39	2.4 GraphBLAS opaque objects	23
40	2.5 Execution model	24
41	2.5.1 Execution modes	25
42	2.5.2 Multi-threaded execution	26
43	2.6 Error model	28
44	3 Objects	31
45	3.1 Enumerations for <code>init()</code> and <code>wait()</code>	31
46	3.2 Indices, index arrays, and scalar arrays	31
47	3.3 Types (domains)	32

48	3.4	Algebraic objects, operators and associated functions	33
49	3.4.1	Operators	34
50	3.4.2	Monoids	39
51	3.4.3	Semirings	39
52	3.5	Collections	43
53	3.5.1	Scalars	43
54	3.5.2	Vectors	43
55	3.5.3	Matrices	44
56	3.5.3.1	External matrix formats	44
57	3.5.4	Masks	44
58	3.6	Descriptors	45
59	3.7	Fields	46
60	3.7.1	Input Types	49
61	3.7.1.1	ENUM Handling	49
62	3.7.1.2	GrB_Scalar Handling	49
63	3.7.1.3	String (char*) Handling	49
64	3.7.1.4	void* Handling	49
65	3.7.2	Hints	49
66	3.7.3	GrB_NAME	50
67	3.8	GrB_Info return values	52
68	4	Methods	55
69	4.1	Context methods	55
70	4.1.1	init: Initialize a GraphBLAS context	55
71	4.1.2	finalize: Finalize a GraphBLAS context	56
72	4.1.3	getVersion: Get the version number of the standard.	57
73	4.2	Object methods	57
74	4.2.1	Get and Set methods	58
75	4.2.1.1	get: Query the value of an object	58
76	4.2.1.2	getPreallocSize: Query the buffer size required for String and Void	
77		properties	59

78	4.2.1.3	set: Set field of an object	59
79	4.2.2	Algebra methods	60
80	4.2.2.1	Type_new: Construct a new GraphBLAS (user-defined) type	60
81	4.2.2.2	UnaryOp_new: Construct a new GraphBLAS unary operator	61
82	4.2.2.3	BinaryOp_new: Construct a new GraphBLAS binary operator . . .	63
83	4.2.2.4	Monoid_new: Construct a new GraphBLAS monoid	64
84	4.2.2.5	Semiring_new: Construct a new GraphBLAS semiring	65
85	4.2.2.6	IndexUnaryOp_new: Construct a new GraphBLAS index unary op-	
86		erator [Scott: NEW CONTENT]	67
87	4.2.3	Scalar methods	69
88	4.2.3.1	Scalar_new: Construct a new scalar	69
89	4.2.3.2	Scalar_dup: Construct a copy of a GraphBLAS scalar	70
90	4.2.3.3	Scalar_clear: Clear/remove a stored value from a scalar	71
91	4.2.3.4	Scalar_nvals: Number of stored elements in a scalar	71
92	4.2.3.5	Scalar_setElement: Set the single element in a scalar	72
93	4.2.3.6	Scalar_extractElement: Extract a single element from a scalar. . . .	74
94	4.2.4	Vector methods	75
95	4.2.4.1	Vector_new: Construct new vector	75
96	4.2.4.2	Vector_dup: Construct a copy of a GraphBLAS vector	76
97	4.2.4.3	Vector_resize: Resize a vector	77
98	4.2.4.4	Vector_clear: Clear a vector	78
99	4.2.4.5	Vector_size: Size of a vector	79
100	4.2.4.6	Vector_nvals: Number of stored elements in a vector	80
101	4.2.4.7	Vector_build: Store elements from tuples into a vector	81
102	4.2.4.8	Vector_setElement: Set a single element in a vector	82
103	4.2.4.9	Vector_removeElement: Remove an element from a vector	84
104	4.2.4.10	Vector_extractElement: Extract a single element from a vector. . . .	85
105	4.2.4.11	Vector_extractTuples: Extract tuples from a vector	87
106	4.2.5	Matrix methods	89
107	4.2.5.1	Matrix_new: Construct new matrix	89
108	4.2.5.2	Matrix_dup: Construct a copy of a GraphBLAS matrix	90

109	4.2.5.3	Matrix_diag: Construct a diagonal GraphBLAS matrix	91
110	4.2.5.4	Matrix_resize: Resize a matrix	92
111	4.2.5.5	Matrix_clear: Clear a matrix	93
112	4.2.5.6	Matrix_nrows: Number of rows in a matrix	94
113	4.2.5.7	Matrix_ncols: Number of columns in a matrix	95
114	4.2.5.8	Matrix_nvals: Number of stored elements in a matrix	96
115	4.2.5.9	Matrix_build: Store elements from tuples into a matrix	97
116	4.2.5.10	Matrix_setElement: Set a single element in matrix	99
117	4.2.5.11	Matrix_removeElement: Remove an element from a matrix	100
118	4.2.5.12	Matrix_extractElement: Extract a single element from a matrix . . .	102
119	4.2.5.13	Matrix_extractTuples: Extract tuples from a matrix	104
120	4.2.5.14	Matrix_exportHint: Provide a hint as to which storage format might	
121		be most efficient for exporting a matrix	106
122	4.2.5.15	Matrix_exportSize: Return the array sizes necessary to export a	
123		GraphBLAS matrix object	107
124	4.2.5.16	Matrix_export: Export a GraphBLAS matrix to a pre-defined format	108
125	4.2.5.17	Matrix_import: Import a matrix into a GraphBLAS object	110
126	4.2.5.18	Matrix_serializeSize: Compute the serialize buffer size	111
127	4.2.5.19	Matrix_serialize: Serialize a GraphBLAS matrix.	112
128	4.2.5.20	Matrix_deserialize: Deserialize a GraphBLAS matrix.	113
129	4.2.6	Descriptor methods	114
130	4.2.6.1	Descriptor_new: Create new descriptor	115
131	4.2.6.2	Descriptor_set: Set content of descriptor	115
132	4.2.7	free: Destroy an object and release its resources	117
133	4.2.8	wait: Return once an object is either <i>complete</i> or <i>materialized</i>	118
134	4.2.9	error: Retrieve an error string	119
135	4.3	GraphBLAS operations	120
136	4.3.1	mxm: Matrix-matrix multiply	124
137	4.3.2	vxm: Vector-matrix multiply	129
138	4.3.3	mxv: Matrix-vector multiply	133
139	4.3.4	eWiseMult: Element-wise multiplication	138

140	4.3.4.1	eWiseMult: Vector variant	138
141	4.3.4.2	eWiseMult: Matrix variant	142
142	4.3.5	eWiseAdd: Element-wise addition	147
143	4.3.5.1	eWiseAdd: Vector variant	148
144	4.3.5.2	eWiseAdd: Matrix variant	152
145	4.3.6	extract: Selecting sub-graphs	158
146	4.3.6.1	extract: Standard vector variant	158
147	4.3.6.2	extract: Standard matrix variant	162
148	4.3.6.3	extract: Column (and row) variant	167
149	4.3.7	assign: Modifying sub-graphs	172
150	4.3.7.1	assign: Standard vector variant	172
151	4.3.7.2	assign: Standard matrix variant	177
152	4.3.7.3	assign: Column variant	183
153	4.3.7.4	assign: Row variant	188
154	4.3.7.5	assign: Constant vector variant[Scott: NEW CONTENT]	194
155	4.3.7.6	assign: Constant matrix variant[Scott: NEW CONTENT]	199
156	4.3.8	apply: Apply a function to the elements of an object	205
157	4.3.8.1	apply: Vector variant	205
158	4.3.8.2	apply: Matrix variant	210
159	4.3.8.3	apply: Vector-BinaryOp variants[Scott: NEW CONTENT]	214
160	4.3.8.4	apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]	220
161	4.3.8.5	apply: Vector index unary operator variant[Scott: NEW CONTENT]	226
162	4.3.8.6	apply: Matrix index unary operator variant[Scott: NEW CONTENT]	231
163	4.3.9	select:	236
164	4.3.9.1	select: Vector variant[Scott: NEW CONTENT]	236
165	4.3.9.2	select: Matrix variant[Scott: NEW CONTENT]	241
166	4.3.10	reduce: Perform a reduction across the elements of an object	247
167	4.3.10.1	reduce: Standard matrix to vector variant	247
168	4.3.10.2	reduce: Vector-scalar variant[Scott: NEW CONTENT]	251
169	4.3.10.3	reduce: Matrix-scalar variant[Scott: NEW CONTENT]	255

170	4.3.11 transpose: Transpose rows and columns of a matrix	258
171	4.3.12 kronecker: Kronecker product of two matrices	262
172	5 Nonpolymorphic interface [Scott: NEW CONTENT]	269
173	A Revision history	283
174	B Non-opaque data format definitions	289
175	B.1 GrB_Format: Specify the format for input/output of a GraphBLAS matrix.	289
176	B.1.1 GrB_CSR_FORMAT	289
177	B.1.2 GrB_CSC_FORMAT	290
178	B.1.3 GrB_COO_FORMAT	290
179	C Examples	291
180	C.1 Example: Level breadth-first search (BFS) in GraphBLAS	292
181	C.2 Example: Level BFS in GraphBLAS using apply	293
182	C.3 Example: Parent BFS in GraphBLAS	294
183	C.4 Example: Betweenness centrality (BC) in GraphBLAS	295
184	C.5 Example: Batched BC in GraphBLAS	297
185	C.6 Example: Maximal independent set (MIS) in GraphBLAS	299
186	C.7 Example: Counting triangles in GraphBLAS	301

List of Tables

187		
188	2.1	Types of GraphBLAS opaque objects. 23
189	2.2	Methods that forced completion prior to GraphBLAS v2.0. 28
190	3.1	Enumeration literals and corresponding values input to various GraphBLAS methods. 32
191	3.2	Predefined GrB_Type values. 33
192	3.3	Operator input for relevant GraphBLAS operations. 34
193	3.4	Properties and recipes for building GraphBLAS algebraic objects. 35
194	3.5	Predefined unary and binary operators for GraphBLAS in C. 37
195	3.6	Predefined index unary operators for GraphBLAS in C. 38
196	3.7	Predefined monoids for GraphBLAS in C. 40
197	3.8	Predefined “true” semirings for GraphBLAS in C. 41
198	3.9	Other useful predefined semirings for GraphBLAS in C. 42
199	3.10	GrB_Format enumeration literals and corresponding values for matrix import and
200		export methods. 44
201	3.11	Descriptor types and literals for fields and values. 47
202	3.12	Predefined GraphBLAS descriptors. 48
203	3.13	Field values of type GrB_Field enumeration, corresponding types, and the objects
204		which must implement that GrB_Field. Collection refers to GrB_Matrix, GrB_Vector,
205		and GrB_Scalar, Algebraic refers to Operators, Monoids, and Semirings, while All refers
206		to all GraphBLAS objects. Global fields are denoted by Global. All fields may be
207		read, some may be written (denoted by W), and some are hints (denoted by H) which
208		may be ignored by the implementation. For * see 3.7 51
209	3.14	Descriptions of select <i>field</i> , <i>value</i> pairs listed in 3.13 52
210	3.15	Field value enumerations. 53
211	3.16	Enumeration literals and corresponding values returned by GraphBLAS methods
212		and operations. 54

213	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in	
214		this specification.	121
215	5.1	Long-name, nonpolymorphic form of GraphBLAS methods.	269
216	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	270
217	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	271
218	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	272
219	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	273
220	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	274
221	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	275
222	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	276
223	5.9	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
224		CONTENT]	277
225	5.10	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
226		CONTENT]	278
227	5.11	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	279
228	5.12	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	280
229	5.13	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	281

List of Figures

231	3.1	Hierarchy of algebraic object classes in GraphBLAS.	43
232	4.1	Flowchart for the GraphBLAS operations.	122
233	B.1	Data layout for CSR format.	289
234	B.2	Data layout for CSC format.	290
235	B.3	Data layout for COO format.	290

Acknowledgments

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

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

- Benjamin Brock (UC Berkeley)
- Aydin Buluç (Lawrence Berkeley National Laboratory)
- Timothy G. Mattson (Intel Corporation)
- Scott McMillan (Software Engineering Institute at Carnegie Mellon University)
- José Moreira (IBM Corporation)

Those who served as C API Subcommittee members for GraphBLAS 1.0 through 1.3 are (in alphabetical order):

- Aydin Buluç (Lawrence Berkeley National Laboratory)
- Timothy G. Mattson (Intel Corporation)
- Scott McMillan (Software Engineering Institute at Carnegie Mellon University)
- José Moreira (IBM Corporation)
- Carl Yang (UC Davis)

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

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

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

is not possible to express the signature of that method.

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

The remainder of this document is organized as follows:

- Chapter 2: Basic Concepts
- Chapter 3: Objects
- Chapter 4: Methods
- Chapter 5: Nonpolymorphic interface
- Appendix A: Revision history
- Appendix B: Non-opaque data format definitions
- Appendix C: Examples

Chapter 2

Basic concepts

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

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

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

2.1 Glossary

2.1.1 GraphBLAS API basic definitions

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

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

2.1.2 GraphBLAS objects and their structure

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

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

- *mask*: An internal GraphBLAS object used to control how values are stored in a method's output object. The mask exists only inside a method; hence, it is called an *internal opaque object*. A mask is formed from the elements of a collection object (vector or matrix) input as a mask parameter to a method. GraphBLAS allows two types of masks:
 1. In the default case, an element of the mask exists for each element that exists in the input collection object when the value of that element, when cast to a Boolean type, evaluates to `true`.
 2. In the *structure only* case, masks have structure but no values. The input collection describes a structure whereby an element of the mask exists for each element stored in the input collection regardless of its value.
- *complement*: The *complement* of a GraphBLAS mask, M , is another mask, M' , where the elements of M' are those elements from M that *do not* exist.

2.1.3 Algebraic structures used in the GraphBLAS

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

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

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

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

- *GraphBLAS operators*: Binary or unary operators that act on elements of GraphBLAS objects. *GraphBLAS operators* are used to express algebraic structures used in the GraphBLAS such as monoids and semirings. They are also used as arguments to several GraphBLAS methods. There are two types of *GraphBLAS operators*: (1) predefined operators found in Table 3.5 and (2) user-defined operators created using `GrB_UnaryOp_new()` or `GrB_BinaryOp_new()` (see Section 4.2.2).
- *monoid*: An algebraic structure consisting of one domain, an associative binary operator, and the identity of that operator. There are two types of GraphBLAS monoids: (1) predefined monoids found in Table 3.7 and (2) user-defined monoids created using `GrB_Monoid_new()` (see Section 4.2.2).
- *semiring*: An algebraic structure consisting of a set of allowed values (the *domain*), a commutative and associative binary operator called addition, a binary operator called multiplication (where multiplication distributes over addition), and identities over addition (0) and multiplication (1). The additive identity is an annihilator over multiplication.
- *GraphBLAS semiring*: is allowed to diverge from the mathematically rigorous definition of a *semiring* since certain combinations of domains, operators, and identity elements are useful in graph algorithms even when they do not strictly match the mathematical definition of a semiring. There are two types of *GraphBLAS semirings*: (1) predefined semirings found in Tables 3.8 and 3.9, and (2) user-defined semirings created using `GrB_Semiring_new()` (see Section 4.2.2).
- *index unary operator*: A variation of the unary operator that operates on elements of GraphBLAS vectors and matrices along with the index values representing their location in the objects. There are predefined index unary operators found in Table 3.6), and user-defined operators created using `GrB_IndexUnaryOp_new` (see Section 4.2.2).

2.1.4 The execution of an application using the GraphBLAS C API

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

- *complete*: A GraphBLAS object is complete when it can be used in a happens-before relationship with a method call that reads the variable on another thread. This concept is used when reasoning about memory orders in multithreaded programs. A GraphBLAS object defined on one thread that is complete can be safely used as an IN or INOUT argument in a method-call on a second thread assuming the method calls are correctly synchronized so the definition on the first thread *happens-before* it is used on the second thread. In blocking-mode, an object is complete after a GraphBLAS method call that writes to that object returns. In nonblocking-mode, an object is complete after a call to the `GrB_wait()` method with the `GrB_COMPLETE` parameter.
- *materialize*: A GraphBLAS object is materialized when it is (1) complete, (2) the computations defined by the sequence that define the object have finished (either fully or stopped at an error) and will not consume any additional computational resources, and (3) any errors associated with that sequence are available to be read according to the GraphBLAS error model. A GraphBLAS object that is never loaded into a non-opaque data structure may potentially never be materialized. This might happen, for example, if the operations associated with the object are fused or otherwise changed by the runtime system that supports the implementation of the GraphBLAS C API. An object can be materialized by a call to the `materialize` mode of the `GrB_wait()` method.
- *context*: An instance of the GraphBLAS C API implementation as seen by an application. An application can have only one context between the start and end of the application. A context begins with the first thread that calls `GrB_init()` and ends with the first thread to call `GrB_finalize()`. It is an error for `GrB_init()` or `GrB_finalize()` to be called more than one time within an application. The context is used to constrain the behavior of an instance of the GraphBLAS C API implementation and support various execution strategies. Currently, the only supported constraints on a context pertain to the mode of program execution.
- *program execution mode*: Defines how a GraphBLAS sequence executes, and is associated with the *context* of a GraphBLAS C API implementation. It is set by an application with its call to `GrB_init()` to one of two possible states. In *blocking mode*, GraphBLAS methods return after the computations complete and any output objects have been materialized. In *nonblocking mode*, a method may return once the arguments are tested as consistent with the method (i.e., there are no API errors), and potentially before any computation has taken place.

2.1.5 GraphBLAS methods: behaviors and error conditions

- *implementation-defined behavior*: Behavior that must be documented by the implementation and is allowed to vary among different compliant implementations.
- *undefined behavior*: Behavior that is not specified by the GraphBLAS C API. A conforming implementation is free to choose results delivered from a method whose behavior is undefined.
- *thread-safe*: Consider a function called from multiple threads with arguments that do not overlap in memory (i.e. the argument lists do not share memory). If the function is *thread-safe*

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

- 486 • *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as param-
487 eters to a GraphBLAS method are dimension (or shape) compatible if they have the correct
488 number of dimensions and sizes for each dimension to satisfy the rules of the mathematical def-
489 inition of the operation associated with the method. If any *dimension compatibility* rule above
490 is violated, execution of the GraphBLAS method ends and the GrB_DIMENSION_MISMATCH
491 error is returned.
- 492 • *domain compatible*: Two domains for which values from one domain can be cast to values in
493 the other domain as per the rules of the C language. In particular, domains from Table 3.2
494 are all compatible with each other, and a domain from a user-defined type is only compatible
495 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS
496 method ends and the GrB_DOMAIN_MISMATCH error is returned.

2.2 Notation

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.
$\mathbf{v}(i)$ 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.
$< type >$	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.

2.3 Mathematical foundations

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

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

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

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

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

The mathematical formalism for graph operations in the language of linear algebra often assumes that we can operate in the field of real numbers. However, the GraphBLAS C binding is designed for implementation on computers, which by necessity have a finite number of bits to represent numbers. Therefore, we require a conforming implementation to use floating point numbers such as those defined by the IEEE-754 standard (both single- and double-precision) wherever real numbers need to be represented. The practical implications of these finite precision numbers is that the result of a sequence of computations may vary from one execution to the next as the grouping of operands (because of associativity) within the operations changes. While techniques are known to reduce 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).

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

2.4 GraphBLAS opaque objects

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

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

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

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

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

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

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

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

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

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

2.5 Execution model

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

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

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

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

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

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

2.5.1 Execution modes

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

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

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

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

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

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

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

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

It is important to note that, processing of nonopaque objects is never deferred in GraphBLAS. That is, methods that consume nonopaque objects (e.g., `GrB_Matrix_build()`, Section 4.2.5.9) and methods that produce nonopaque objects (e.g., `GrB_Matrix_extractTuples()`, Section 4.2.5.13) always finish consuming or producing those nonopaque objects before returning regardless of the execution mode.

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

2.5.2 Multi-threaded execution

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

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

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

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

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

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

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

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

Completion is the central concept that allows GraphBLAS objects to be used in happens-before relations between threads. In earlier versions of GraphBLAS (1.X) completion was implied by any operation that produced non-opaque values from a GraphBLAS object. These operations are summarized in Table 2.2). In GraphBLAS 2.0, these methods no longer imply completion. This change was made since there are cases where the non-opaque value is needed but the object from which it is computed is not. We want implementations of the GraphBLAS to be able to exploit 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	4.2.4.6
GrB_Vector_extractElement	4.2.4.10
GrB_Vector_extractTuples	4.2.4.11
GrB_Matrix_nvals	4.2.5.8
GrB_Matrix_extractElement	4.2.5.12
GrB_Matrix_extractTuples	4.2.5.13
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 3.16. 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 3.16(a) are non-negative and include `GrB_SUCCESS` (a value of 0) and `GrB_NO_VALUE`.

An API error (listed in Table 3.16(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 3.16(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

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

746 In nonblocking mode, execution errors can be deferred. A return value of `GrB_SUCCESS` only
747 guarantees that there are no API errors in the method invocation. If an execution error value is
748 returned by a method with output object `obj` in nonblocking mode, it indicates that an error was
749 found during execution of any of the pending operations on `obj`, up to and including the `GrB_wait()`
750 method (Section 4.2.8) call that completes those pending operations. When possible, that return
751 value will provide information concerning the cause of the error.

752 As discussed in Section 4.2.8, a `GrB_wait(obj)` on a specific GraphBLAS object `obj` completes all
753 pending operations on that object. No additional errors on the methods that precede the call to
754 `GrB_wait` and have `obj` as an `OUT` or `INOUT` argument can be reported. From a GraphBLAS
755 perspective, those methods are *complete*. Details on the guaranteed state of objects after a call to
756 `GrB_wait` can be found in Section 4.2.8.

757 After a call to any GraphBLAS method that modifies an opaque object, the program can re-
758 trieve additional error information (beyond the error code returned by the method) though a call
759 to the function `GrB_error()`, passing the method's output object as described in Section 4.2.9.
760 The function returns a pointer to a NULL-terminated string, and the contents of that string are
761 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error
762 string. `GrB_error()` is a thread-safe function, in the sense that multiple threads can call it simul-
763 taneously and each will get its own error string back, referring to the object passed as an input
764 argument.

Chapter 3

Objects

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

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

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

3.2 Indices, index arrays, and scalar arrays

In order to interface with third-party software (i.e., software other than an implementation of the GraphBLAS), operations such as `GrB_Matrix_build` (Section 4.2.5.9) and `GrB_Matrix_extractTuples` (Section 4.2.5.13) must specify how the data should be laid out in non-opaque data structures. To this end we explicitly define the types for indices and the arrays used by these operations.

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

```
typedef uint64_t GrB_Index;
```

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

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

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

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

801 3.3 Types (domains)

802 In GraphBLAS, domains correspond to the valid values for types from the host language (in our
 803 case, the C programming language). GraphBLAS defines a number of operators that take elements
 804 from one or more domains and produce elements of a (possibly) different domain. GraphBLAS
 805 also defines three kinds of collections: matrices, vectors and scalars. For any given collection, the
 806 elements of the collection belong to a *domain*, which is the set of valid values for the elements. For
 807 any variable or object V in GraphBLAS we denote as $\mathbf{D}(V)$ the domain of V , that is, the set of
 808 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	GrB_Type_Code	Suffix	C type	Domain
-	GrB_UDT_CODE=0	UDT	-	-
GrB_BOOL	GrB_BOOL_CODE=1	BOOL	bool	{false, true}
GrB_INT8	GrB_INT8_CODE=2	INT8	int8_t	$\mathbb{Z} \cap [-2^7, 2^7)$
GrB_UINT8	GrB_UINT8_CODE=3	UINT8	uint8_t	$\mathbb{Z} \cap [0, 2^8)$
GrB_INT16	GrB_INT16_CODE=4	INT16	int16_t	$\mathbb{Z} \cap [-2^{15}, 2^{15})$
GrB_UINT16	GrB_UINT16_CODE=5	UINT16	uint16_t	$\mathbb{Z} \cap [0, 2^{16})$
GrB_INT32	GrB_INT32_CODE=6	INT32	int32_t	$\mathbb{Z} \cap [-2^{31}, 2^{31})$
GrB_UINT32	GrB_UINT32_CODE=7	UINT32	uint32_t	$\mathbb{Z} \cap [0, 2^{32})$
GrB_INT64	GrB_INT64_CODE=8	INT64	int64_t	$\mathbb{Z} \cap [-2^{63}, 2^{63})$
GrB_UINT64	GrB_UINT64_CODE=9	UINT64	uint64_t	$\mathbb{Z} \cap [0, 2^{64})$
GrB_FP32	GrB_FP32_CODE=10	FP32	float	IEEE 754 binary32
GrB_FP64	GrB_FP64_CODE=11	FP64	double	IEEE 754 binary64

The domains for elements that can be stored in collections and operated on through GraphBLAS methods are defined by GraphBLAS objects called `GrB_Type`. The predefined types and corresponding domains used in the GraphBLAS C API are shown in Table 3.2. The Boolean type (`bool`) is defined in `stdbool.h`, the integral types (`int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`) are defined in `stdint.h`, and the floating-point types (`float`, `double`) are native to the language and platform and in most cases defined by the IEEE-754 standard. UDT stands for user-defined type and is the type code returned for all objects which use a non-predefined type. Implementations which add new types should start their `GrB_Type_Codes` at 100 to avoid possible conflicts with built-in types which may be added in the future.

3.4 Algebraic objects, operators and associated functions

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

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

Some GraphBLAS operations require a monoid or semiring. A monoid contains an associative

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
kronecker	binary operator monoid semiring
dup argument (build methods)	binary operator
accum argument (various methods)	binary operator

binary operator where the input and output domains are the same. The monoid also includes an identity value of the operator. The semiring consists of a binary operator – referred to as the “times” operator – with up to three different domains (two inputs and one output) and a monoid – referred to as the “plus” operator – that is also commutative. Furthermore, the domain of the monoid must be the same as the output domain of the “times” operator.

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

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

3.4.1 Operators

A GraphBLAS *unary operator* $F_u = \langle D_{out}, D_{in}, f \rangle$ is defined by two domains, D_{out} and D_{in} , and an operation $f : D_{in} \rightarrow D_{out}$. For a given GraphBLAS unary operator $F_u = \langle D_{out}, D_{in}, f \rangle$, we

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.

847 define $\mathbf{D}_{out}(F_u) = D_{out}$, $\mathbf{D}_{in}(F_u) = D_{in}$, and $\mathbf{f}(F_u) = f$.

848 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} ,
849 D_{in_2} , and an operation $\odot : D_{in_1} \times D_{in_2} \rightarrow D_{out}$. For a given GraphBLAS binary operator $F_b =$
850 $\langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$, 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 $\odot(F_b) =$
851 \odot . Note that \odot could be used in place of either \oplus or \otimes in other methods and operations.

852 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
853 domains, D_{out} , D_{in_1} , D_{in_2} , the domain of GraphBLAS indices, and an operation $f_i : D_{in_1} \times I_{U64}^2 \times$
854 $D_{in_2} \rightarrow D_{out}$ (where I_{U64} corresponds to the domain of a `GrB_Index`). For a given GraphBLAS
855 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 $\mathbf{f}(F_i) = f_i$.

856 User-defined operators can be created with calls to `GrB_UnaryOp_new`, `GrB_BinaryOp_new`, and
857 `GrB_IndexUnaryOp_new`, respectively. See Section 4.2.2 for information on these methods. The
858 GraphBLAS C API predefines a number of these operators. These are listed in Tables 3.5 and 3.6.
859 Note that most entries in these tables represent a “family” of predefined operators for a set of
860 different types represented by the T , I , or F in their names. For example, the multiplicative
861 inverse (`GrB_MINV_F`) function is only defined for floating-point types ($F = \text{FP32}$ or FP64). The
862 division (`GrB_DIV_T`) function is defined for all types, but only if $y \neq 0$ for integral and floating
863 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	$I \times I \rightarrow I$	$f(x, y) = x \wedge y$, bitwise XOR
GrB_BinaryOp	GrB_BXNOR_ I	$I \times I \rightarrow I$	$f(x, y) = \overline{x \wedge y}$, 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 identifier	Domains (– is don’t care) A, u i, j s result				Description
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)
		–	I_{U64}	$I_{32/64}$	$I_{32/64}$	$f(u_i, i, 0, s) = (i + s)$
GrB_IndexUnaryOp	GrB_COLINDEX_ $I_{32/64}$	–	I_{U64}	$I_{32/64}$	$I_{32/64}$	$f(A_{ij}, i, j, s) = (j + s)$ 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
		–	I_{U64}	I_{64}	bool	$f(u_i, i, 0, s) = (i \leq s)$
GrB_IndexUnaryOp	GrB_ROWGT	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s) = (i > s)$, rows greater than s
		–	I_{U64}	I_{64}	bool	$f(u_i, i, 0, s) = (i > s)$
GrB_IndexUnaryOp	GrB_VALUEEQ_ T	T	–	T	bool	$f(A_{ij}, i, j, s) = (A_{ij} == s)$, elements equal to value s
		T	–	T	bool	$f(u_i, i, 0, s) = (u_i == s)$
GrB_IndexUnaryOp	GrB_VALUENE_ T	T	–	T	bool	$f(A_{ij}, i, j, s) = (A_{ij} \neq s)$, elements not equal to value s
		T	–	T	bool	$f(u_i, i, 0, s) = (u_i \neq s)$
GrB_IndexUnaryOp	GrB_VALUELT_ T	T	–	T	bool	$f(A_{ij}, i, j, s) = (A_{ij} < s)$, elements less than value s
		T	–	T	bool	$f(u_i, i, 0, s) = (u_i < s)$
GrB_IndexUnaryOp	GrB_VALUELE_ T	T	–	T	bool	$f(A_{ij}, i, j, s) = (A_{ij} \leq s)$, elements less or equal to value s
		T	–	T	bool	$f(u_i, i, 0, s) = (u_i \leq s)$
GrB_IndexUnaryOp	GrB_VALUEGT_ T	T	–	T	bool	$f(A_{ij}, i, j, s) = (A_{ij} > s)$, elements greater than value s
		T	–	T	bool	$f(u_i, i, 0, s) = (u_i > s)$
GrB_IndexUnaryOp	GrB_VALUEGE_ T	T	–	T	bool	$f(A_{ij}, i, j, s) = (A_{ij} \geq s)$, elements greater or equal to value s
		T	–	T	bool	$f(u_i, i, 0, s) = (u_i \geq s)$

3.4.2 Monoids

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

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

User-defined monoids can be created with calls to `GrB_Monoid_new` (see Section 4.2.2). The GraphBLAS C API predefines a number of monoids that are listed in Table 3.7. Predefined monoids are named `GrB_op_MONOID_T`, where *op* is the name of the predefined GraphBLAS operator used as the associative binary operation of the monoid and *T* is the domain (type) of the monoid.

3.4.3 Semirings

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

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, then $S = \langle A, F \rangle = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$ is a semiring.

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

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

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

User-defined semirings can be created with calls to `GrB_Semiring_new` (see Section 4.2.2). A list of predefined true semirings and convenience semirings can be found in Tables 3.8 and 3.9, respectively. Predefined semirings are named `GrB_add_mul_SEMIRING_T`, where *add* is the semiring additive 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 `UINT x` or `INT x` , and can be 32 or 64 in `FP x` .

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

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 $\text{UINT}x$ or $\text{INT}x$, and can be 32 or 64 in $\text{FP}x$.

GraphBLAS identifier	Domains, T ($T \times T \rightarrow T$)	+ identity	Description
<code>GrB_MAX_PLUS_SEMIRING_T</code>	$\text{UINT}x$	0	max-plus semiring
<code>GrB_MIN_TIMES_SEMIRING_T</code>	$\text{INT}x$	$\text{INT}x_MAX$	min-times semiring
	$\text{FP}x$	$INFINITY$	
<code>GrB_MAX_TIMES_SEMIRING_T</code>	$\text{INT}x$	$\text{INT}x_MIN$	max-times semiring
	$\text{FP}x$	$-INFINITY$	
<code>GrB_PLUS_MIN_SEMIRING_T</code>	$\text{INT}x$	0	plus-min semiring
	$\text{FP}x$	0	
<code>GrB_MIN_FIRST_SEMIRING_T</code>	$\text{UINT}x$	$\text{UINT}x_MAX$	min-select first semiring
	$\text{INT}x$	$\text{INT}x_MAX$	
	$\text{FP}x$	$INFINITY$	
<code>GrB_MIN_SECOND_SEMIRING_T</code>	$\text{UINT}x$	$\text{UINT}x_MAX$	min-select second semiring
	$\text{INT}x$	$\text{INT}x_MAX$	
	$\text{FP}x$	$INFINITY$	
<code>GrB_MAX_FIRST_SEMIRING_T</code>	$\text{UINT}x$	0	max-select first semiring
	$\text{INT}x$	$\text{INT}x_MIN$	
	$\text{FP}x$	$-INFINITY$	
<code>GrB_MAX_SECOND_SEMIRING_T</code>	$\text{UINT}x$	0	max-select second semiring
	$\text{INT}x$	$\text{INT}x_MIN$	
	$\text{FP}x$	$-INFINITY$	

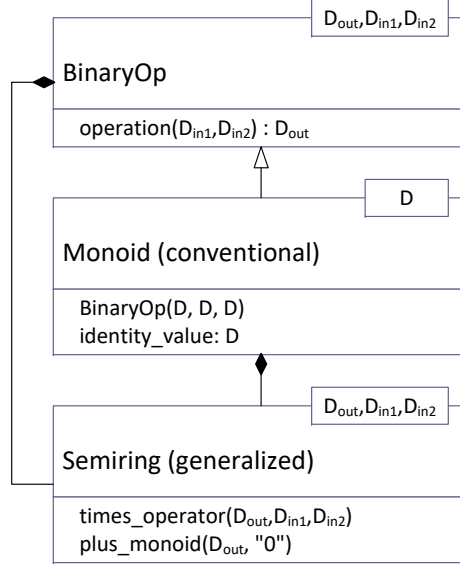


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

3.5 Collections

3.5.1 Scalars

A *GraphBLAS scalar*, $s = \langle D, \{\sigma\} \rangle$, is defined by a domain D , and a set of zero or one *scalar value*, 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 *contents* of the GraphBLAS scalar s . We also define $\mathbf{D}(s) = D$. Finally, $\mathbf{val}(s)$ is a reference to the scalar value, σ , if the GraphBLAS scalar is not empty, and is undefined otherwise.

3.5.2 Vectors

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) 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 $\mathbf{size}(\mathbf{v}) = N$ 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 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} , $\mathbf{v}(i)$ is a reference to v_i if $(i, v_i) \in \mathbf{L}(\mathbf{v})$ and is undefined otherwise.

3.5.3 Matrices

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 number of columns $N > 0$, and a set of tuples (i, j, A_{ij}) where $0 \leq i < M$, $0 \leq j < N$, and $A_{ij} \in D$. A particular pair of values i, j can appear at most once in \mathbf{A} . We define $\mathbf{ncols}(\mathbf{A}) = N$, $\mathbf{nrows}(\mathbf{A}) = M$, 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 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 are the sets of nonempty rows and columns of \mathbf{A} , respectively.) The *structure* of matrix \mathbf{A} is the 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 A_{ij} if $(i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})$ and is undefined otherwise.

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

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

3.5.3.1 External matrix formats

The specification also supports the export and import of matrices to/from a number of commonly used formats, such as COO, CSR, and CSC formats. When importing or exporting a matrix to or from a GraphBLAS object using `GrB_Matrix_import` (§ 4.2.5.17) or `GrB_Matrix_export` (§ 4.2.5.16), it is necessary to specify the data format for the matrix data external to GraphBLAS, which is being imported from or exported to. This non-opaque data format is specified using an argument of enumeration type `GrB_Format` that is used to indicate one of a number of predefined formats. The predefined values of `GrB_Format` are specified in Table 3.10. A precise definition of the non-opaque 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.

3.5.4 Masks

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

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

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

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

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

A two-dimensional mask $\mathbf{M} = \langle M, N, \{(i, j)\} \rangle$ is defined by its number of rows $M > 0$, its number 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 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 also define the sets $\mathbf{indrow}(\mathbf{M}) = \{i : \exists (i, j) \in \mathbf{ind}(\mathbf{M})\}$ and $\mathbf{indcol}(\mathbf{M}) = \{j : \exists (i, j) \in \mathbf{ind}(\mathbf{M})\}$. These are the sets of nonempty rows and columns of \mathbf{M} , respectively. The set $\mathbf{ind}(\mathbf{M})$ is called the *structure* of mask \mathbf{M} .

One common operation on masks is the *complement*. For a one-dimensional mask \mathbf{m} this is denoted as $\neg \mathbf{m}$. For a two-dimensional mask \mathbf{M} , this is denoted as $\neg \mathbf{M}$. The complement of a one-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 possible indices that do not appear in \mathbf{m} . The complement of a two-dimensional mask \mathbf{M} is defined 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 indices that do not appear in \mathbf{M} .

3.6 Descriptors

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

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

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

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

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

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

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

3.7 Fields

All GraphBLAS objects and implementations contain fields like those in the descriptor, which provide information to users and allow setting runtime parameters and hints. All GraphBLAS objects are required to implement the `GrB_get`, `GrB_getPreallocSize`, and `GrB_set` methods required to query and set these fields. The library itself also contains several *(field, value)* pairs, which provide defaults to object level fields, and implementation information such as the version number or implementation name.

The *value, field* pairs available for each object are defined in 3.13, although implementations may add `GrB_Field` enum values to extend the behavior of objects and methods. A field must always be readable, but in many cases may not be writable. Such read-only fields might contain static, compile-time information such as `GrB_API_VER`, while others are determined by other operations, such as `GrB_BLOCKING_MODE` which is determined by `GrB_Init`.

`GrB_INVALID_VALUE` must be returned when attempting to write to fields which are read only.

The `GrB_Field` enumeration is defined by the values in Table 3.13, and selected values are described in Table 3.14.

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
GrB_Descriptor	Type of a GraphBLAS descriptor object.
GrB_Desc_Field	The descriptor field enumeration.
GrB_Desc_Value	The descriptor value enumeration.

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

Field Name	Value	Description
GrB_OUTP	0	Field name for the output GraphBLAS object.
GrB_MASK	1	Field name for the mask GraphBLAS object.
GrB_INP0	2	Field name for the first input GraphBLAS object.
GrB_INP1	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
GrB_REPLACE	1	Clear the output object before assigning computed values.
GrB_COMP	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.
GrB_TRAN	3	Use the transpose of the associated object.
GrB_STRUCTURE	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

1014 3.7.1 Input Types

1015 Allowable types used in `GrB_get` and `GrB_set` are `ENUM`, `GrB_Scalar`, `char*`, and `void*`. Each
1016 `GrB_Field` is associated with exactly one of these types as defined in Table 3.13. Implementations
1017 that add additional `GrB_Fields` must document the type associated with each `GrB_Field`.

1018 3.7.1.1 ENUM Handling

1019 `ENUM` types use standard `INT32` enumerations defined in C. User code should use the enum name
1020 rather than the integer value directly when getting or setting a field value.

1021 3.7.1.2 GrB_Scalar Handling

1022 When calling `GrB_get`, the user must provide an already initialized `GrB_Scalar` object to which
1023 the implementation will write a value of the correct element type. When calling `GrB_set`, the
1024 `GrB_Scalar` must not be empty, otherwise a `GrB_EMPTY_OBJECT` error is raised.

1025 3.7.1.3 String (char*) Handling

1026 When the input to `GrB_set` is a `char*` the input array is null terminated. The GraphBLAS imple-
1027 mentation must copy this array into internal data structures. Prior to calling `GrB_get` for strings,
1028 `GrB_getPreallocSize` must be called with the same arguments, replacing the final `char*` with `int*`
1029 to retrieve the required string buffer size. The user creates a `char` buffer of this size and pass the
1030 pointer to `GrB_get`. The GraphBLAS implementation will write to this buffer, including a trailing
1031 null terminator. The preallocated size returned will include one extra byte for the null terminator.

1032 3.7.1.4 void* Handling

1033 When the input to `GrB_set` is a `void*`, an extra `int` argument is passed to indicate the size of the
1034 buffer. The GraphBLAS implementation must copy this many bytes from the buffer into internal
1035 data structures. Similar to reading strings, prior to calling `GrB_get` for `void*`, `GrB_getPreallocSize`
1036 must be called to find the required buffer size. The user must create a buffer and pass the pointer
1037 to `GrB_get`. The implementation will write to this buffer. No standard specification or protocol is
1038 required for the contents of `void*`. It is meant to be a mechanism to allow full freedom for GraphBLAS
1039 implementations with needs that cannot be handled using `ENUM`, `GrB_Scalar`, or `Strings`.

1040 3.7.2 Hints

1041 Several fields are *hints* (marked H in Table 3.13). A GraphBLAS implementation is free to ignore
1042 a hint and return `GrB_SUCCESS`. When `GrB_get` is called, the provided hint should be returned
1043 by the implementation, even if it chooses to ignore the hint.

1044 3.7.3 GrB_NAME

1045 The GrB_NAME field is a special case regarding writability. All objects which have a GrB_NAME
1046 field default to an empty string. Collections and GrB_Descriptors may have their GrB_NAME set
1047 at any time. User-defined algebraic objects and GrB_Types may only have their GrB_NAME set
1048 once to a globally unique value. Attempting to set this field after it has already been set will return
1049 a GrB_ALREADY_SET error code.

1050 Built-in algebraic objects and GrB_Types have names which can be read, but not written to. The
1051 name returned will be the string form of the GrB_Type listed in Table 3.2 or the GraphBLAS
1052 identifier listed in Tables 3.5, 3.6, 3.7, 3.8, and 3.9. For example, the name of GrB_INT32 type is
1053 "GrB_INT32" and the name of GrB_MIN_FP64 binary op is "GrB_MIN_FP64".

Table 3.13: Field values of type GrB_Field enumeration, corresponding types, and the objects which must implement that GrB_Field. Collection refers to GrB_Matrix, GrB_Vector, and GrB_Scalar, Algebraic refers to Operators, Monoids, and Semirings, while All refers to all GraphBLAS objects. Global fields are denoted by Global. All fields may be read, some may be written (denoted by W), and some are hints (denoted by H) which may be ignored by the implementation. For * see 3.7

Field Name	W H	Value	Implementing Objects	Type
GrB_OUTP	W —	0	GrB_Descriptor	ENUM of GrB_Desc_Value
GrB_MASK	W —	1	GrB_Descriptor	ENUM of GrB_Desc_Value
GrB_INP0	W —	2	GrB_Descriptor	ENUM of GrB_Desc_Value
GrB_INP1	W —	3	GrB_Descriptor	ENUM of GrB_Desc_Value
GrB_NAME	*	10	All	Null terminated char*
GrB_LIBRARY_VER_MAJOR	— —	11	Global	GrB_Scalar (INT32)
GrB_LIBRARY_VER_MINOR	— —	12	Global	GrB_Scalar (INT32)
GrB_LIBRARY_VER_PATCH	— —	13	Global	GrB_Scalar (INT32)
GrB_API_VER_MAJOR	— —	14	Global	GrB_Scalar (INT32)
GrB_API_VER_MINOR	— —	15	Global	GrB_Scalar (INT32)
GrB_API_VER_PATCH	— —	16	Global	GrB_Scalar (INT32)
GrB_BLOCKING_MODE	— —	17	Global	ENUM of GrB_Mode
GrB_STORAGE_ORIENTATION_HINT	W H	100	Global, Collection	ENUM of GrB_Orientation
GrB_STORAGE_SPARSITY_HINT	W H	101	Collection	ENUM of GrB_Sparsity
GrB_ELTYPE_CODE	— —	102	Collection	ENUM of GrB_Type_Code
GrB_INPUT1TYPE_CODE	— —	103	Algebraic	ENUM of GrB_Type_Code
GrB_INPUT2TYPE_CODE	— —	104	Algebraic	ENUM of GrB_Type_Code
GrB_OUTPUTTYPE_CODE	— —	105	Algebraic	ENUM of GrB_Type_Code
GrB_ELTYPE_STRING	— —	106	Collection	Null terminated char*
GrB_INPUT1TYPE_STRING	— —	107	Algebraic	Null terminated char*
GrB_INPUT2TYPE_STRING	— —	108	Algebraic	Null terminated char*
GrB_OUTPUTTYPE_STRING	— —	109	Algebraic	Null terminated char*

Table 3.14: Descriptions of select *field*, *value* pairs listed in 3.13

Field Name	Description
GrB_NAME	The name of any GraphBLAS object, or the name of the library implementation.
GrB_BLOCKING_MODE	The blocking mode as set by GrB_init
GrB_STORAGE_ORIENTATION_HINT	Hint to the library that a collection is best stored in a row (lexicographic) or column (colexicographic) major format.
GrB_STORAGE_SPARSITY_HINT	Hint to the library that it should use a storage format appropriate for the expected sparsity of an object.
GrB_ELTYPE_(CODE/STRING)	The element type of a collection.
GrB_INPUT1TYPE_(CODE/STRING)	The type of the first argument to an operator.
GrB_INPUT2TYPE_(CODE/STRING)	The type of the second argument to an operator.
GrB_OUTPUTTYPE_(CODE/STRING)	The type of the output of an operator.

3.8 GrB_Info return values

All GraphBLAS methods return a GrB_Info enumeration value. The three types of return codes (informational, API error, and execution error) and their corresponding values are listed in Table 3.16.

Table 3.15: Enumerations not defined elsewhere in the documents and used when getting or setting fields are defined in the following tables.

(a) Field values of type GrB_Orientation.

Value Name	Value	Description
GrB_ROWMAJOR	0	The majority of iteration over the object will be row-wise.
GrB_COLMAJOR	1	The majority of iteration over the object will be column-wise.

(b) Field values of type GrB_Storage_Sparsity.

Field Name	Value	Description
GrB_DENSE	0	Most or all of the elements will be populated.
GrB_SPARSE	1	A normal amount of sparsity with most rows and columns containing a few values.
GrB_HYPERSPARSE	2	Many rows or columns will contain no values, resulting in extreme sparsity.

Table 3.16: 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 <code>new</code> 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.
GrB_ALREADY_SET	-9	An attempt was made to write to a field which may only be written to once.

(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.

Chapter 4

Methods

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

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

4.1 Context methods

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

4.1.1 `init`: Initialize a GraphBLAS context

Creates and initializes a GraphBLAS C API context.

C Syntax

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

Parameters

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

1076 **Return Values**

1077 `GrB_SUCCESS` operation completed successfully.

1078 `GrB_PANIC` unknown internal error.

1079 `GrB_INVALID_VALUE` invalid mode specified, or method called multiple times.

1080 **Description**

1081 The `init` method creates and initializes a GraphBLAS C API context. The argument to `GrB_init`
1082 defines the mode for the context. The two available modes are:

- 1083 • `GrB_BLOCKING`: In this mode, each method in a sequence returns after its computations have
1084 completed and output arguments are available to subsequent statements in an application.
1085 When executing in `GrB_BLOCKING` mode, the methods execute in program order.
- 1086 • `GrB_NONBLOCKING`: In this mode, methods in a sequence may return after arguments in
1087 the method have been tested for dimension and domain compatibility within the method
1088 but potentially before their computations complete. Output arguments are available to sub-
1089 sequent GraphBLAS methods in an application. When executing in `GrB_NONBLOCKING`
1090 mode, the methods in a sequence may execute in any order that preserves the mathematical
1091 result defined by the sequence.

1092 An application can only create one context per execution instance. An application may only call
1093 `GrB_Init` once. Calling `GrB_Init` more than once results in undefined behavior.

1094 **4.1.2 finalize: Finalize a GraphBLAS context**

1095 Terminates and frees any internal resources created to support the GraphBLAS C API context.

1096 **C Syntax**

1097 `GrB_Info GrB_finalize();`

1098 **Return Values**

1099 `GrB_SUCCESS` operation completed successfully.

1100 `GrB_PANIC` unknown internal error.

1101 **Description**

1102 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS
1103 C API context. `GrB_finalize` may only be called after a context has been initialized by calling
1104 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-
1105 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined
1106 behavior.

1107 **4.1.3 getVersion: Get the version number of the standard.**

1108 Query the library for the version number of the standard that this library implements.

1109 **C Syntax**

```
1110         GrB_Info GrB_getVersion(unsigned int *version,  
1111                                unsigned int *subversion);
```

1112 **Parameters**

1113 version (OUT) On successful return will hold the value of the major version number.

1114 version (OUT) On successful return will hold the value of the subversion number.

1115 **Return Values**

1116 GrB_SUCCESS operation completed successfully.

1117 GrB_PANIC unknown internal error.

1118 **Description**

1119 The `getVersion` method is used to query the major and minor version number of the GraphBLAS
1120 C API specification that the library implements at runtime. To support compile time queries the
1121 following two macros shall also be defined by the library.

```
1122         #define GRB_VERSION      2  
1123         #define GRB_SUBVERSION  0
```

1124 **4.2 Object methods**

1125 This section describes methods that setup and operate on GraphBLAS opaque objects but are not
1126 part of the the GraphBLAS math specification.

1127 4.2.1 Get and Set methods

1128 The methods in this section query and, optionally, set internal fields of GraphBLAS objects.

1129 4.2.1.1 get: Query the value of an object

1130 C Syntax

```
1131 GrB_Info GrB_get(GrB_<OBJ> o, GrB_Field field, <type> value);
```

1132

```
1133 GrB_Info GrB_get(GrB_Field field, <type> value);
```

1134 Parameters

1135 OBJ (IN) An existing, valid GraphBLAS object (collection, operation, type) which is
1136 being queried. In the signature without GrB_<OBJ>, the Global context is used.

1137 field (IN) The field being queried.

1138 value (OUT) A pointer to or GrB_Scalar containing a value whose type is dependent
1139 on field which will be filled with the current value of the field. type may be int*,
1140 GrB_Scalar, char* or void*.

1141 Return Value

1142 GrB_SUCCESS The method completed successfully.

1143 GrB_PANIC unknown internal error.

1144 GrB_OUT_OF_MEMORY not enough memory available for operation.

1145 GrB_UNINITIALIZED_OBJECT the value parameter is GrB_Scalar and has not been initialized by
1146 a call to new.

1147 GrB_INVALID_VALUE invalid value type provided for the field or invalid field.

1148 Description

1149 Queries a field of an existing GraphBLAS object. The type of the argument is uniquely determined
1150 by field. Fields marked as hints in Table 3.13 will return the hint when queried, not the true internal
1151 value. The size of provided char* and void* buffers is found using GrB_getPreallocSize.

1152 4.2.1.2 `getPreallocSize`: Query the buffer size required for String and Void properties

1153 C Syntax

```
1154     GrB_Info GrB_getPreallocSize(GrB_<OBJ> o, GrB_Field field, int* size);
1155
1156     GrB_Info GrB_getPreallocSize(GrB_Field field, int* size);
```

1157 Parameters

1158 OBJ (IN) An existing, valid GraphBLAS object (collection, operation, type) which is
1159 being queried. In the signature without `GrB_<OBJ>`, the Global context is used.

1160 field (IN) The field being queried.

1161 size (OUT) A pointer to an int which will hold the required buffer size.

1162 Return Value

1163 GrB_SUCCESS The method completed successfully.

1164 GrB_PANIC unknown internal error.

1165 GrB_OUT_OF_MEMORY not enough memory available for operation.

1166 GrB_INVALID_VALUE invalid field or value type associated with the field doesn't return
1167 `char*` or `void*`.

1168 Description

1169 Queries the buffer size required to contain a property of an existing GraphBLAS object. This only
1170 applied to fields which return `char*` or `void*`. The user must create the appropriate buffer of the
1171 indicated size and pass a pointer to that allocated buffer to `GrB_get`.

1172 4.2.1.3 `set`: Set field of an object

1173 Set the content for a field for an existing GraphBLAS object.

1174 C Syntax

```
1175     GrB_Info GrB_set(GrB_<OBJ> o, GrB_Field field, <type> value);
1176     GrB_Info GrB_set(GrB_<OBJ> o, GrB_Field field, void* value, int voidSize);
1177
1178     GrB_Info GrB_set(GrB_Field field, <type> value);
1179     GrB_Info GrB_set(GrB_Field field, void* value, int voidSize);
```

1180 Parameters

1181 OBJ (IN) The GraphBLAS object which is having field set. In the sig-
1182 natures without GrB_<OBJ>, the Global context is used.

1183 field (IN) The field being set.

1184 value (IN) A value whose type is dependent on field. type may be a int,
1185 GrB_Scalar, char* or void*.

1186 voidSize (IN) The size of the void* buffer. Note that a size is not needed for
1187 char* because the string is null-terminated.

1188 Return Values

1189 GrB_SUCCESS The method completed successfully.

1190 GrB_PANIC unknown internal error.

1191 GrB_OUT_OF_MEMORY not enough memory available for operation.

1192 GrB_UNINITIALIZED_OBJECT the GrB_Scalar parameter has not been initialized by a call to new.

1193 GrB_INVALID_VALUE invalid value set on the field, invalid field, or field is read-only.

1194 GrB_ALREADY_SET this field has already been set, and may only be set once.

1195 Description

1196 Set a field of OBJ or the Global context to a new value.

1197 4.2.2 Algebra methods

1198 4.2.2.1 Type_new: Construct a new GraphBLAS (user-defined) type

1199 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,
1200 monoids, semirings, vectors and matrices.

1201 C Syntax

```
1202 GrB_Info GrB_Type_new(GrB_Type *utype,  
1203                       size_t    sizeof(ctype));
```

1204 **Parameters**

1205 `utype` (INOUT) On successful return, contains a handle to the newly created user-defined
1206 GraphBLAS type object.

1207 `ctype` (IN) A C type that defines the new GraphBLAS user-defined type.

1208 **Return Values**

1209 `GrB_SUCCESS` operation completed successfully.

1210 `GrB_PANIC` unknown internal error.

1211 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1212 `GrB_NULL_POINTER` `utype` pointer is NULL.

1213 **Description**

1214 Given a C type `ctype`, the `Type_new` method returns in `utype` a handle to a new GraphBLAS type
1215 that is equivalent to the C type. Variables of this `ctype` must be a struct, union, or fixed-size array.
1216 In particular, given two variables, `src` and `dst`, of type `ctype`, the following operation must be a
1217 valid way to copy the contents of `src` to `dst`:

1218 `memcpy(&dst, &src, sizeof(ctype))`

1219 A new, user-defined type `utype` should be destroyed with a call to `GrB_free(utype)` when no longer
1220 needed.

1221 It is not an error to call this method more than once on the same variable; however, the handle to
1222 the previously created object will be overwritten.

1223 **4.2.2.2 UnaryOp_new: Construct a new GraphBLAS unary operator**

1224 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types
1225 (domains).

1226 **C Syntax**

```
1227     GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1228                             void          (*unary_func)(void*, const void*),  
1229                             GrB_Type      d_out,  
1230                             GrB_Type      d_in);
```

1231 Parameters

1232 **unary_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1233 unary operator object.

1234 **unary_func** (IN) a pointer to a user-defined function that takes one input parameter of **d_in**'s
1235 type and returns a value of **d_out**'s type, both passed as **void** pointers. Specifically
1236 the signature of the function is expected to be of the form:

1237 **void func(void *out, const void *in);**
1238

1239 **d_out** (IN) The **GrB_Type** of the return value of the unary operator being created. Should
1240 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1241 BLAS type.

1242 **d_in** (IN) The **GrB_Type** of the input argument of the unary operator being created.
1243 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1244 GraphBLAS type.

1245 Return Values

1246 **GrB_SUCCESS** operation completed successfully.

1247 **GrB_PANIC** unknown internal error.

1248 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1249 **GrB_UNINITIALIZED_OBJECT** any **GrB_Type** parameter (for user-defined types) has not been ini-
1250 tialized by a call to **GrB_Type_new**.

1251 **GrB_NULL_POINTER** **unary_op** or **unary_func** pointers are **NULL**.

1252 Description

1253 The **UnaryOp_new** method creates a new GraphBLAS unary operator

1254 $f_u = \langle \mathbf{D}(\mathbf{d_out}), \mathbf{D}(\mathbf{d_in}), \text{unary_func} \rangle$

1255 and returns a handle to it in **unary_op**.

1256 The implementation of **unary_func** must be such that it works even if the **d_out** and **d_in** arguments
1257 are aliased. In other words, for all invocations of the function:

1258 **unary_func(out, in);**

1259 the value of **out** must be the same as if the following code was executed:

```

1260     D(d_in) *tmp = malloc(sizeof(D(d_in)));
1261     memcpy(tmp,in,sizeof(D(d_in)));
1262     unary_func(out,tmp);
1263     free(tmp);

```

1264 It is not an error to call this method more than once on the same variable; however, the handle to
1265 the previously created object will be overwritten.

1266 4.2.2.3 BinaryOp_new: Construct a new GraphBLAS binary operator

1267 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types
1268 (domains).

1269 C Syntax

```

1270     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,
1271                               void          (*binary_func)(void*,
1272                               const void*,
1273                               const void*),
1274                               GrB_Type      d_out,
1275                               GrB_Type      d_in1,
1276                               GrB_Type      d_in2);

```

1277 Parameters

1278 **binary_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1279 binary operator object.

1280 **binary_func** (IN) A pointer to a user-defined function that takes two input parameters of types
1281 **d_in1** and **d_in2** and returns a value of type **d_out**, all passed as void pointers.
1282 Specifically the signature of the function is expected to be of the form:

```

1283     void func(void *out, const void *in1, const void *in2);
1284

```

1285 **d_out** (IN) The **GrB_Type** of the return value of the binary operator being created. Should
1286 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1287 BLAS type.

1288 **d_in1** (IN) The **GrB_Type** of the left hand argument of the binary operator being created.
1289 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1290 GraphBLAS type.

1291 **d_in2** (IN) The **GrB_Type** of the right hand argument of the binary operator being cre-
1292 ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-
1293 defined GraphBLAS type.

1294 **Return Values**

1295 `GrB_SUCCESS` operation completed successfully.

1296 `GrB_PANIC` unknown internal error.

1297 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1298 `GrB_UNINITIALIZED_OBJECT` the `GrB_Type` (for user-defined types) has not been initialized by a
1299 call to `GrB_Type_new`.

1300 `GrB_NULL_POINTER` `binary_op` or `binary_func` pointer is `NULL`.

1301 **Description**

1302 The `BinaryOp_new` method creates a new GraphBLAS binary operator

1303
$$f_b = \langle \mathbf{D}(d_out), \mathbf{D}(d_in1), \mathbf{D}(d_in2), \text{binary_func} \rangle$$

1304 and returns a handle to it in `binary_op`.

1305 The implementation of `binary_func` must be such that it works even if any of the `d_out`, `d_in1`, and
1306 `d_in2` arguments are aliased to each other. In other words, for all invocations of the function:

1307 `binary_func(out, in1, in2);`

1308 the value of `out` must be the same as if the following code was executed:

```
1309 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));  
1310 D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));  
1311 memcpy(tmp1, in1, sizeof(D(d_in1)));  
1312 memcpy(tmp2, in2, sizeof(D(d_in2)));  
1313 binary_func(out, tmp1, tmp2);  
1314 free(tmp2);  
1315 free(tmp1);
```

1316 It is not an error to call this method more than once on the same variable; however, the handle to
1317 the previously created object will be overwritten.

1318 **4.2.2.4 Monoid_new: Construct a new GraphBLAS monoid**

1319 Creates a new monoid with specified binary operator and identity value.

1320 C Syntax

```
1321         GrB_Info GrB_Monoid_new(GrB_Monoid    *monoid,  
1322                                 GrB_BinaryOp    binary_op,  
1323                                 <type>          identity);
```

1324 Parameters

1325 **monoid** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1326 monoid object.

1327 **binary_op** (IN) An existing GraphBLAS associative binary operator whose input and output
1328 types are the same.

1329 **identity** (IN) The value of the identity element of the monoid. Must be the same type as
1330 the type used by the **binary_op** operator.

1331 Return Values

1332 **GrB_SUCCESS** operation completed successfully.

1333 **GrB_PANIC** unknown internal error.

1334 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1335 **GrB_UNINITIALIZED_OBJECT** the **GrB_BinaryOp** (for user-defined operators) has not been initial-
1336 ized by a call to **GrB_BinaryOp_new**.

1337 **GrB_NULL_POINTER** monoid pointer is NULL.

1338 **GrB_DOMAIN_MISMATCH** all three argument types of the binary operator and the type of the
1339 identity value are not the same.

1340 Description

1341 The **Monoid_new** method creates a new monoid $M = \langle \mathbf{D}(\text{binary_op}), \text{binary_op}, \text{identity} \rangle$ and re-
1342 turns a handle to it in **monoid**.

1343 If **binary_op** is not associative, the results of GraphBLAS operations that require associativity of
1344 this monoid will be undefined.

1345 It is not an error to call this method more than once on the same variable; however, the handle to
1346 the previously created object will be overwritten.

1347 4.2.2.5 Semiring_new: Construct a new GraphBLAS semiring

1348 Creates a new semiring with specified domain, operators, and elements.

1349 C Syntax

```
1350      GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1351                               GrB_Monoid    add_op,  
1352                               GrB_BinaryOp   mul_op);
```

1353 Parameters

1354 **semiring** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1355 semiring.

1356 **add_op** (IN) An existing GraphBLAS commutative monoid that specifies the addition op-
1357 erator and its identity.

1358 **mul_op** (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-
1359 plication operator. In addition, **mul_op**'s output domain, $\mathbf{D}_{out}(\text{mul_op})$, must be
1360 the same as the **add_op**'s domain $\mathbf{D}(\text{add_op})$.

1361 Return Values

1362 **GrB_SUCCESS** operation completed successfully.

1363 **GrB_PANIC** unknown internal error.

1364 **GrB_OUT_OF_MEMORY** not enough memory available for this method to complete.

1365 **GrB_UNINITIALIZED_OBJECT** the **add_op** (for user-define monoids) object has not been initialized
1366 with a call to **GrB_Monoid_new** or the **mul_op** (for user-defined
1367 operators) object has not been not been initialized by a call to
1368 **GrB_BinaryOp_new**.

1369 **GrB_NULL_POINTER** semiring pointer is NULL.

1370 **GrB_DOMAIN_MISMATCH** the output domain of **mul_op** does not match the domain of the
1371 **add_op** monoid.

1372 Description

1373 The **Semiring_new** method creates a new semiring:

$$1374 \quad S = \langle \mathbf{D}_{out}(\text{mul_op}), \mathbf{D}_{in_1}(\text{mul_op}), \mathbf{D}_{in_2}(\text{mul_op}), \text{add_op}, \text{mul_op}, \mathbf{0}(\text{add_op}) \rangle$$

1375 and returns a handle to it in **semiring**. Note that $\mathbf{D}_{out}(\text{mul_op})$ must be the same as $\mathbf{D}(\text{add_op})$.

1376 If **add_op** is not commutative, then GraphBLAS operations using this semiring will be undefined.

1377 It is not an error to call this method more than once on the same variable; however, the handle to
1378 the previously created object will be overwritten.

1379 **4.2.2.6 IndexUnaryOp_new: Construct a new GraphBLAS index unary operator [Scott:**
1380 **NEW CONTENT]**

1381 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its
1382 types (domains).

1383 **C Syntax**

```
1384     GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp    *index_unary_op,  
1385                                   void (*index_unary_func)(void*,  
1386                                                             const void*,  
1387                                                             GrB_Index,  
1388                                                             GrB_Index,  
1389                                                             const void*),  
1390                                   GrB_Type             d_out,  
1391                                   GrB_Type             d_in1,  
1392                                   GrB_Type             d_in2);
```

1393 **Parameters**

1394 **index_unary_op** (INOUT) On successful return, contains a handle to the newly created Graph-
1395 BLAS index unary operator object.

1396 **index_unary_func** (IN) A pointer to a user-defined function that takes input parameters of types
1397 **d_in1**, **GrB_Index**, **GrB_Index** and **d_in2** and returns a value of type **d_out**. Ex-
1398 cept for the **GrB_Index** parameters, all are passed as **void** pointers. Specifically
1399 the signature of the function is expected to be of the form:

```
1400         void func(void      *out,  
1401                   const void *in1,  
1402                   GrB_Index  row_index,  
1403                   GrB_Index  col_index,  
1404                   const void *in2);  
1405
```

1406 **d_out** (IN) The **GrB_Type** of the return value of the index unary operator being created.
1407 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1408 GraphBLAS type.

1409 **d_in1** (IN) The **GrB_Type** of the first input argument of the index unary operator being
1410 created and corresponds to the stored values of the **GrB_Vector** or **GrB_Matrix**
1411 being operated on. Should be one of the predefined GraphBLAS types in Ta-
1412 ble 3.2, or a user-defined GraphBLAS type.

1413 **d_in2** (IN) The **GrB_Type** of the last input argument of the index unary operator be-
1414 ing created and corresponds to a scalar provided by the GraphBLAS operation

1415 that uses this operator. Should be one of the predefined GraphBLAS types in
1416 Table 3.2, or a user-defined GraphBLAS type.

1417 Return Values

1418 GrB_SUCCESS operation completed successfully.

1419 GrB_PANIC unknown internal error.

1420 GrB_OUT_OF_MEMORY not enough memory available for operation.

1421 GrB_UNINITIALIZED_OBJECT the GrB_Type (for user-defined types) has not been initialized by a
1422 call to GrB_Type_new.

1423 GrB_NULL_POINTER index_unary_op or index_unary_func pointer is NULL.

1424 Description

1425 The IndexUnaryOp_new methods creates a new GraphBLAS index unary operator

1426 $f_i = \langle \mathbf{D}(d_out), \mathbf{D}(d_in1), \mathbf{D}(GrB_Index), \mathbf{D}(GrB_Index), \mathbf{D}(d_in2), index_unary_func \rangle$

1427 and returns a handle to it in index_unary_op.

1428 The implementation of index_unary_func must be such that it works even if any of the d_out,
1429 d_in1, and d_in2 arguments are aliased to each other. In other words, for all invocations of the
1430 function:

1431 `index_unary_func(out,in1,row_index,col_index,n,in2);`

1432 the value of out must be the same as if the following code was executed (shown here for matrices):

```
1433 GrB_Index row_index = ...;  
1434 GrB_Index col_index = ...;  
1435 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));  
1436 D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));  
1437 memcpy(tmp1,in1,sizeof(D(d_in1)));  
1438 memcpy(tmp2,in2,sizeof(D(d_in2)));  
1439 index_unary_func(out,tmp1,row_index,col_index,tmp2);  
1440 free(tmp2);  
1441 free(tmp1);
```

1442 It is not an error to call this method more than once on the same variable; however, the handle to
1443 the previously created object will be overwritten.

1444 4.2.3 Scalar methods

1445 4.2.3.1 Scalar_new: Construct a new scalar

1446 Creates a new empty scalar with specified domain.

1447 C Syntax

```
1448         GrB_Info GrB_Scalar_new(GrB_Scalar *s,  
1449                                GrB_Type    d);
```

1450 Parameters

1451 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1452 scalar.

1453 **d** (IN) The type corresponding to the domain of the scalar being created. Can be
1454 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1455 GraphBLAS type.

1456 Return Values

1457 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1458 blocking mode, this indicates that the API checks for the input
1459 arguments passed successfully. Either way, output scalar **s** is ready
1460 to be used in the next method of the sequence.

1461 **GrB_PANIC** Unknown internal error.

1462 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1463 GraphBLAS objects (input or output) is in an invalid state caused
1464 by a previous execution error. Call **GrB_error()** to access any error
1465 messages generated by the implementation.

1466 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1467 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
1468 (needed for user-defined types).

1469 **GrB_NULL_POINTER** The **s** pointer is NULL.

1470 Description

1471 Creates a new GraphBLAS scalar **s** of domain **D(d)** and empty **L(s)**. The method returns a handle
1472 to the new scalar in **s**.

1473 It is not an error to call this method more than once on the same variable; however, the handle to
1474 the previously created object will be overwritten.

1475 **4.2.3.2 Scalar_dup: Construct a copy of a GraphBLAS scalar**

1476 Creates a new scalar with the same domain and contents as another scalar.

1477 **C Syntax**

```
1478         GrB_Info GrB_Scalar_dup(GrB_Scalar      *t,  
1479                                const GrB_Scalar  s);
```

1480 **Parameters**

1481 **t** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1482 scalar.

1483 **s** (IN) The GraphBLAS scalar to be duplicated.

1484 **Return Values**

1485 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1486 blocking mode, this indicates that the API checks for the input
1487 arguments passed successfully. Either way, output scalar **t** is ready
1488 to be used in the next method of the sequence.

1489 **GrB_PANIC** Unknown internal error.

1490 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1491 GraphBLAS objects (input or output) is in an invalid state caused
1492 by a previous execution error. Call **GrB_error()** to access any error
1493 messages generated by the implementation.

1494 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1495 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to
1496 **Scalar_new** or **Scalar_dup**.

1497 **GrB_NULL_POINTER** The **t** pointer is NULL.

1498 **Description**

1499 Creates a new scalar *t* of domain **D(s)** and contents **L(s)**. The method returns a handle to the new
1500 scalar in **t**.

1501 It is not an error to call this method more than once with the same output variable; however, the
1502 handle to the previously created object will be overwritten.

1503 **4.2.3.3 Scalar_clear: Clear/remove a stored value from a scalar**

1504 Removes the stored value from a scalar.

1505 **C Syntax**

```
1506         GrB_Info GrB_Scalar_clear(GrB_Scalar s);
```

1507 **Parameters**

1508 s (INOUT) An existing GraphBLAS scalar to clear.

1509 **Return Values**

1510 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1511 blocking mode, this indicates that the API checks for the input
1512 arguments passed successfully. Either way, output scalar s is ready
1513 to be used in the next method of the sequence.

1514 GrB_PANIC Unknown internal error.

1515 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1516 GraphBLAS objects (input or output) is in an invalid state caused
1517 by a previous execution error. Call GrB_error() to access any error
1518 messages generated by the implementation.

1519 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1520 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, s, has not been initialized by a call to
1521 Scalar_new or Scalar_dup.

1522 **Description**

1523 Removes the stored value from an existing scalar. After the call, **L(s)** is empty. The size of the
1524 scalar does not change.

1525 **4.2.3.4 Scalar_nvals: Number of stored elements in a scalar**

1526 Retrieve the number of stored elements in a scalar (either zero or one).

1527 C Syntax

```
1528         GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,  
1529                                   const GrB_Scalar s);
```

1530 Parameters

1531 **nvals** (OUT) On successful return, this is set to the number of stored elements in the
1532 scalar (zero or one).

1533 **s** (IN) An existing GraphBLAS scalar being queried.

1534 Return Values

1535 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1536 cessfully and the value of **nvals** has been set.

1537 **GrB_PANIC** Unknown internal error.

1538 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1539 GraphBLAS objects (input or output) is in an invalid state caused
1540 by a previous execution error. Call **GrB_error()** to access any error
1541 messages generated by the implementation.

1542 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1543 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to
1544 **Scalar_new** or **Scalar_dup**.

1545 **GrB_NULL_POINTER** The **nvals** pointer is NULL.

1546 Description

1547 Return **nvals(s)** in **nvals**. This is the number of stored elements in scalar **s**, which is the size of
1548 **L(s)**, and can only be either zero or one (see Section 3.5.1).

1549 4.2.3.5 Scalar_setElement: Set the single element in a scalar

1550 Set the single element of a scalar to a given value.

1551 C Syntax

```
1552         GrB_Info GrB_Scalar_setElement(GrB_Scalar  s,  
1553                                         <type>    val);
```


1554 Parameters

1555 **s** (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1556 **val** (IN) Scalar value to assign. The type must be compatible with the domain of **s**.

1557 Return Values

1558 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1559 blocking mode, this indicates that the compatibility tests on in-
1560 dex/dimensions and domains for the input arguments passed suc-
1561 cessfully. Either way, the output scalar **s** is ready to be used in the
1562 next method of the sequence.

1563 **GrB_PANIC** Unknown internal error.

1564 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1565 GraphBLAS objects (input or output) is in an invalid state caused
1566 by a previous execution error. Call **GrB_error()** to access any error
1567 messages generated by the implementation.

1568 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1569 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to
1570 **Scalar_new** or **Scalar_dup**.

1571 **GrB_DOMAIN_MISMATCH** The domains of **s** and **val** are incompatible.

1572 Description

1573 First, **val** and output GraphBLAS scalar are tested for domain compatibility as follows: **D(val)** must
1574 be compatible with **D(s)**. Two domains are compatible with each other if values from one domain
1575 can be cast to values in the other domain as per the rules of the C language. In particular, domains
1576 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-
1577 patible with itself. If any compatibility rule above is violated, execution of **GrB_Scalar_setElement**
1578 ends and the domain mismatch error listed above is returned.

1579 We are now ready to carry out the assignment **val**; that is:

$$1580 \qquad \qquad \qquad \mathbf{s}(0) = \mathbf{val}$$

1581 If **s** already had a stored value, it will be overwritten; otherwise, the new value is stored in **s**.

1582 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new contents
1583 of **s** is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method exits with
1584 return value **GrB_SUCCESS** and the new content of scalar **s** is as defined above but may not be
1585 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1586 **4.2.3.6 Scalar_extractElement: Extract a single element from a scalar.**

1587 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

1588 C Syntax

```
1589         GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1590                                         const GrB_Scalar s);
```

1591 Parameters

1592 **val** (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain
1593 of scalar **s**. On successful return, **val** holds the result of the operation, and any
1594 previous value in **val** is overwritten.

1595 **s** (IN) The GraphBLAS scalar from which an element is extracted.

1596 Return Values

1597 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1598 cessfully. This indicates that the compatibility tests on dimensions
1599 and domains for the input arguments passed successfully, and the
1600 output scalar, **val**, has been computed and is ready to be used in
1601 the next method of the sequence.

1602 **GrB_PANIC** Unknown internal error.

1603 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1604 GraphBLAS objects (input or output) is in an invalid state caused
1605 by a previous execution error. Call **GrB_error()** to access any error
1606 messages generated by the implementation.

1607 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1608 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to
1609 **Scalar_new** or **Scalar_dup**.

1610 **GrB_NULL_POINTER** **val** pointer is NULL.

1611 **GrB_DOMAIN_MISMATCH** The domains of the scalar or scalar are incompatible.

1612 **GrB_NO_VALUE** There is no stored value in the scalar.

1613 Description

1614 First, `val` and input GraphBLAS scalar are tested for domain compatibility as follows: `D(val)`
1615 must be compatible with `D(s)`. Two domains are compatible with each other if values from
1616 one domain can be cast to values in the other domain as per the rules of the C language. In
1617 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1618 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1619 `GrB_Scalar_extractElement` ends and the domain mismatch error listed above is returned.

1620 Then, if no value is currently stored in the GraphBLAS scalar, the method returns `GrB_NO_VALUE`
1621 and `val` remains unchanged.

1622 Finally the extract into the output argument, `val` can be performed; that is:

1623
$$\text{val} = \text{s}(0)$$

1624 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1625 `GrB_SUCCESS`, the new contents of `val` are as defined above.

1626 4.2.4 Vector methods

1627 4.2.4.1 Vector_new: Construct new vector

1628 Creates a new vector with specified domain and size.

1629 C Syntax

```
1630      GrB_Info GrB_Vector_new(GrB_Vector *v,  
1631                             GrB_Type    d,  
1632                             GrB_Index   nsize);
```

1633 Parameters

1634 `v` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1635 vector.

1636 `d` (IN) The type corresponding to the domain of the vector being created. Can be
1637 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1638 GraphBLAS type.

1639 `nsize` (IN) The size of the vector being created.

1640 Return Values

1641 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1642 blocking mode, this indicates that the API checks for the input

1643 arguments passed successfully. Either way, output vector \mathbf{v} is ready
1644 to be used in the next method of the sequence.

1645 **GrB_PANIC** Unknown internal error.

1646 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1647 GraphBLAS objects (input or output) is in an invalid state caused
1648 by a previous execution error. Call **GrB_error()** to access any error
1649 messages generated by the implementation.

1650 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1651 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
1652 (needed for user-defined types).

1653 **GrB_NULL_POINTER** The \mathbf{v} pointer is NULL.

1654 **GrB_INVALID_VALUE** \mathbf{nsz} is zero or outside the range of the type **GrB_Index**.

1655 Description

1656 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
1657 to the new vector in \mathbf{v} .

1658 It is not an error to call this method more than once on the same variable; however, the handle to
1659 the previously created object will be overwritten.

1660 4.2.4.2 Vector_dup: Construct a copy of a GraphBLAS vector

1661 Creates a new vector with the same domain, size, and contents as another vector.

1662 C Syntax

```
1663 GrB_Info GrB_Vector_dup(GrB_Vector      *w,  
1664                        const GrB_Vector u);
```

1665 Parameters

1666 \mathbf{w} (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1667 vector.

1668 \mathbf{u} (IN) The GraphBLAS vector to be duplicated.

1669 Return Values

1670 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1671 blocking mode, this indicates that the API checks for the input
1672 arguments passed successfully. Either way, output vector **w** is ready
1673 to be used in the next method of the sequence.

1674 GrB_PANIC Unknown internal error.

1675 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1676 GraphBLAS objects (input or output) is in an invalid state caused
1677 by a previous execution error. Call **GrB_error()** to access any error
1678 messages generated by the implementation.

1679 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1680 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, **u**, has not been initialized by a call to
1681 **Vector_new** or **Vector_dup**.

1682 GrB_NULL_POINTER The **w** pointer is **NULL**.

1683 Description

1684 Creates a new vector **w** of domain **D(u)**, size **size(u)**, and contents **L(u)**. The method returns a
1685 handle to the new vector in **w**.

1686 It is not an error to call this method more than once on the same variable; however, the handle to
1687 the previously created object will be overwritten.

1688 4.2.4.3 Vector_resize: Resize a vector

1689 Changes the size of an existing vector.

1690 C Syntax

```
1691           GrB_Info GrB_Vector_resize(GrB_Vector w,  
1692                                       GrB_Index nsize);
```

1693 Parameters

1694 **w** (INOUT) An existing Vector object that is being resized.

1695 **nsize** (IN) The new size of the vector. It can be smaller or larger than the current size.

1696 Return Values

1697 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1698 blocking mode, this indicates that the API checks for the input
1699 arguments passed successfully. Either way, output vector w is ready
1700 to be used in the next method of the sequence.

1701 GrB_PANIC Unknown internal error.

1702 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1703 GraphBLAS objects (input or output) is in an invalid state caused
1704 by a previous execution error. Call `GrB_error()` to access any error
1705 messages generated by the implementation.

1706 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1707 GrB_NULL_POINTER The w pointer is NULL.

1708 GrB_INVALID_VALUE nsz is zero or outside the range of the type `GrB_Index`.

1709 Description

1710 Changes the size of w to nsz . The domain $\mathbf{D}(w)$ of vector w remains the same. The contents $\mathbf{L}(w)$
1711 are modified as described below.

1712 Let $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$ when the method is called. When the method returns, $w = \langle \mathbf{D}(w), nsz, \mathbf{L}'(w) \rangle$
1713 where $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < nsz)\}$. That is, all elements of w with index greater
1714 than or equal to the new vector size (nsz) are dropped.

1715 4.2.4.4 Vector_clear: Clear a vector

1716 Removes all the elements (tuples) from a vector.

1717 C Syntax

1718 GrB_Info GrB_Vector_clear(GrB_Vector v);

1719 Parameters

1720 v (INOUT) An existing GraphBLAS vector to clear.

1721 Return Values

1722 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1723 blocking mode, this indicates that the API checks for the input

1724 arguments passed successfully. Either way, output vector v is ready
1725 to be used in the next method of the sequence.

1726 **GrB_PANIC** Unknown internal error.

1727 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1728 GraphBLAS objects (input or output) is in an invalid state caused
1729 by a previous execution error. Call **GrB_error()** to access any error
1730 messages generated by the implementation.

1731 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1732 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, v , has not been initialized by a call to
1733 **Vector_new** or **Vector_dup**.

1734 Description

1735 Removes all elements (tuples) from an existing vector. After the call to **GrB_Vector_clear(v)**,
1736 $L(v) = \emptyset$. The size of the vector does not change.

1737 4.2.4.5 Vector_size: Size of a vector

1738 Retrieve the size of a vector.

1739 C Syntax

```
1740 GrB_Info GrB_Vector_size(GrB_Index      *nsize,  
1741                          const GrB_Vector v);
```

1742 Parameters

1743 **nsize** (OUT) On successful return, is set to the size of the vector.

1744 **v** (IN) An existing GraphBLAS vector being queried.

1745 Return Values

1746 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1747 cessfully and the value of **nsize** has been set.

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_UNINITIALIZED_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to
1754 `Vector_new` or `Vector_dup`.

1755 GrB_NULL_POINTER `nsz` pointer is NULL.

1756 Description

1757 Return `size(v)` in `nsz`.

1758 4.2.4.6 Vector_nvals: Number of stored elements in a vector

1759 Retrieve the number of stored elements (tuples) in a vector.

1760 C Syntax

```
1761 GrB_Info GrB_Vector_nvals(GrB_Index      *nvals,  
1762                          const GrB_Vector v);
```

1763 Parameters

1764 `nvals` (OUT) On successful return, this is set to the number of stored elements (tuples)
1765 in the vector.

1766 `v` (IN) An existing GraphBLAS vector being queried.

1767 Return Values

1768 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1769 cessfully and the value of `nvals` has been set.

1770 GrB_PANIC Unknown internal error.

1771 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1772 GraphBLAS objects (input or output) is in an invalid state caused
1773 by a previous execution error. Call `GrB_error()` to access any error
1774 messages generated by the implementation.

1775 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1776 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to
1777 `Vector_new` or `Vector_dup`.

1778 GrB_NULL_POINTER The `nvals` pointer is NULL.

1779 **Description**

1780 Return **nvals**(**v**) in **nvals**. This is the number of stored elements in vector **v**, which is the size of
1781 **L**(**v**) (see Section 3.5.2).

1782 **4.2.4.7 Vector_build: Store elements from tuples into a vector**

1783 **C Syntax**

```
1784         GrB_Info GrB_Vector_build(GrB_Vector          w,  
1785                                   const GrB_Index     *indices,  
1786                                   const <type>         *values,  
1787                                   GrB_Index             n,  
1788                                   const GrB_BinaryOp     dup);
```

1789 **Parameters**

1790 **w** (INOUT) An existing Vector object to store the result.

1791 **indices** (IN) Pointer to an array of indices.

1792 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of
1793 vector **w**.

1794 **n** (IN) The number of entries contained in each array (the same for **indices** and **values**).

1795 **dup** (IN) An associative and commutative binary operator to apply when duplicate
1796 values for the same location are present in the input arrays. All three domains of
1797 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
1798 then duplicate locations will result in an error.

1799 **Return Values**

1800 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1801 blocking mode, this indicates that the API checks for the input
1802 arguments passed successfully. Either way, output vector **w** is
1803 ready to be used in the next method of the sequence.

1804 **GrB_PANIC** Unknown internal error.

1805 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
1806 opaque GraphBLAS objects (input or output) is in an invalid
1807 state caused by a previous execution error. Call **GrB_error()** to
1808 access any error messages generated by the implementation.

1809 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1810 GrB_UNINITIALIZED_OBJECT Either `w` has not been initialized by a call to `GrB_Vector_new`
 1811 or by `GrB_Vector_dup`, or `dup` has not been initialized by a call
 1812 to `GrB_BinaryOp_new`.

1813 GrB_NULL_POINTER `indices` or `values` pointer is NULL.

1814 GrB_INDEX_OUT_OF_BOUNDS A value in `indices` is outside the allowed range for `w`.

1815 GrB_DOMAIN_MISMATCH Either the domains of the GraphBLAS binary operator `dup` are
 1816 not all the same, or the domains of `values` and `w` are incompatible
 1817 with each other or D_{dup} .

1818 GrB_OUTPUT_NOT_EMPTY Output vector `w` already contains valid tuples (elements). In
 1819 other words, `GrB_Vector_nvals(C)` returns a positive value.

1820 GrB_INVALID_VALUE `indices` contains a duplicate location and `dup` is GrB_NULL.

1821 Description

1822 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
 1823 from `w` in its domain; otherwise, $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$.

1824 Each tuple $\{\text{indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the output in the form of

$$1825 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1826 If multiple values for the same location are present in the input arrays and `dup` is not GrB_NULL,
 1827 `dup` is used to reduce the values before assignment into $\tilde{\mathbf{w}}$ as follows:

$$1828 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1829 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{w}}$ is copied into `w` via typecasting its
 1830 values to $\mathbf{D}(\mathbf{w})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

1831 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1832 It is an error to call this function on an output object with existing elements. In other words,
 1833 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1834 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`
 1835 or `values`.

1836 4.2.4.8 Vector_setElement: Set a single element in a vector

1837 Set one element of a vector to a given value.

1838 C Syntax

```
1839 // scalar value
1840 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1841                               <type>         val,
1842                               GrB_Index       index);
1843
1844 // GraphBLAS scalar
1845 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1846                               const GrB_Scalar s,
1847                               GrB_Index       index);
```

1848 Parameters

1849 **w** (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1850 **val** or **s** (IN) Scalar assign. Its domain (type) must be compatible with the domain of **w**.

1851 **index** (IN) The location of the element to be assigned.

1852 Return Values

1853 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1854 blocking mode, this indicates that the compatibility tests on in-
1855 dex/dimensions and domains for the input arguments passed suc-
1856 cessfully. Either way, the output vector **w** is ready to be used in
1857 the next method of the sequence.

1858 **GrB_PANIC** Unknown internal error.

1859 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1860 GraphBLAS objects (input or output) is in an invalid state caused
1861 by a previous execution error. Call **GrB_error()** to access any error
1862 messages generated by the implementation.

1863 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1864 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been
1865 initialized by a call to a respective constructor.

1866 **GrB_INVALID_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1867 **GrB_DOMAIN_MISMATCH** The domains of the vector and the scalar are incompatible.

1868 Description

1869 First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)**
 1870 must be compatible with **D(w)**. Two domains are compatible with each other if values from
 1871 one domain can be cast to values in the other domain as per the rules of the C language. In
 1872 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
 1873 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
 1874 **GrB_Vector_setElement** ends and the domain mismatch error listed above is returned.

1875 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1876 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1877 If this condition is violated, execution of **GrB_Vector_setElement** ends and the invalid index error
 1878 listed above is returned.

1879 We are now ready to carry out the assignment; that is:

$$1880 \quad \mathbf{w}(\text{index}) = \begin{cases} \mathbf{L}(\mathbf{s}), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

1881 In the case of a transparent scalar or if **L(s)** is not empty, then a value will be stored at the
 1882 specified location in **w**, overwriting any value that may have been stored there before. In the case
 1883 of a GraphBLAS scalar, if **L(s)** is empty, then any value stored at the specified location in **w** will
 1884 be removed.

1885 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new contents
 1886 of **w** is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method exits with
 1887 return value **GrB_SUCCESS** and the new contents of vector **w** is as defined above but may not be
 1888 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1889 4.2.4.9 Vector_removeElement: Remove an element from a vector

1890 Remove (annihilate) one stored element from a vector.

1891 C Syntax

```
1892         GrB_Info GrB_Vector_removeElement(GrB_Vector  w,
1893                                         GrB_Index    index);
```

1894 Parameters

1895 **w** (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1896 **index** (IN) The location of the element to be removed.

1897 Return Values

1898 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1899 blocking mode, this indicates that the compatibility tests on in-
1900 dex/dimensions and domains for the input arguments passed suc-
1901 cessfully. Either way, the output vector **w** is ready to be used in
1902 the next method of the sequence.

1903 GrB_PANIC Unknown internal error.

1904 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1905 GraphBLAS objects (input or output) is in an invalid state caused
1906 by a previous execution error. Call **GrB_error()** to access any error
1907 messages generated by the implementation.

1908 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1909 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, **w**, has not been initialized by a call to
1910 **Vector_new** or **Vector_dup**.

1911 GrB_INVALID_INDEX **index** specifies a location that is outside the dimensions of **w**.

1912 Description

1913 First, the **index** parameter is checked for a valid value where the following condition must hold:

$$1914 \qquad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1915 If this condition is violated, execution of **GrB_Vector_removeElement** ends and the invalid index
1916 error listed above is returned.

1917 We are now ready to carry out the removal of a value that may be stored at the location specified
1918 by **index**. If a value does not exist at the specified location in **w**, no error is reported and the
1919 operation has no effect on the state of **w**. In either case, the following will be true on return from
1920 the method: **index** \notin **ind(w)**.

1921 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new contents
1922 of **w** is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method exits with
1923 return value **GrB_SUCCESS** and the new content of vector **w** is as defined above but may not be
1924 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1925 4.2.4.10 Vector_extractElement: Extract a single element from a vector.

1926 Extract one element of a vector into a scalar.

1927 C Syntax

```
1928      // scalar value
1929      GrB_Info GrB_Vector_extractElement(<type>          *val,
1930                                         const GrB_Vector u,
1931                                         GrB_Index       index);
1932
1933      // GraphBLAS scalar
1934      GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1935                                         const GrB_Vector u,
1936                                         GrB_Index       index);
```

1937 Parameters

1938 **val** or **s** (INOUT) An existing scalar of whose domain is compatible with the domain of vector
1939 **u**. On successful return, this scalar holds the result of the extract. Any previous
1940 value stored in **val** or **s** is overwritten.

1941 **u** (IN) The GraphBLAS vector from which an element is extracted.

1942 **index** (IN) The location in **u** to extract.

1943 Return Values

1944 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1945 cessfully. This indicates that the compatibility tests on dimensions
1946 and domains for the input arguments passed successfully, and the
1947 output scalar, **val** or **s**, has been computed and is ready to be used
1948 in the next method of the sequence.

1949 **GrB_NO_VALUE** When using the transparent scalar, **val**, this is returned when there
1950 is no stored value at specified location.

1951 **GrB_PANIC** Unknown internal error.

1952 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1953 GraphBLAS objects (input or output) is in an invalid state caused
1954 by a previous execution error. Call **GrB_error()** to access any error
1955 messages generated by the implementation.

1956 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1957 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **u**, or scalar, **s**, has not been initialized by
1958 a call to a corresponding constructor.

1959 **GrB_NULL_POINTER** **val** pointer is NULL.

1960 **GrB_INVALID_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1961 GrB_DOMAIN_MISMATCH The domains of the vector and scalar are incompatible.

1962 Description

1963 First, the scalar and input vector are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ or $\mathbf{D}(\mathbf{s})$
1964 must be compatible with $\mathbf{D}(\mathbf{u})$. Two domains are compatible with each other if values from
1965 one domain can be cast to values in the other domain as per the rules of the C language. In
1966 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1967 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1968 GrB_Vector_extractElement ends and the domain mismatch error listed above is returned.

1969 Then, the index parameter is checked for a valid value where the following condition must hold:

$$1970 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1971 If this condition is violated, execution of GrB_Vector_extractElement ends and the invalid index
1972 error listed above is returned.

1973 We are now ready to carry out the extract into the output scalar; that is:

$$1974 \qquad \left. \begin{array}{l} \mathbf{L}(\mathbf{s}) \\ \text{val} \end{array} \right\} = \mathbf{u}(\text{index})$$

1975 If $\text{index} \in \text{ind}(\mathbf{u})$, then the corresponding value from \mathbf{u} is copied into \mathbf{s} or val with casting as
1976 necessary. If $\text{index} \notin \text{ind}(\mathbf{u})$, then one of the follow occurs depending on output scalar type:

- 1977 • The GraphBLAS scalar, \mathbf{s} , is cleared and GrB_SUCCESS is returned.
- 1978 • The non-opaque scalar, val , is unchanged, and GrB_NO_VALUE is returned.

1979 When using the non-opaque scalar variant (val) in both GrB_BLOCKING mode GrB_NONBLOCKING
1980 mode, the new contents of val are as defined above if the method exits with return value GrB_SUCCESS
1981 or GrB_NO_VALUE.

1982 When using the GraphBLAS scalar variant (\mathbf{s}) with a GrB_SUCCESS return value, the method
1983 exits and the new contents of \mathbf{s} is as defined above and fully computed in GrB_BLOCKING mode.
1984 In GrB_NONBLOCKING mode, the new contents of \mathbf{s} is as defined above but may not be fully
1985 computed; however, it can be used in the next GraphBLAS method call in a sequence.

1986 4.2.4.11 Vector_extractTuples: Extract tuples from a vector

1987 Extract the contents of a GraphBLAS vector into non-opaque data structures.

1988 C Syntax

1989 GrB_Info GrB_Vector_extractTuples(GrB_Index *indices,

```

1990                                     <type>                *values,
1991                                     GrB_Index              *n,
1992                                     const GrB_Vector        v);
1993

```

1994 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored
1995 values' indices.

1996 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
1997 the stored values whose type is compatible with **D(v)**.

1998 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the
1999 **values** and **indices** arrays can hold. Upon return, it will contain the number of
2000 values written to the arrays.

2001 **v** (IN) An existing GraphBLAS vector.

2002 Return Values

2003 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2004 cessfully. This indicates that the compatibility tests on the input
2005 argument passed successfully, and the output arrays, **indices** and
2006 **values**, have been computed.

2007 **GrB_PANIC** Unknown internal error.

2008 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2009 GraphBLAS objects (input or output) is in an invalid state caused
2010 by a previous execution error. Call **GrB_error()** to access any error
2011 messages generated by the implementation.

2012 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2013 **GrB_INSUFFICIENT_SPACE** Not enough space in **indices** and **values** (as indicated by the **n** pa-
2014 rameter) to hold all of the tuples that will be extracted.

2015 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to
2016 **Vector_new** or **Vector_dup**.

2017 **GrB_NULL_POINTER** **indices**, **values**, or **n** pointer is NULL.

2018 **GrB_DOMAIN_MISMATCH** The domains of the **v** vector or **values** array are incompatible with
2019 one another.

2020 Description

2021 This method will extract all the tuples from the GraphBLAS vector **v**. The values associated
2022 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.

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 k th location in output vectors:

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

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 $0 \leq j < k$.

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.

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.

4.2.5 Matrix methods

4.2.5.1 Matrix_new: Construct new matrix

Creates a new matrix with specified domain and dimensions.

C Syntax

```
GrB_Info GrB_Matrix_new(GrB_Matrix *A,
                        GrB_Type    d,
                        GrB_Index    nrows,
                        GrB_Index    ncols);
```

Parameters

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

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

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

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

2051 Return Values

- 2052 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2053 blocking mode, this indicates that the API checks for the input ar-
2054 guments passed successfully. Either way, output matrix **A** is ready
2055 to be used in the next method of the sequence.
- 2056 GrB_PANIC Unknown internal error.
- 2057 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2058 GraphBLAS objects (input or output) is in an invalid state caused
2059 by a previous execution error. Call `GrB_error()` to access any error
2060 messages generated by the implementation.
- 2061 GrB_OUT_OF_MEMORY Not enough memory available for operation.
- 2062 GrB_UNINITIALIZED_OBJECT The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`
2063 (needed for user-defined types).
- 2064 GrB_NULL_POINTER The **A** pointer is NULL.
- 2065 GrB_INVALID_VALUE `nrows` or `ncols` is zero or outside the range of the type `GrB_Index`.

2066 Description

- 2067 Creates a new matrix **A** of domain **D**(**d**), size `nrows` \times `ncols`, and empty **L**(**A**). The method returns
2068 a handle to the new matrix in **A**.
- 2069 It is not an error to call this method more than once on the same variable; however, the handle to
2070 the previously created object will be overwritten.

2071 4.2.5.2 Matrix_dup: Construct a copy of a GraphBLAS matrix

- 2072 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

2073 C Syntax

```
2074           GrB_Info GrB_Matrix_dup(GrB_Matrix        *C,  
2075                                    const GrB_Matrix  A);
```

2076 Parameters

- 2077 **C** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2078 matrix.
- 2079 **A** (IN) The GraphBLAS matrix to be duplicated.

2080 Return Values

2081 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2082 blocking mode, this indicates that the API checks for the input
2083 arguments passed successfully. Either way, output matrix **C** is ready
2084 to be used in the next method of the sequence.

2085 GrB_PANIC Unknown internal error.

2086 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2087 GraphBLAS objects (input or output) is in an invalid state caused
2088 by a previous execution error. Call **GrB_error()** to access any error
2089 messages generated by the implementation.

2090 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2091 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, **A**, has not been initialized by a call to
2092 any matrix constructor.

2093 GrB_NULL_POINTER The **C** pointer is **NULL**.

2094 Description

2095 Creates a new matrix **C** of domain **D(A)**, size **nrows(A) × ncols(A)**, and contents **L(A)**. It returns
2096 a handle to it in **C**.

2097 It is not an error to call this method more than once on the same variable; however, the handle to
2098 the previously created object will be overwritten.

2099 4.2.5.3 Matrix_diag: Construct a diagonal GraphBLAS matrix

2100 Creates a new matrix with the same domain and contents as a **GrB_Vector**, and square dimensions
2101 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

2102 C Syntax

```
2103           GrB_Info GrB_Matrix_diag(GrB_Matrix        *C,  
2104                                    const GrB_Vector   v,  
2105                                    int64_t           k);
```

2106 Parameters

2107 **C** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2108 matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

2139 Parameters

2140 C (INOUT) An existing Matrix object that is being resized.

2141 nrows (IN) The new number of rows of the matrix. It can be smaller or larger than the
2142 current number of rows.

2143 ncols (IN) The new number of columns of the matrix. It can be smaller or larger than
2144 the current number of columns.

2145 Return Values

2146 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2147 blocking mode, this indicates that the API checks for the input
2148 arguments passed successfully. Either way, output matrix C is ready
2149 to be used in the next method of the sequence.

2150 GrB_PANIC Unknown internal error.

2151 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2152 GraphBLAS objects (input or output) is in an invalid state caused
2153 by a previous execution error. Call GrB_error() to access any error
2154 messages generated by the implementation.

2155 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2156 GrB_NULL_POINTER The C pointer is NULL.

2157 GrB_INVALID_VALUE nrows or ncols is zero or outside the range of the type GrB_Index.

2158 Description

2159 Changes the number of rows and columns of C to nrows and ncols, respectively. The domain $\mathbf{D}(\mathbf{C})$
2160 of matrix C remains the same. The contents $\mathbf{L}(\mathbf{C})$ are modified as described below.

2161 Let $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), M, N, \mathbf{L}(\mathbf{C}) \rangle$ when the method is called. When the method returns C is modified
2162 to $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \text{nrows}, \text{ncols}, \mathbf{L}'(\mathbf{C}) \rangle$ where $\mathbf{L}'(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j, C_{ij}) \in \mathbf{L}(\mathbf{C}) \wedge (i < \text{nrows}) \wedge (j < \text{ncols})\}$. That is, all elements of C with row index greater than or equal to nrows or column index
2163 greater than or equal to ncols are dropped.
2164

2165 4.2.5.5 Matrix_clear: Clear a matrix

2166 Removes all elements (tuples) from a matrix.

2167 C Syntax

2168 GrB_Info GrB_Matrix_clear(GrB_Matrix A);

2169 Parameters

2170 A (IN) An existing GraphBLAS matrix to clear.

2171 Return Values

2172 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2173 blocking mode, this indicates that the API checks for the input ar-
2174 guments passed successfully. Either way, output matrix A is ready
2175 to be used in the next method of the sequence.

2176 GrB_PANIC Unknown internal error.

2177 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2178 GraphBLAS objects (input or output) is in an invalid state caused
2179 by a previous execution error. Call GrB_error() to access any error
2180 messages generated by the implementation.

2181 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2182 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2183 any matrix constructor.

2184 Description

2185 Removes all elements (tuples) from an existing matrix. After the call to GrB_Matrix_clear(A),
2186 $L(A) = \emptyset$. The dimensions of the matrix do not change.

2187 4.2.5.6 Matrix_nrows: Number of rows in a matrix

2188 Retrieve the number of rows in a matrix.

2189 C Syntax

```
2190           GrB_Info GrB_Matrix_nrows(GrB_Index           *nrows,  
2191                                       const GrB_Matrix  A);
```

2192 Parameters

2193 nrows (OUT) On successful return, contains the number of rows in the matrix.

2194 A (IN) An existing GraphBLAS matrix being queried.

2195 **Return Values**

2196 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2197 cessfully and the value of `nrows` has been set.

2198 GrB_PANIC Unknown internal error.

2199 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2200 GraphBLAS objects (input or output) is in an invalid state caused
2201 by a previous execution error. Call `GrB_error()` to access any error
2202 messages generated by the implementation.

2203 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
2204 any matrix constructor.

2205 GrB_NULL_POINTER `nrows` pointer is NULL.

2206 **Description**

2207 Return `nrows(A)` in `nrows` (the number of rows).

2208 **4.2.5.7 Matrix_ncols: Number of columns in a matrix**

2209 Retrieve the number of columns in a matrix.

2210 **C Syntax**

```
2211           GrB_Info GrB_Matrix_ncols(GrB_Index           *ncols,  
2212                                    const GrB_Matrix A);
```

2213 **Parameters**

2214 ncols (OUT) On successful return, contains the number of columns in the matrix.

2215 A (IN) An existing GraphBLAS matrix being queried.

2216 **Return Values**

2217 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2218 cessfully and the value of `ncols` has been set.

2219 GrB_PANIC Unknown internal error.

2220 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 2221 GraphBLAS objects (input or output) is in an invalid state caused
 2222 by a previous execution error. Call GrB_error() to access any error
 2223 messages generated by the implementation.

2224 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
 2225 any matrix constructor.

2226 GrB_NULL_POINTER ncols pointer is NULL.

2227 Description

2228 Return **ncols(A)** in **ncols** (the number of columns).

2229 4.2.5.8 Matrix_nvals: Number of stored elements in a matrix

2230 Retrieve the number of stored elements (tuples) in a matrix.

2231 C Syntax

```
2232           GrB_Info GrB_Matrix_nvals(GrB_Index           *nvals,  
2233                                    const GrB_Matrix A);
```

2234 Parameters

2235 **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in
 2236 the matrix.

2237 **A** (IN) An existing GraphBLAS matrix being queried.

2238 Return Values

2239 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
 2240 cessfully and the value of **nvals** has been set.

2241 GrB_PANIC Unknown internal error.

2242 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 2243 GraphBLAS objects (input or output) is in an invalid state caused
 2244 by a previous execution error. Call GrB_error() to access any error
 2245 messages generated by the implementation.

2246 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2247 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
 2248 any matrix constructor.

2249 GrB_NULL_POINTER The nvals pointer is NULL.

2250 Description

2251 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix A, which is the size of
 2252 **L(A)** (see Section 3.5.3).

2253 4.2.5.9 Matrix_build: Store elements from tuples into a matrix

2254 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);
```

2255 Parameters

2256 C (INOUT) An existing Matrix object to store the result.

2257 row_indices (IN) Pointer to an array of row indices.

2258 col_indices (IN) Pointer to an array of column indices.

2259 values (IN) Pointer to an array of scalars of a type that is compatible with the domain of
 2260 matrix, C.

2261 n (IN) The number of entries contained in each array (the same for row_indices,
 2262 col_indices, and values).

2263 dup (IN) An associative and commutative binary operator to apply when duplicate
 2264 values for the same location are present in the input arrays. All three domains of
 2265 dup must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If dup is GrB_NULL,
 2266 then duplicate locations will result in an error.

2267 Return Values

2268 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 2269 blocking mode, this indicates that the API checks for the input
 2270 arguments passed successfully. Either way, output matrix C is
 2271 ready to be used in the next method of the sequence.

2272 **GrB_PANIC** Unknown internal error.

2273 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
 2274 opaque GraphBLAS objects (input or output) is in an invalid
 2275 state caused by a previous execution error. Call **GrB_error()** to
 2276 access any error messages generated by the implementation.

2277 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2278 **GrB_UNINITIALIZED_OBJECT** Either **C** has not been initialized by a call to any matrix construc-
 2279 tor, or **dup** has not been initialized by a call to **GrB_BinaryOp_new**.

2280 **GrB_NULL_POINTER** **row_indices**, **col_indices** or **values** pointer is **NULL**.

2281 **GrB_INDEX_OUT_OF_BOUNDS** A value in **row_indices** or **col_indices** is outside the allowed range
 2282 for **C**.

2283 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
 2284 not all the same, or the domains of **values** and **C** are incompatible
 2285 with each other or D_{dup} .

2286 **GrB_OUTPUT_NOT_EMPTY** Output matrix **C** already contains valid tuples (elements). In
 2287 other words, **GrB_Matrix_nvals(C)** returns a positive value.

2288 **GrB_INVALID_VALUE** **indices** contains a duplicate location and **dup** is **GrB_NULL**.

2289 Description

2290 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
 2291 only differs from **C** in its domain; otherwise, $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$.

2292 Each tuple $\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the
 2293 output in the form of

$$2294 \quad \tilde{\mathbf{C}}(\text{row_indices}[k], \text{col_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2295 If multiple values for the same location are present in the input arrays and **dup** is not **GrB_NULL**,
 2296 **dup** is used to reduce the values before assignment into $\tilde{\mathbf{C}}$ as follows:

$$2297 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row_indices}[k]=i \wedge \text{col_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2298 where \oplus is the **dup** binary operator. Finally, the resulting $\tilde{\mathbf{C}}$ is copied into **C** via typecasting its
 2299 values to $\mathbf{D}(\mathbf{C})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

2300 The nonopaque input arrays **row_indices**, **col_indices**, and **values** must be at least as large as **n**.

2301 It is an error to call this function on an output object with existing elements. In other words,
 2302 **GrB_Matrix_nvals(C)** should evaluate to zero prior to calling this function.

2303 After GrB_Matrix_build returns, it is safe for a programmer to modify or delete the arrays row_indices,
2304 col_indices, or values.

2305 4.2.5.10 Matrix_setElement: Set a single element in matrix

2306 Set one element of a matrix to a given value.

2307 C Syntax

```
2308         // scalar value
2309         GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2310                                     <type>           val,
2311                                     GrB_Index          row_index,
2312                                     GrB_Index          col_index);
2313
2314         // GraphBLAS scalar
2315         GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2316                                     const GrB_Scalar    s,
2317                                     GrB_Index          row_index,
2318                                     GrB_Index          col_index);
```

2319 Parameters

2320 C (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.

2321 val or s (IN) Scalar to assign. Its domain (type) must be compatible with the domain of
2322 C.

2323 row_index (IN) Row index of element to be assigned

2324 col_index (IN) Column index of element to be assigned

2325 Return Values

2326 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2327 blocking mode, this indicates that the compatibility tests on in-
2328 dex/dimensions and domains for the input arguments passed suc-
2329 cessfully. Either way, the output matrix C is ready to be used in
2330 the next method of the sequence.

2331 GrB_PANIC Unknown internal error.

2332 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2333 GraphBLAS objects (input or output) is in an invalid state caused

2334 by a previous execution error. Call `GrB_error()` to access any error
 2335 messages generated by the implementation.

2336 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2337 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been
 2338 initialized by a call to a respective constructor.

2339 **GrB_INVALID_INDEX** `row_index` or `col_index` is outside the allowable range (i.e., not less
 2340 than `nrows(C)` or `ncols(C)`, respectively).

2341 **GrB_DOMAIN_MISMATCH** The domains of the matrix and the scalar are incompatible.

2342 Description

2343 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or
 2344 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from
 2345 one domain can be cast to values in the other domain as per the rules of the C language. In
 2346 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
 2347 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
 2348 `GrB_Matrix_setElement` ends and the domain mismatch error listed above is returned.

2349 Then, both index parameters are checked for valid values where following conditions must hold:

$$2350 \begin{aligned} 0 &\leq \text{row_index} < \text{nrows}(C), \\ 0 &\leq \text{col_index} < \text{ncols}(C) \end{aligned}$$

2351 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid
 2352 index error listed above is returned.

2353 We are now ready to carry out the assignment; that is:

$$2354 C(\text{row_index}, \text{col_index}) = \begin{cases} \mathbf{L}(s), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

2355 In the case of a transparent scalar or if **L(s)** is not empty, then a value will be stored at the
 2356 specified location in **C**, overwriting any value that may have been stored there before. In the case
 2357 of a GraphBLAS scalar and if **L(s)** is empty, then any value stored at the specified location in **C**
 2358 will be removed.

2359 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new contents
 2360 of **C** is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method exits with
 2361 return value **GrB_SUCCESS** and the new content of vector **C** is as defined above but may not be
 2362 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2363 4.2.5.11 `Matrix_removeElement`: Remove an element from a matrix

2364 Remove (annihilate) one stored element from a matrix.

2365 C Syntax

```
2366      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,  
2367                                         GrB_Index   row_index,  
2368                                         GrB_Index   col_index);
```

2369 Parameters

2370 C (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2371 row_index (IN) Row index of element to be removed

2372 col_index (IN) Column index of element to be removed

2373 Return Values

2374 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2375 blocking mode, this indicates that the compatibility tests on in-
2376 dex/dimensions and domains for the input arguments passed suc-
2377 cessfully. Either way, the output matrix C is ready to be used in
2378 the next method of the sequence.

2379 GrB_PANIC Unknown internal error.

2380 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2381 GraphBLAS objects (input or output) is in an invalid state caused
2382 by a previous execution error. Call GrB_error() to access any error
2383 messages generated by the implementation.

2384 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2385 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to
2386 any matrix constructor.

2387 GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e., not less
2388 than **nrows**(C) or **ncols**(C), respectively).

2389 Description

2390 First, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2391 \quad & 0 \leq \text{row_index} < \mathbf{nrows}(C), \\ & 0 \leq \text{col_index} < \mathbf{ncols}(C) \end{aligned}$$

2392 If either of these conditions is violated, execution of GrB_Matrix_removeElement ends and the
2393 invalid index error listed above is returned.

2394 We are now ready to carry out the removal of a value that may be stored at the location specified by
 2395 (row_index, col_index). If a value does not exist at the specified location in C, no error is reported
 2396 and the operation has no effect on the state of C. In either case, the following will be true on return
 2397 from this method: (row_index, col_index) \notin ind(C)

2398 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
 2399 of C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
 2400 return value GrB_SUCCESS and the new content of vector C is as defined above but may not be
 2401 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2402 **4.2.5.12 Matrix_extractElement: Extract a single element from a matrix**

2403 Extract one element of a matrix into a scalar.

2404 **C Syntax**

```
2405         // scalar value
2406         GrB_Info GrB_Matrix_extractElement(<type>          *val,
2407                                         const GrB_Matrix  A,
2408                                         GrB_Index         row_index,
2409                                         GrB_Index         col_index);
2410
2411         // GraphBLAS scalar
2412         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2413                                         const GrB_Matrix  A,
2414                                         GrB_Index         row_index,
2415                                         GrB_Index         col_index);
2416
```

2417 **Parameters**

2418 val or s (INOUT) An existing scalar whose domain is compatible with the domain of matrix
 2419 A. On successful return, this scalar holds the result of the extract. Any previous
 2420 value stored in val or s is overwritten.

2421 A (IN) The GraphBLAS matrix from which an element is extracted.

2422 row_index (IN) The row index of location in A to extract.

2423 col_index (IN) The column index of location in A to extract.

2424 **Return Values**

2425 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
 2426 cessfully. This indicates that the compatibility tests on dimensions

2427 and domains for the input arguments passed successfully, and the
 2428 output scalar, **val** or **s**, has been computed and is ready to be used
 2429 in the next method of the sequence.

2430 **GrB_NO_VALUE** When using the transparent scalar, **val**, this is returned when there
 2431 is no stored value at specified location.

2432 **GrB_PANIC** Unknown internal error.

2433 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 2434 GraphBLAS objects (input or output) is in an invalid state caused
 2435 by a previous execution error. Call **GrB_error()** to access any error
 2436 messages generated by the implementation.

2437 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2438 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by
 2439 a call to a corresponding constructor.

2440 **GrB_NULL_POINTER** **val** pointer is NULL.

2441 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e. less than
 2442 zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-
 2443 tively).

2444 **GrB_DOMAIN_MISMATCH** The domains of the matrix and scalar are incompatible.

2445 Description

2446 First, the scalar and input matrix are tested for domain compatibility as follows: **D(val)** or **D(s)**
 2447 must be compatible with **D(A)**. Two domains are compatible with each other if values from
 2448 one domain can be cast to values in the other domain as per the rules of the C language. In
 2449 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
 2450 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
 2451 **GrB_Matrix_extractElement** ends and the domain mismatch error listed above is returned.

2452 Then, both index parameters are checked for valid values where following conditions must hold:

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

2454 If either condition is violated, execution of **GrB_Matrix_extractElement** ends and the invalid index
 2455 error listed above is returned.

2456 We are now ready to carry out the extract into the output scalar; that is,

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

2458 If $(\text{row_index}, \text{col_index}) \in \mathbf{ind}(\mathbf{A})$, then the corresponding value from \mathbf{A} is copied into \mathbf{s} or \mathbf{val}
 2459 with casting as necessary. If $(\text{row_index}, \text{col_index}) \notin \mathbf{ind}(\mathbf{A})$, then one of the follow occurs
 2460 depending on output scalar type:

- 2461 • The GraphBLAS scalar, \mathbf{s} , is cleared and GrB_SUCCESS is returned.
- 2462 • The non-opaque scalar, \mathbf{val} , is unchanged, and GrB_NO_VALUE is returned.

2463 When using the non-opaque scalar variant (\mathbf{val}) in both GrB_BLOCKING mode GrB_NONBLOCKING
 2464 mode, the new contents of \mathbf{val} are as defined above if the method exits with return value GrB_SUCCESS
 2465 or GrB_NO_VALUE .

2466 When using the GraphBLAS scalar variant (\mathbf{s}) with a GrB_SUCCESS return value, the method
 2467 exits and the new contents of \mathbf{s} is as defined above and fully computed in GrB_BLOCKING mode.
 2468 In GrB_NONBLOCKING mode, the new contents of \mathbf{s} is as defined above but may not be fully
 2469 computed; however, it can be used in the next GraphBLAS method call in a sequence.

2470 4.2.5.13 Matrix_extractTuples: Extract tuples from a matrix

2471 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

2472 C Syntax

```

2473     GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2474                                     GrB_Index      *col_indices,
2475                                     <type>          *values,
2476                                     GrB_Index      *n,
2477                                     const GrB_Matrix A);

```

2478 Parameters

2479 **row_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the
 2480 row indices.

2481 **col_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the
 2482 column indices.

2483 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
 2484 the stored values whose type is compatible with $\mathbf{D}(\mathbf{A})$.

2485 **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,
 2486 **row_indices**, and **col_indices** arrays can hold. Upon return, it will contain the
 2487 number of values written to the arrays.

2488 **A** (IN) An existing GraphBLAS matrix.

2489 Return Values

2490	GrB_SUCCESS	In blocking or non-blocking mode, the operation completed successfully. This indicates that the compatibility tests on the input
2491		argument passed successfully, and the output arrays, indices and
2492		values , have been computed.
2493		
2494	GrB_PANIC	Unknown internal error.
2495	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque
2496		GraphBLAS objects (input or output) is in an invalid state caused
2497		by a previous execution error. Call GrB_error() to access any error
2498		messages generated by the implementation.
2499	GrB_OUT_OF_MEMORY	Not enough memory available for operation.
2500	GrB_INSUFFICIENT_SPACE	Not enough space in row_indices , col_indices , and values (as indi-
2501		cated by the n parameter) to hold all of the tuples that will be
2502		extracted.
2503	GrB_UNINITIALIZED_OBJECT	The GraphBLAS matrix, A , has not been initialized by a call to
2504		any matrix constructor.
2505	GrB_NULL_POINTER	row_indices , col_indices , values or n pointer is NULL .
2506	GrB_DOMAIN_MISMATCH	The domains of the A matrix and values array are incompatible
2507		with one another.

2508 Description

2509 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with
2510 those tuples are placed in the **values** array, the column indices are placed in the **col_indices** array,
2511 and the row indices are placed in the **row_indices** array. These output arrays are pre-allocated by
2512 the user before calling this function such that each output array has enough space to hold at least
2513 **GrB_Matrix_nvals(A)** elements.

2514 Upon return of this function, a pair of $\{\text{row_indices}[k], \text{col_indices}[k]\}$ are unique for every valid
2515 k , but they are not required to be sorted in any particular order. Each tuple (i, j, A_{ij}) in **A** is
2516 unzipped and copied into a distinct k th location in output vectors:

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

2517 where $0 \leq k < \text{GrB_Matrix_nvals}(v)$. No gaps in output vectors are allowed; that is, if **row_indices**[k],
2518 **col_indices**[k] and **values**[k] exist upon return, so does **row_indices**[j], **col_indices**[j] and **values**[j] for
2519 all j such that $0 \leq j < k$.

2520 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,
2521 then a **GrB_INSUFFICIENT_SPACE** error is returned since it is undefined which subset of values
2522 would be extracted.

2523 In both GrB_BLOCKING mode GrB_NONBLOCKING mode if the method exits with return value
2524 GrB_SUCCESS, the new contents of the arrays row_indices, col_indices and values are as defined
2525 above.

2526 **4.2.5.14 Matrix_exportHint: Provide a hint as to which storage format might be most**
2527 **efficient for exporting a matrix**

2528 C Syntax

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

2529 Parameters

2530 hint (OUT) Pointer to a value of type GrB_Format.

2531 A (IN) A GraphBLAS matrix object.

2532 Return Values

2533 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2534 cessfully and the value of hint has been set.

2535 GrB_PANIC Unknown internal error.

2536 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
2537 opaque GraphBLAS objects (input or output) is in an invalid
2538 state caused by a previous execution error. Call GrB_error() to
2539 access any error messages generated by the implementation.

2540 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2541 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2542 any matrix constructor.

2543 GrB_NULL_POINTER hint is NULL.

2544 GrB_NO_VALUE If the implementation does not have a preferred format, it may
2545 return the value GrB_NO_VALUE.

2546 Description

2547 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for
2548 exporting the matrix A. GraphBLAS implementations might return the current storage format of
2549 the matrix, or the format to which it could most efficiently be exported. However, implementations
2550 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is
2551 free to refuse to provide a format hint, returning GrB_NO_VALUE.

2552 **4.2.5.15 Matrix_exportSize:** Return the array sizes necessary to export a GraphBLAS
 2553 matrix object

2554 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);

```

2555 Parameters

2556 **n_indptr** (OUT) Pointer to a value of type GrB_Index.

2557 **n_indices** (OUT) Pointer to a value of type GrB_Index.

2558 **n_values** (OUT) Pointer to a value of type GrB_Index.

2559 **format** (IN) a value indicating the format in which the matrix will be exported, as defined
 2560 in Section 3.5.3.1.

2561 **A** (IN) A GraphBLAS matrix object.

2562 Return Values

2563 **GrB_SUCCESS** In blocking mode or non-blocking mode, the operation completed successfully. This indicates that the API checks for the
 2564 input arguments passed successfully, and the number of elements
 2565 necessary for the export buffers have been written to **n_indptr**,
 2566 **n_indices**, and **n_values**, respectively.

2567 **GrB_PANIC** Unknown internal error.

2568 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
 2569 opaque GraphBLAS objects (input or output) is in an invalid
 2570 state caused by a previous execution error. Call **GrB_error()** to
 2571 access any error messages generated by the implementation.

2572 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2573 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS Matrix, **A**, has not been initialized by a call to
 2574 any matrix constructor.

2575 **GrB_NULL_POINTER** **n_indptr**, **n_indices**, or **n_values** is NULL.

2576

2577 **Description**

2578 Given a matrix **A**, returns the required capacities of arrays **values**, **indptr**, and **indices** necessary to
2579 export the matrix in the format specified by **format**. The output values **n_values**, **n_indptr**, and
2580 **indices** will contain the corresponding sizes of the arrays (in number of elements) that must be
2581 allocated to hold the exported matrix. The argument **format** can be chosen arbitrarily by the user
2582 as one of the values defined in Section 3.5.3.1.

2583 **4.2.5.16 Matrix_export: Export a GraphBLAS matrix to a pre-defined format**

2584 **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);
```

2585 **Parameters**

2586 **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-
2587 dices, depending on the value of **format**. It must be large enough to hold at
2588 least **n_indptr** elements of type **GrB_Index**, where **n_indices** was returned from
2589 **GrB_Matrix_exportSize()** method.

2590 **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements
2591 in **values**, depending on the value of **format**. It must be large enough to hold at
2592 least **n_indices** elements of type **GrB_Index**, where **n_indices** was returned from
2593 **GrB_Matrix_exportSize()** method.

2594 **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-
2595 ment must match the type of the values stored in **A**. It must be large enough
2596 to hold at least **n_values** elements of that type, where **n_values** was returned from
2597 **GrB_Matrix_exportSize**.

2598 **n_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**
2599 array can hold. Upon return, it will contain the number of elements written to the
2600 array.

2601 **n_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**
2602 array can hold. Upon return, it will contain the number of elements written to the
2603 array.

2604 **n_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**
2605 array can hold. Upon return, it will contain the number of elements written to the
2606 array.

2607 **format** (IN) a value indicating the format in which the matrix will be exported, as defined
2608 in Section 3.5.3.1.

2609 **A** (IN) A GraphBLAS matrix object.

2610 **Return Values**

2611 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2612 cessfully. This indicates that the compatibility tests on the input
2613 argument passed successfully, and the output arrays, **indptr**, **in-**
2614 **indices** and **values**, have been computed.

2615 **GrB_PANIC** Unknown internal error.

2616 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2617 opaque GraphBLAS objects (input or output) is in an invalid
2618 state caused by a previous execution error. Call **GrB_error()** to
2619 access any error messages generated by the implementation.

2620 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2621 **GrB_INSUFFICIENT_SPACE** Not enough space in **indptr**, **indices**, and/or **values** (as indicated
2622 by the corresponding **n_*** parameter) to hold all of the corre-
2623 sponding elements that will be extacted.

2624 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2625 any matrix constructor.

2626 **GrB_NULL_POINTER** **indptr**, **indices**, **values** **n_indptr**, **n_indices**, **n_values** pointer is
2627 NULL.

2628 **GrB_DOMAIN_MISMATCH** The domain of **A** does not match with the type of **values**.

2629 **Description**

2630 Given a matrix **A**, this method exports the contents of the matrix into one of the pre-defined
2631 **GrB_Format** formats from Section 3.5.3.1. The user-allocated arrays pointed to by **indptr**, **indices**,
2632 and **values** must be at least large enough to hold the corresponding number of elements returned
2633 by calling **GrB_Matrix_exportSize**. The value of **format** can be chosen arbitrarily, but a call to
2634 **GrB_Matrix_exportHint** may suggest a format that results in the most efficient export. Details
2635 of the contents of **indptr**, **indices**, and **values** corresponding to each supported format is given in
2636 Appendix B.

2637 4.2.5.17 Matrix_import: Import a matrix into a GraphBLAS object

2638 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);
```

2639 Parameters

2640 A (INOUT) On a successful return, contains a handle to the newly created Graph-
2641 BLAS matrix.

2642 d (IN) The type corresponding to the domain of the matrix being created. Can be
2643 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
2644 GraphBLAS type.

2645 nrows (IN) Integer value holding the number of rows in the matrix.

2646 ncols (IN) Integer value holding the number of columns in the matrix.

2647 indptr (IN) Pointer to an array of row or column offsets, or row indices, depending on the
2648 value of format.

2649 indices (IN) Pointer to an array row or column indices of the elements in values, depending
2650 on the value of format.

2651 values (IN) Pointer to an array of values. Type must match the type of d.

2652 n_indptr (IN) Integer value holding the number of elements in the array pointed to by indptr.

2653 n_indices (IN) Integer value holding the number of elements in the array pointed to by indices.

2654 n_values (IN) Integer value holding the number of elements in the array pointed to by values.

2655 format (IN) a value indicating the format of the matrix being imported, as defined in
2656 Section 3.5.3.1.

2657 Return Values

2658	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
2659		blocking mode, this indicates that the API checks for the input
2660		arguments passed successfully and the input arrays have been
2661		consumed. Either way, output matrix A is ready to be used in
2662		the next method of the sequence.
2663	GrB_PANIC	Unknown internal error.
2664	GrB_OUT_OF_MEMORY	Not enough memory available for operation.
2665	GrB_UNINITIALIZED_OBJECT	The GrB_Type object has not been initialized by a call to GrB_Type_new
2666		(needed for user-defined types).
2667	GrB_NULL_POINTER	A , indptr , indices or values pointer is NULL.
2668	GrB_INDEX_OUT_OF_BOUNDS	A value in indptr or indices is outside the allowed range for indices
2669		in A and or the size of values , n_values , depending on the value
2670		of format .
2671	GrB_INVALID_VALUE	nrows or ncols is zero or outside the range of the type GrB_Index.
2672	GrB_DOMAIN_MISMATCH	The domain given in parameter d does not match the element
2673		type of values .

2674 Description

2675 Creates a new matrix **A** of domain **D(d)** and dimension **nrows** \times **ncols**. The new GraphBLAS
2676 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.
2677 The method returns a handle to the new matrix in **A**. The structure of the data being imported is
2678 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of
2679 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2680 It is not an error to call this method more than once on the same output matrix; however, the
2681 handle to the previously created object will be overwritten.

2682 4.2.5.18 Matrix_serializeSize: Compute the serialize buffer size

2683 Compute the buffer size (in bytes) necessary to serialize a GrB_Matrix using GrB_Matrix_serialize.

2684 C Syntax

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

2685 Parameters

2686 size (OUT) Pointer to GrB_Index value where size in bytes of serialized object will be
2687 written.

2688 A (IN) A GraphBLAS matrix object.

2689 Return Values

2690 GrB_SUCCESS The operation completed successfully and the value pointed to
2691 by *size has been computed and is ready to use.

2692 GrB_PANIC Unknown internal error.

2693 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2694 GrB_NULL_POINTER size is NULL.

2695 Description

2696 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object A.
2697 Users may then allocate a buffer of size bytes to pass as a parameter to GrB_Matrix_serialize.

2698 4.2.5.19 Matrix_serialize: Serialize a GraphBLAS matrix.

2699 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2700 C Syntax

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

2701 Parameters

2702 serialized_data (INOUT) Pointer to the preallocated buffer where the serialized matrix will be
2703 written.

2704 serialized_size (INOUT) On input, the size in bytes of the buffer pointed to by serialized_data.
2705 On output, the number of bytes written to serialized_data.

2706 A (IN) A GraphBLAS matrix object.

2707 Return Values

2708 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2709 cessfully. This indicates that the compatibility tests on the in-
2710 put argument passed successfully, and the output buffer serial-
2711 ized_data and serialized_size, have been computed and are ready
2712 to use.

2713 GrB_PANIC Unknown internal error.

2714 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
2715 opaque GraphBLAS objects (input or output) is in an invalid
2716 state caused by a previous execution error. Call GrB_error() to
2717 access any error messages generated by the implementation.

2718 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2719 GrB_NULL_POINTER serialized_data or serialize_size is NULL.

2720 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2721 any matrix constructor.

2722 GrB_INSUFFICIENT_SPACE The size of the buffer serialized_data (provided as an input seri-
2723 alized_size) was not large enough.

2724 Description

2725 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,
2726 the size of the buffer pointed to by serialized_data, provided as an input by serialized_size, must
2727 be of at least the number of bytes returned from GrB_Matrix_serializeSize. The actual size of the
2728 serialized matrix written to serialized_data is provided upon completion as an output written to
2729 serialized_size.

2730 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created
2731 with one library implementation is not necessarily valid for deserialization with another implemen-
2732 tation.

2733 4.2.5.20 Matrix_deserialize: Deserialize a GraphBLAS matrix.

2734 Construct a new GraphBLAS matrix from a serialized object.

2735 C Syntax

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

2736 Parameters

2737 A (INOUT) On a successful return, contains a handle to the newly created Graph-
2738 BLAS matrix.

2739 d (IN) the type of the matrix that was serialized in `serialized_data`.

2740 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2741 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

2742 Return Values

2743 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2744 blocking mode, this indicates that the API checks for the input
2745 arguments passed successfully. Either way, output matrix A is
2746 ready to be used in the next method of the sequence.

2747 GrB_PANIC Unknown internal error.

2748 GrB_INVALID_OBJECT This is returned if `serialized_data` is invalid or corrupted.

2749 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2750 GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to `GrB_Type_new`
2751 (needed for user-defined types).

2752 GrB_NULL_POINTER `serialized_data` or A is NULL.

2753 GrB_DOMAIN_MISMATCH The type given in d does not match the type of the matrix
2754 serialized in `serialized_data`.

2755 Description

2756 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object
2757 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The
2758 domain of the matrix is given as an input in d, which must match the domain of the matrix serialized
2759 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2760 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix
2761 serialized in one library implementation can be deserialized by another.

2762 It is not an error to call this method more than once on the same output matrix; however, the
2763 handle to the previously created object will be overwritten.

2764 4.2.6 Descriptor methods

2765 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-
2766 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

2767 **4.2.6.1 Descriptor_new: Create new descriptor**

2768 Creates a new (empty or default) descriptor.

2769 **C Syntax**

```
2770      GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);
```

2771 **Parameters**

2772 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2773 descriptor.

2774 **Return Value**

2775 GrB_SUCCESS The method completed successfully.

2776 GrB_PANIC unknown internal error.

2777 GrB_OUT_OF_MEMORY not enough memory available for operation.

2778 GrB_NULL_POINTER desc pointer is NULL.

2779 **Description**

2780 Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can
2781 be populated by calls to Descriptor_set.

2782 It is not an error to call this method more than once on the same variable; however, the handle to
2783 the previously created object will be overwritten.

2784 **4.2.6.2 Descriptor_set: Set content of descriptor**

2785 Sets the content for a field for an existing descriptor.

2786 **C Syntax**

```
2787      GrB_Info GrB_Descriptor_set(GrB_Descriptor      desc,  
2788                                GrB_Desc_Field      field,  
2789                                GrB_Desc_Value      val);
```

2790 Parameters

2791 desc (IN) An existing GraphBLAS descriptor to be modified.

2792 field (IN) The field being set.

2793 val (IN) New value for the field being set.

2794 Return Values

2795 GrB_SUCCESS operation completed successfully.

2796 GrB_PANIC unknown internal error.

2797 GrB_OUT_OF_MEMORY not enough memory available for operation.

2798 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to new.

2799 GrB_INVALID_VALUE invalid value set on the field, or invalid field.

2800 Description

2801 For a given descriptor, the GrB_Descriptor_set method can be called for each field in the descriptor
2802 to set the value associated with that field. Valid values for the field parameter include the following:

2803 GrB_OUTP refers to the output parameter (result) of the operation.

2804 GrB_MASK refers to the mask parameter of the operation.

2805 GrB_INP0 refers to the first input parameters of the operation (matrices and vectors).

2806 GrB_INP1 refers to the second input parameters of the operation (matrices and vectors).

2807 Valid values for the val parameter are:

2808 GrB_STRUCTURE Use only the structure of the stored values of the corresponding mask
2809 (GrB_MASK) parameter.

2810 GrB_COMP Use the complement of the corresponding mask (GrB_MASK) param-
2811 eter. When combined with GrB_STRUCTURE, the complement of the
2812 structure of the mask is used without evaluating the values stored.

2813 GrB_TRAN Use the transpose of the corresponding matrix parameter (valid for input
2814 matrix parameters only).

2815 GrB_REPLACE When assigning the masked values to the output matrix or vector, clear
2816 the matrix first (or clear the non-masked entries). The default behavior
2817 is to leave non-masked locations unchanged. Valid for the GrB_OUTP
2818 parameter only.

2819 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of GrB_MASK,
2820 multiple values can be set and all will apply (for example, both GrB_COMP and GrB_STRUCTURE).
2821 A value for a given field may be set multiple times but will have no additional effect. Fields that
2822 have no values set result in their default behavior, as defined in Section 3.6.

2823 4.2.7 free: Destroy an object and release its resources

2824 Destroys a previously created GraphBLAS object and releases any resources associated with the
2825 object.

2826 C Syntax

```
2827 GrB_Info GrB_free(<GrB_Object> *obj);
```

2828 Parameters

2829 obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have
2830 been created by an explicit call to a GraphBLAS constructor. It can be any of the
2831 opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,
2832 binary op, unary op, or type. On successful completion of GrB_free, obj behaves
2833 as an uninitialized object.

2834 Return Values

2835 GrB_SUCCESS operation completed successfully

2836 GrB_PANIC unknown internal error. If this return value is encountered when
2837 in nonblocking mode, the error responsible for the panic condition
2838 could be from any method involved in the computation of the input
2839 object. The GrB_error() method should be called for additional
2840 information.

2841 Description

2842 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime
2843 system. A call to GrB_free frees those resources so they are available for use by other GraphBLAS
2844 objects.

2845 The parameter passed into GrB_free is a handle referencing a GraphBLAS opaque object of a data
2846 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-
2847 structor. The behavior of a program that calls GrB_free on a pre-defined object is implementation
2848 defined.

2849 After the `GrB_free` method returns, the object referenced by the input handle is destroyed and the
2850 handle has the value `GrB_INVALID_HANDLE`. The handle can be used in subsequent GraphBLAS
2851 methods but only after the handle has been reinitialized with a call the the appropriate `_new` or
2852 `_dup` method.

2853 Note that unlike other GraphBLAS methods, calling `GrB_free` with an object with an invalid handle
2854 is legal. The system may attempt to free resources that might be associated with that object, if
2855 possible, and return normally.

2856 When using `GrB_free` it is possible to create a dangling reference to an object. This would occur
2857 when a handle is assigned to a second variable of the same opaque type. This creates two handles
2858 that reference the same object. If `GrB_free` is called with one of the variables, the object is destroyed
2859 and the handle associated with the other variable no longer references a valid object. This is not an
2860 error condition that the implementation of the GraphBLAS API can be expected to catch, hence
2861 programmers must take care to prevent this situation from occurring.

2862 4.2.8 wait: Return once an object is either *complete* or *materialized*

2863 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2864 C Syntax

```
2865      GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);
```

2866 Parameters

2867 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an
2868 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2869 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2870 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another
2871 thread (completion) or all computing to produce `obj` by all GraphBLAS operations
2872 in its sequence have finished (materialization).

2873 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the
2874 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or
2875 `GrB_MATERIALIZE`.

2876 Return values

2877 `GrB_SUCCESS` operation completed successfully.

2878 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-
2879 pletion of pending operations.

2880 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion
2881 of pending operations.

2882 GrB_UNINITIALIZED_OBJECT object has not been initialized by a call to the respective *_new,
2883 or other constructor, method.

2884 GrB_PANIC unknown internal error.

2885 GrB_INVALID_VALUE method called with a GrB_WaitMode other than GrB_COMPLETE
2886 GrB_MATERIALIZE.

2887 Description

2888 On successful return from GrB_wait(), the input object, `obj` is in one of two states depending on
2889 the mode of GrB_wait:

- 2890 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program
2891 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another
2892 thread. This result occurs when the mode parameter is set to GrB_COMPLETE.
- 2893 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on
2894 behalf of `obj` and error information is available. This result occurs when the mode parameter
2895 is set to GrB_MATERIALIZE.

2896 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,
2897 GrB_wait(`obj`,mode) has no effect when called in blocking mode.

2898 In non-blocking mode, the status of any pending method calls, other than those associated with pro-
2899 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to GrB_wait(`obj`,mode).
2900 Methods in the sequence for `obj`, however, most likely would be impacted by a call to GrB_wait(`obj`,mode);
2901 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be
2902 finished prior to the return from GrB_wait(`obj`,mode).

2903 4.2.9 error: Retrieve an error string

2904 Retrieve an error-message about any errors encountered during the processing associated with an
2905 object.

2906 C Syntax

```
2907 GrB_Info GrB_error(const char      **error,  
2908                   const GrB_Object  obj);
```

2909 Parameters

2910 error (OUT) A pointer to a null-terminated string. The contents of the string are im-
2911 plementation defined.

2912 obj (IN) An existing GraphBLAS object. The object must have been created by an
2913 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2914 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2915 or type.

2916 **Return value**

2917 GrB_SUCCESS operation completed successfully.

2918 GrB_UNINITIALIZED_OBJECT object has not been initialized by a call to the respective *_new,
2919 or other constructor, method.

2920 GrB_PANIC unknown internal error.

2921 **Description**

2922 This method retrieves a message related to any errors that were encountered during the last Graph-
2923 BLAS method that had the opaque GraphBLAS object, obj, as an OUT or INOUT parameter.
2924 The function returns a pointer to a null-terminated string and the contents of that string are
2925 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error
2926 string. The string that is returned is owned by obj and will be valid until the next time obj is
2927 used as an OUT or INOUT parameter or the object is freed by a call to GrB_free(obj). This is a
2928 thread-safe function. It can be safely called by multiple threads for the same object in a race-free
2929 program.

2930 **4.3 GraphBLAS operations**

2931 The GraphBLAS operations are defined in the GraphBLAS math specification and summarized in
2932 Table 4.1. In addition to methods that implement these fundamental GraphBLAS operations, we
2933 support a number of variants that have been found to be especially useful in algorithm development.
2934 A flowchart of the overall behavior of a GraphBLAS operation is shown in Figure 4.1.

2935 **Domains and Casting**

2936 A GraphBLAS operation is only valid when the domains of the GraphBLAS objects are mathemat-
2937 ically consistent. The C programming language defines implicit casts between built-in data types.
2938 For example, floats, doubles, and ints can be freely mixed according to the rules defined for implicit
2939 casts. It is the responsibility of the user to assure that these casts are appropriate for the algorithm
2940 in question. For example, a cast to int implies truncation of a floating point type. Depending on
2941 the operation, this truncation error could lead to erroneous results. Furthermore, casting a wider
2942 type onto a narrower type can lead to overflow errors. The GraphBLAS operations do not attempt
2943 to protect a user from these sorts of errors.

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	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A}^T$
kronecker	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

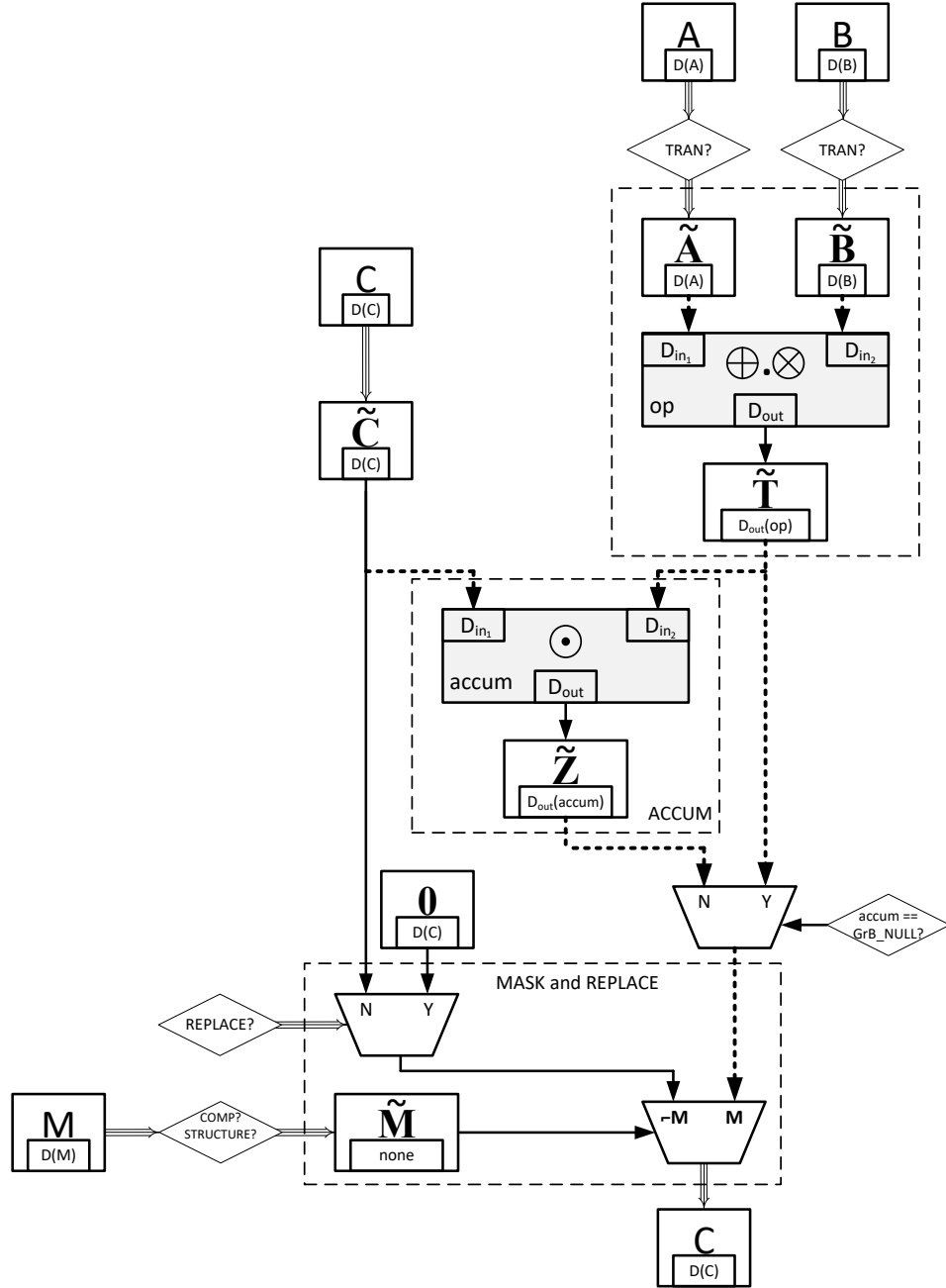


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows (\Rightarrow) denote where “as if copy” takes place (including both collections and descriptor settings). The bold, dotted arrows indicate where casting may occur between different domains.

2944 When user-defined types are involved, however, GraphBLAS requires strict equivalence between
 2945 types and no casting is supported. If GraphBLAS detects these mismatches, it will return a
 2946 domain mismatch error.

2947 Dimensions and Transposes

2948 GraphBLAS operations also make assumptions about the numbers of dimensions and the sizes of
 2949 vectors and matrices in an operation. An operation will test these sizes and report an error if they
 2950 are not *shape compatible*. For example, when multiplying two matrices, $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, the number
 2951 of rows of \mathbf{C} must equal the number of rows of \mathbf{A} , the number of columns of \mathbf{A} must match the
 2952 number of rows of \mathbf{B} , and the number of columns of \mathbf{C} must match the number of columns of \mathbf{B} .
 2953 This is the behavior expected given the mathematical definition of the operations.

2954 For most of the GraphBLAS operations involving matrices, an optional descriptor can modify the
 2955 matrix associated with an input GraphBLAS matrix object. For example, if an input matrix is an
 2956 argument to a GraphBLAS operation and the associated descriptor indicates the transpose option,
 2957 then the operation occurs as if on the transposed matrix. In this case, the relationships between
 2958 the sizes in each dimension shift in the mathematically expected way.

2959 Masks: Structure-only, Complement, and Replace

2960 When a GraphBLAS operation supports the use of an optional mask, that mask is specified through
 2961 a GraphBLAS vector (for one-dimensional masks) or a GraphBLAS matrix (for two-dimensional
 2962 masks). When a mask is used and the `GrB_STRUCTURE` descriptor value is not set, it is applied
 2963 to the result from the operation wherever the stored values in the mask evaluate to true. If the
 2964 `GrB_STRUCTURE` descriptor is set, the mask is applied to the result from the operation wherever the
 2965 mask as a stored value (regardless of that value). Wherever the mask is applied, the result from
 2966 the operation is either assigned to the provided output matrix/vector or, if a binary accumulation
 2967 operation is provided, the result is accumulated into the corresponding elements of the provided
 2968 output matrix/vector.

2969 Given a GraphBLAS vector $\mathbf{v} = \langle D, N, \{(i, v_i)\} \rangle$, a one-dimensional mask is derived for use in the
 2970 operation as follows:

$$2971 \quad \mathbf{m} = \begin{cases} \langle N, \{\mathbf{ind}(\mathbf{v})\} \rangle, & \text{if } \text{GrB_STRUCTURE} \text{ is specified,} \\ \langle N, \{i : (\text{bool})v_i = \text{true}\} \rangle, & \text{otherwise} \end{cases}$$

2972 where $(\text{bool})v_i$ denotes casting the value v_i to a Boolean value (true or false). Likewise, given a
 2973 GraphBLAS matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$, a two-dimensional mask is derived for use in the
 2974 operation as follows:

$$2975 \quad \mathbf{M} = \begin{cases} \langle M, N, \{\mathbf{ind}(\mathbf{A})\} \rangle, & \text{if } \text{GrB_STRUCTURE} \text{ is specified,} \\ \langle M, N, \{(i, j) : (\text{bool})A_{ij} = \text{true}\} \rangle, & \text{otherwise} \end{cases}$$

2976 where $(\text{bool})A_{ij}$ denotes casting the value A_{ij} to a Boolean value. (true or false)

2977 In both the one- and two-dimensional cases, the mask may also have a subsequent complement
 2978 operation applied (*Section 3.5.4*) as specified in the descriptor, before a final mask is generated for
 2979 use in the operation.

2980 When the descriptor of an operation with a mask has specified that the `GrB_REPLACE` value is
 2981 to be applied to the output (`GrB_OUTP`), then anywhere the mask is not `true`, the corresponding
 2982 location in the output is cleared.

2983 Invalid and uninitialized objects

2984 Upon entering a GraphBLAS operation, the first step is a check that all objects are valid and ini-
 2985 tialized. (Optional parameters can be set to `GrB_NULL`, which always counts as a valid object.) An
 2986 invalid object is one that could not be computed due to a previous execution error. An uninitialized
 2987 object is one that has not yet been created by a corresponding `new` or `dup` method. Appropriate
 2988 error codes are returned if an object is not initialized (`GrB_UNINITIALIZED_OBJECT`) or invalid
 2989 (`GrB_INVALID_OBJECT`).

2990 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-
 2991 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at
 2992 the point of their declaration, as shown in the following examples:

```
2993         GrB_Type      type = GrB_INVALID_HANDLE;
2994         GrB_Semiring   semiring = GrB_INVALID_HANDLE;
2995         GrB_Matrix     matrix = GrB_INVALID_HANDLE;
```

2996 Compliance

2997 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.
 2998 That is, for each operation we give a recipe for producing its outcome. Any implementation that
 2999 produces the same outcome, and follows the GraphBLAS execution model (*Section 2.5*) and error
 3000 model (*Section 2.6*) is a conforming implementation.

3001 4.3.1 mxm: Matrix-matrix multiply

3002 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

3003 C Syntax

```
3004         GrB_Info GrB_mxm(GrB_Matrix      C,
3005                           const GrB_Matrix Mask,
3006                           const GrB_BinaryOp accum,
3007                           const GrB_Semiring op,
3008                           const GrB_Matrix A,
3009                           const GrB_Matrix B,
```

3010 `const GrB_Descriptor desc);`

3011 Parameters

3012 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3013 that may be accumulated with the result of the matrix product. On output, the
3014 matrix holds the results of the operation.

3015 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3016 stored into the output matrix **C**. The mask dimensions must match those of the
3017 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3018 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
3019 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3020 dimensions of **C**), **GrB_NULL** should be specified.

3021 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
3022 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
3023 specified.

3024 **op** (IN) The semiring used in the matrix-matrix multiply.

3025 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
3026 multiplication.

3027 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
3028 multiplication.

3029 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
3030 should be specified. Non-default field/value pairs are listed as follows:
3031

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.

3033 Return Values

3034 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
3035 blocking mode, this indicates that the compatibility tests on di-
3036 mensions and domains for the input arguments passed successfully.

3037 Either way, output matrix C is ready to be used in the next method
3038 of the sequence.

3039 **GrB_PANIC** Unknown internal error.

3040 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
3041 GraphBLAS objects (input or output) is in an invalid state caused
3042 by a previous execution error. Call **GrB_error()** to access any error
3043 messages generated by the implementation.

3044 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

3045 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
3046 a call to **new** (or **Matrix_dup** for matrix parameters).

3047 **GrB_DIMENSION_MISMATCH** Mask and/or matrix dimensions are incompatible.

3048 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
3049 corresponding domains of the semiring or accumulation operator,
3050 or the mask's domain is not compatible with **bool** (in the case where
3051 **desc[GrB_MASK].GrB_STRUCTURE** is not set).

3052 Description

3053 **GrB_mxm** computes the matrix product $C = A \oplus . \otimes B$ or, if an optional binary accumulation operator
3054 (\odot) is provided, $C = C \odot (A \oplus . \otimes B)$ (where matrices A and B can be optionally transposed).
3055 Logically, this operation occurs in three steps:

3056 **Setup** The internal matrices and mask used in the computation are formed and their domains
3057 and dimensions are tested for compatibility.

3058 **Compute** The indicated computations are carried out.

3059 **Output** The result is written into the output matrix, possibly under control of a mask.

3060 Up to four argument matrices are used in the **GrB_mxm** operation:

- 3061 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3062 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)
- 3063 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3064 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3065 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for
3066 domain compatibility as follows:

- 3067 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
3068 must be from one of the pre-defined types of Table 3.2.
- 3069 2. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 3070 3. $\mathbf{D}(\mathbf{B})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 3071 4. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 3072 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3073 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
3074 of the accumulation operator.

3075 Two domains are compatible with each other if values from one domain can be cast to values in
3076 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
3077 all compatible with each other. A domain from a user-defined type is only compatible with itself.
3078 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch
3079 error listed above is returned.

3080 From the argument matrices, the internal matrices and mask used in the computation are formed
3081 (\leftarrow denotes copy):

- 3082 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3083 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3084 (a) If `Mask` = `GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
3085 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3086 (b) If `Mask` \neq `GrB_NULL`,
 - 3087 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
3088 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - 3089 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
3090 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - 3091 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3092 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3093 4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3094 The internal matrices and masks are checked for dimension compatibility. The following conditions
3095 must hold:

- 3096 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 3097 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 3098 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 3099 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3100 5. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.

3101 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch
3102 error listed above is returned.

3103 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3104 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3105 We are now ready to carry out the matrix multiplication and any additional associated operations.
3106 We describe this in terms of two intermediate matrices:

- 3107 • $\tilde{\mathbf{T}}$: The matrix holding the product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3108 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3109 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j)) \neq \emptyset\} \rangle$ is created. The value of each of its elements is computed by

$$3111 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3112 where \oplus and \otimes are the additive and multiplicative operators of semiring `op`, respectively.

3113 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3114 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3115 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3116 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3117 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
3118 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 3119 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3120 \\ 3121 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3122 \\ 3123 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \end{aligned}$$

3124 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3125 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
3126 using what is called a *standard matrix mask and replace*. This is carried out under control of the
3127 mask which acts as a “write mask”.

- 3128 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
3129 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3130 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 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, \mathbf{C} , and elements of \mathbf{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 \mathbf{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 \mathbf{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.

4.3.2 vxm: Vector-matrix multiply

Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

C Syntax

```
GrB_Info GrB_vxm(GrB_Vector      w,
                  const GrB_Vector mask,
                  const GrB_BinaryOp accum,
                  const GrB_Semiring op,
                  const GrB_Vector u,
                  const GrB_Matrix A,
                  const GrB_Descriptor desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the vector-matrix product. On output, this vector holds the results of the operation.

mask (IN) An optional “write” mask that controls which results from this operation are stored into the output vector **w**. The mask dimensions must match those of the vector **w**. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain of the **mask** vector must be of type **bool** or any of the predefined “built-in” types in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the dimensions of **w**), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing **w** entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) Semiring used in the vector-matrix multiply.

3164 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 3165 multiplication.

3166 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3167 multiplication.

3168 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 3169 should be specified. Non-default field/value pairs are listed as follows:

3170

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.

3171

3172 **Return Values**

3173 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 3174 blocking mode, this indicates that the compatibility tests on di-
 3175 mensions and domains for the input arguments passed successfully.
 3176 Either way, output vector **w** is ready to be used in the next method
 3177 of the sequence.

3178 **GrB_PANIC** Unknown internal error.

3179 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 3180 GraphBLAS objects (input or output) is in an invalid state caused
 3181 by a previous execution error. Call **GrB_error()** to access any error
 3182 messages generated by the implementation.

3183 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

3184 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 3185 a call to **new** (or **dup** for matrix or vector parameters).

3186 **GrB_DIMENSION_MISMATCH** Mask, vector, and/or matrix dimensions are incompatible.

3187 **GrB_DOMAIN_MISMATCH** The domains of the various vectors/matrices are incompatible with
 3188 the corresponding domains of the semiring or accumulation opera-
 3189 tor, or the mask's domain is not compatible with **bool** (in the case
 3190 where **desc[GrB_MASK].GrB_STRUCTURE** is not set).

3191 Description

3192 GrB_vxm computes the vector-matrix product $w^T = u^T \oplus . \otimes A$, or, if an optional binary accu-
 3193 mulation operator (\odot) is provided, $w^T = w^T \odot (u^T \oplus . \otimes A)$ (where matrix A can be optionally
 3194 transposed). Logically, this operation occurs in three steps:

3195 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
 3196 domains/dimensions are tested for compatibility.

3197 **Compute** The indicated computations are carried out.

3198 **Output** The result is written into the output vector, possibly under control of a mask.

3199 Up to four argument vectors or matrices are used in the GrB_vxm operation:

- 3200 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3201 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3202 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3203 4. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

3204 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
 3205 tested for domain compatibility as follows:

- 3206 1. If mask is not GrB_NULL, and $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is not set, then $\mathbf{D}(\text{mask})$
 3207 must be from one of the pre-defined types of Table 3.2.
- 3208 2. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 3209 3. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 3210 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 3211 5. If accum is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3212 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 3213 of the accumulation operator.

3214 Two domains are compatible with each other if values from one domain can be cast to values in
 3215 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3216 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 3217 If any compatibility rule above is violated, execution of GrB_vxm ends and the domain mismatch
 3218 error listed above is returned.

3219 From the argument vectors and matrices, the internal matrices and mask used in the computation
 3220 are formed (\leftarrow denotes copy):

- 3221 1. Vector $\tilde{w} \leftarrow w$.

- 3222 2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 3223 (a) If `mask = GrB_NULL`, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 3224 (b) If `mask \neq GrB_NULL`,
- 3225 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 3226 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 3227 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
- 3228 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3229 4. Matrix $\widetilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

3230 The internal matrices and masks are checked for shape compatibility. The following conditions
3231 must hold:

- 3232 1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}})$.
- 3233 2. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.
- 3234 3. $\mathbf{size}(\widetilde{\mathbf{u}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

3235 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch
3236 error listed above is returned.

3237 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3238 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3239 We are now ready to carry out the vector-matrix multiplication and any additional associated
3240 operations. We describe this in terms of two intermediate vectors:

- 3241 • $\widetilde{\mathbf{t}}$: The vector holding the product of vector $\widetilde{\mathbf{u}}^T$ and matrix $\widetilde{\mathbf{A}}$.
- 3242 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3243 The intermediate vector $\widetilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \{(j, t_j) : \mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{A}}(:, j)) \neq \emptyset\} \rangle$ is created.
3244 The value of each of its elements is computed by

$$3245 \quad t_j = \bigoplus_{k \in \mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{A}}(:, j))} (\widetilde{\mathbf{u}}(k) \otimes \widetilde{\mathbf{A}}(k, j)),$$

3246 where \oplus and \otimes are the additive and multiplicative operators of semiring `op`, respectively.

3247 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3248 • If `accum = GrB_NULL`, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.

3249 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3250 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3251 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3252 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3253 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3254 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3255 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3256 \end{aligned}$$

3257 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.
3258

3259 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
3260 using what is called a *standard vector mask and replace*. This is carried out under control of the
3261 mask which acts as a “write mask”.

3262 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
3263 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3264 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3265 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
3266 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
3267 mask are unchanged:

$$3268 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3269 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
3270 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
3271 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
3272 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3273 sequence.

3274 4.3.3 mxv: Matrix-vector multiply

3275 Multiplies a matrix by a vector on a semiring. The result is a vector.

3276 C Syntax

```
3277 GrB_Info GrB_mxv(GrB_Vector w,
3278                  const GrB_Vector mask,
3279                  const GrB_BinaryOp accum,
3280                  const GrB_Semiring op,
3281                  const GrB_Matrix A,
3282                  const GrB_Vector u,
3283                  const GrB_Descriptor desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the matrix-vector product. On output, this vector holds the results of the operation.

mask (IN) An optional “write” mask that controls which results from this operation are stored into the output vector **w**. The mask dimensions must match those of the vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain of the **mask** vector must be of type **bool** or any of the predefined “built-in” types in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the dimensions of **w**), **GrB_NULL** should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing **w** entries. If assignment rather than accumulation is desired, **GrB_NULL** should be specified.

op (IN) Semiring used in the vector-matrix multiply.

A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the multiplication.

u (IN) The GraphBLAS vector holding the values for the right-hand vector in the multiplication.

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
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.

Return Values

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.

GrB_PANIC Unknown internal error.

3313 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
3314 GraphBLAS objects (input or output) is in an invalid state caused
3315 by a previous execution error. Call `GrB_error()` to access any error
3316 messages generated by the implementation.

3317 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

3318 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
3319 a call to `new` (or `dup` for matrix or vector parameters).

3320 **GrB_DIMENSION_MISMATCH** Mask, vector, and/or matrix dimensions are incompatible.

3321 **GrB_DOMAIN_MISMATCH** The domains of the various vectors/matrices are incompatible with
3322 the corresponding domains of the semiring or accumulation opera-
3323 tor, or the mask's domain is not compatible with `bool` (in the case
3324 where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3325 Description

3326 **GrB_mvx** computes the matrix-vector product $w = A \oplus . \otimes u$, or, if an optional binary accumulation
3327 operator (\odot) is provided, $w = w \odot (A \oplus . \otimes u)$ (where matrix A can be optionally transposed).
3328 Logically, this operation occurs in three steps:

3329 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
3330 domains/dimensions are tested for compatibility.

3331 **Compute** The indicated computations are carried out.

3332 **Output** The result is written into the output vector, possibly under control of a mask.

3333 Up to four argument vectors or matrices are used in the **GrB_mvx** operation:

- 3334 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3335 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3336 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3337 4. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3338 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
3339 tested for domain compatibility as follows:

- 3340 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3341 must be from one of the pre-defined types of Table 3.2.
- 3342 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 3343 3. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.

- 3344 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 3345 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
- 3346 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
- 3347 of the accumulation operator.

3348 Two domains are compatible with each other if values from one domain can be cast to values in

3349 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are

3350 all compatible with each other. A domain from a user-defined type is only compatible with itself.

3351 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the domain mismatch

3352 error listed above is returned.

3353 From the argument vectors and matrices, the internal matrices and mask used in the computation

3354 are formed (\leftarrow denotes copy):

- 3355 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3356 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 3357 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
- 3358 (b) If `mask \neq GrB_NULL`,
- 3359 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
- 3360 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 3361 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3362 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3363 4. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3364 The internal matrices and masks are checked for shape compatibility. The following conditions

3365 must hold:

- 3366 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.
- 3367 2. $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}(\tilde{\mathbf{A}})$.
- 3368 3. $\text{size}(\tilde{\mathbf{u}}) = \text{ncols}(\tilde{\mathbf{A}})$.

3369 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the dimension mismatch

3370 error listed above is returned.

3371 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

3372 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3373 We are now ready to carry out the matrix-vector multiplication and any additional associated

3374 operations. We describe this in terms of two intermediate vectors:

- 3375 • $\tilde{\mathbf{t}}$: The vector holding the product of matrix $\tilde{\mathbf{A}}$ and vector $\tilde{\mathbf{u}}$.

3376 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3377 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.
 3378 The value of each of its elements is computed by

$$3379 \quad t_i = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3380 where \oplus and \otimes are the additive and multiplicative operators of semiring \mathbf{op} , respectively.

3381 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3382 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3383 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3384 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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 $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3387 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3388 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3389 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3390 \quad & \\ 3391 \end{aligned}$$

3392 where $\odot = \odot(\mathbf{accum})$, 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 $\mathbf{desc}[\mathbf{GrB_OUTP}].\mathbf{GrB_REPLACE}$ 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 $\mathbf{desc}[\mathbf{GrB_OUTP}].\mathbf{GrB_REPLACE}$ 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 $\mathbf{GrB_BLOCKING}$ mode, the method exits with return value $\mathbf{GrB_SUCCESS}$ and the new content
 3404 of vector \mathbf{w} is as defined above and fully computed. In $\mathbf{GrB_NONBLOCKING}$ mode, the method
 3405 exits with return value $\mathbf{GrB_SUCCESS}$ 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 eWiseMult: Element-wise multiplication

3409 **Note:** The difference between eWiseAdd and eWiseMult is not about the element-wise operation
3410 but how the index sets are treated. eWiseAdd returns an object whose indices are the “union” of
3411 the indices of the inputs whereas eWiseMult returns an object whose indices are the “intersection”
3412 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
3413 the set of values from the resulting index set.

3414 4.3.4.1 eWiseMult: Vector variant

3415 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-
3416 ducing a third vector as result.

3417 C Syntax

```
3418     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3419                           const GrB_Vector mask,  
3420                           const GrB_BinaryOp accum,  
3421                           const GrB_Semiring op,  
3422                           const GrB_Vector u,  
3423                           const GrB_Vector v,  
3424                           const GrB_Descriptor desc);  
3425  
3426     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3427                           const GrB_Vector mask,  
3428                           const GrB_BinaryOp accum,  
3429                           const GrB_Monoid op,  
3430                           const GrB_Vector u,  
3431                           const GrB_Vector v,  
3432                           const GrB_Descriptor desc);  
3433  
3434     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3435                           const GrB_Vector mask,  
3436                           const GrB_BinaryOp accum,  
3437                           const GrB_BinaryOp op,  
3438                           const GrB_Vector u,  
3439                           const GrB_Vector v,  
3440                           const GrB_Descriptor desc);
```

3441 Parameters

3442 w (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3443 that may be accumulated with the result of the element-wise operation. On output,
3444 this vector holds the results of the operation.

3445 **mask** (IN) An optional “write” mask that controls which results from this operation are
3446 stored into the output vector **w**. The mask dimensions must match those of the
3447 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3448 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3449 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3450 dimensions of **w**), **GrB_NULL** should be specified.

3451 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3452 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
3453 specified.

3454 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3455 operation. Depending on which type is passed, the following defines the binary
3456 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3457 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3458 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-
3459 nored.

3460 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
3461 is ignored.

3462 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3463 operation.

3464 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
3465 operation.

3466 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
3467 should be specified. Non-default field/value pairs are listed as follows:

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 .

3470 Return Values

3471 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
3472 blocking mode, this indicates that the compatibility tests on di-
3473 mensions and domains for the input arguments passed successfully.
3474 Either way, output vector **w** is ready to be used in the next method
3475 of the sequence.

3476 GrB_PANIC Unknown internal error.

3477 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3478 GraphBLAS objects (input or output) is in an invalid state caused
3479 by a previous execution error. Call GrB_error() to access any error
3480 messages generated by the implementation.

3481 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3482 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3483 a call to new (or dup for vector parameters).

3484 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3485 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
3486 responding domains of the binary operator (op) or accumulation
3487 operator, or the mask's domain is not compatible with bool (in the
3488 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

3489 Description

3490 This variant of GrB_eWiseMult computes the element-wise “product” of two GraphBLAS vectors:
3491 $w = u \otimes v$, or, if an optional binary accumulation operator (\odot) is provided, $w = w \odot (u \otimes v)$.
3492 Logically, this operation occurs in three steps:

3493 **Setup** The internal vectors and mask used in the computation are formed and their domains
3494 and dimensions are tested for compatibility.

3495 **Compute** The indicated computations are carried out.

3496 **Output** The result is written into the output vector, possibly under control of a mask.

3497 Up to four argument vectors are used in the GrB_eWiseMult operation:

- 3498 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3499 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3500 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3501 4. $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3502 The argument vectors, the “product” operator (op), and the accumulation operator (if provided)
3503 are tested for domain compatibility as follows:

- 3504 1. If mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
3505 must be from one of the pre-defined types of Table 3.2.
- 3506 2. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.

- 3507 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 3508 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3509 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3510 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
 3511 the accumulation operator.

3512 Two domains are compatible with each other if values from one domain can be cast to values in
 3513 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3514 compatible with each other. A domain from a user-defined type is only compatible with itself. If any
 3515 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3516 error listed above is returned.

3517 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3518 denotes copy):

- 3519 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3520 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3521 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3522 (b) If `mask` \neq `GrB_NULL`,
 - 3523 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3524 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 3525 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3526 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3527 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3528 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3529 must hold:

- 3530 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}}) = \mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{v}})$.

3531 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3532 mismatch error listed above is returned.

3533 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3534 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3535 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3536 We describe this in terms of two intermediate vectors:

- 3537 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “product” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3538 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3539 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
 3540 value of each of its elements is computed by:

$$3541 \quad t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}))$$

3542 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3543 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3544 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3545 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3546 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3547 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3548 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3549 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3550 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3551 \end{aligned}$$

3552 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3554 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3555 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3556 mask which acts as a “write mask”.

- 3557 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 3558 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3559 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3560 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3561 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3562 mask are unchanged:

$$3563 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3564 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3565 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3566 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
 3567 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3568 sequence.

3569 4.3.4.2 eWiseMult: Matrix variant

3570 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-
 3571 ducing a third matrix as result.

3572 C Syntax

```

3573     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3574                           const GrB_Matrix Mask,
3575                           const GrB_BinaryOp accum,
3576                           const GrB_Semiring op,
3577                           const GrB_Matrix A,
3578                           const GrB_Matrix B,
3579                           const GrB_Descriptor desc);
3580
3581     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3582                           const GrB_Matrix Mask,
3583                           const GrB_BinaryOp accum,
3584                           const GrB_Monoid op,
3585                           const GrB_Matrix A,
3586                           const GrB_Matrix B,
3587                           const GrB_Descriptor desc);
3588
3589     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3590                           const GrB_Matrix Mask,
3591                           const GrB_BinaryOp accum,
3592                           const GrB_BinaryOp op,
3593                           const GrB_Matrix A,
3594                           const GrB_Matrix B,
3595                           const GrB_Descriptor desc);

```

3596 Parameters

3597 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3598 that may be accumulated with the result of the element-wise operation. On output,
3599 the matrix holds the results of the operation.

3600 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3601 stored into the output matrix C. The mask dimensions must match those of the
3602 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3603 of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types
3604 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
3605 dimensions of C), `GrB_NULL` should be specified.

3606 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3607 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3608 specified.

3609 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3610 operation. Depending on which type is passed, the following defines the binary
3611 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3640 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3641 corresponding domains of the binary operator (\otimes) or accumulation
 3642 operator, or the mask's domain is not compatible with `bool` (in the
 3643 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3644 Description

3645 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:
 3646 $C = A \otimes B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \otimes B)$.
 3647 Logically, this operation occurs in three steps:

3648 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3649 and dimensions are tested for compatibility.

3650 **Compute** The indicated computations are carried out.

3651 **Output** The result is written into the output matrix, possibly under control of a mask.

3652 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3653 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3654 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)
- 3655 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3656 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3657 The argument matrices, the “product” operator (\otimes), and the accumulation operator (if provided)
 3658 are tested for domain compatibility as follows:

- 3659 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3660 must be from one of the pre-defined types of Table 3.2.
- 3661 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\otimes)$.
- 3662 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\otimes)$.
- 3663 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\otimes)$.
- 3664 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3665 of the accumulation operator and $\mathbf{D}_{out}(\otimes)$ of \otimes must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3666 the accumulation operator.

3667 Two domains are compatible with each other if values from one domain can be cast to values in
 3668 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3669 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3670 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3671 error listed above is returned.

3672 From the argument matrices, the internal matrices and mask used in the computation are formed
 3673 (\leftarrow denotes copy):

- 3674 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3675 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3676 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3677 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3678 (b) If `Mask \neq GrB_NULL`,
 - 3679 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3680 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3681 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3682 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$.
 - 3683 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3684 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3685 4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3686 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3687 must hold:

- 3688 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3689 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3690 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3691 mismatch error listed above is returned.

3692 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3693 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3694 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3695 We describe this in terms of two intermediate matrices:

- 3696 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise product of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3697 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3698 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$
 3699 is created. The value of each of its elements is computed by

$$3700 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3701 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

3702 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

3703 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3704 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3705 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
3706 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3707 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3708 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3710 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3712 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3713 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
3714 using what is called a *standard matrix mask and replace*. This is carried out under control of the
3715 mask which acts as a “write mask”.

3716 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
3717 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3718 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3719 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
3720 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
3721 mask are unchanged:

$$3722 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3723 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
3724 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
3725 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
3726 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3727 sequence.

3728 4.3.5 eWiseAdd: Element-wise addition

3729 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation
3730 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of
3731 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”
3732 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
3733 the set of values from the resulting index set.

3734 4.3.5.1 eWiseAdd: Vector variant

3735 Perform element-wise (general) addition on the elements of two vectors, producing a third vector
3736 as result.

3737 C Syntax

```
3738     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3739                          const GrB_Vector mask,  
3740                          const GrB_BinaryOp accum,  
3741                          const GrB_Semiring op,  
3742                          const GrB_Vector u,  
3743                          const GrB_Vector v,  
3744                          const GrB_Descriptor desc);  
3745  
3746     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3747                          const GrB_Vector mask,  
3748                          const GrB_BinaryOp accum,  
3749                          const GrB_Monoid op,  
3750                          const GrB_Vector u,  
3751                          const GrB_Vector v,  
3752                          const GrB_Descriptor desc);  
3753  
3754     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3755                          const GrB_Vector mask,  
3756                          const GrB_BinaryOp accum,  
3757                          const GrB_BinaryOp op,  
3758                          const GrB_Vector u,  
3759                          const GrB_Vector v,  
3760                          const GrB_Descriptor desc);
```

3761 Parameters

3762 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3763 that may be accumulated with the result of the element-wise operation. On output,
3764 this vector 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 vector **w**. The mask dimensions must match those of the
3767 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3768 of the **mask** vector 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 **w**), **GrB_NULL** should be specified.

3771 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

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 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3778 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-
3779 nored.

3780 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-
3781 nary op and additive identity are ignored.

3782 u (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3783 operation.

3784 v (IN) The GraphBLAS vector holding the values for the right-hand vector in the
3785 operation.

3786 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
3787 should be specified. Non-default field/value pairs are listed as follows:
3788

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.

3790 Return Values

3791 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3792 blocking mode, this indicates that the compatibility tests on di-
3793 mensions and domains for the input arguments passed successfully.
3794 Either way, output vector w is ready to be used in the next method
3795 of the sequence.

3796 GrB_PANIC Unknown internal error.

3797 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3798 GraphBLAS objects (input or output) is in an invalid state caused
3799 by a previous execution error. Call GrB_error() to access any error
3800 messages generated by the implementation.

3801 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3802 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3803 a call to `new` (or `dup` for vector parameters).

3804 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3805 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 3806 responding domains of the binary operator (`op`) 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 vectors: $\mathbf{w} =$
 3811 $\mathbf{u} \oplus \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$. Logically,
 3812 this operation occurs in three steps:

3813 **Setup** The internal vectors 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 vector, possibly under control of a mask.

3817 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3818 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3819 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3820 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3821 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3822 The argument vectors, the “sum” operator (`op`), 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 $\mathbf{D}(\mathbf{mask})$
 3825 must be from one of the pre-defined types of Table 3.2.
- 3826 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 3827 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 3828 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3829 5. $\mathbf{D}(\mathbf{u})$ and $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3830 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3831 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ 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 vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3839 denotes copy):

- 3840 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3841 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3842 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3843 (b) If `mask \neq GrB_NULL`,
 - 3844 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3845 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3846 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3847 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3848 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3849 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3850 must hold:

- 3851 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$.

3852 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 3853 mismatch error listed above is returned.

3854 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3855 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3856 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3857 We describe this in terms of two intermediate vectors:

- 3858 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “sum” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3859 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3860 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{u}}) \cup \text{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$ is created. The
 3861 value of each of its elements is computed by:

$$\begin{aligned}
 3862 \quad t_i &= (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})) \\
 3863 \quad t_i &= \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))) \\
 3864
 \end{aligned}$$

3865
3866

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3867 where the difference operator in the previous expressions refers to set difference.

3868 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3869 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3870 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3872 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3873 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3874 z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3875 z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3876 z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

3877
3878
3879 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3880 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
3881 using what is called a *standard vector mask and replace*. This is carried out under control of the
3882 mask which acts as a “write mask”.

- 3883 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
3884 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3886 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
3887 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
3888 mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3890 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
3891 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
3892 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
3893 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3894 sequence.

3895 4.3.5.2 eWiseAdd: Matrix variant

3896 Perform element-wise (general) addition on the elements of two matrices, producing a third matrix
3897 as result.

3898 C Syntax

```

3899     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3900                          const GrB_Matrix Mask,
3901                          const GrB_BinaryOp accum,
3902                          const GrB_Semiring op,
3903                          const GrB_Matrix A,
3904                          const GrB_Matrix B,
3905                          const GrB_Descriptor desc);
3906
3907     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3908                          const GrB_Matrix Mask,
3909                          const GrB_BinaryOp accum,
3910                          const GrB_Monoid op,
3911                          const GrB_Matrix A,
3912                          const GrB_Matrix B,
3913                          const GrB_Descriptor desc);
3914
3915     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3916                          const GrB_Matrix Mask,
3917                          const GrB_BinaryOp accum,
3918                          const GrB_BinaryOp op,
3919                          const GrB_Matrix A,
3920                          const GrB_Matrix B,
3921                          const GrB_Descriptor desc);

```

3922 Parameters

3923 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3924 that may be accumulated with the result of the element-wise operation. On output,
3925 the matrix holds the results of the operation.

3926 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3927 stored into the output matrix C. The mask dimensions must match those of the
3928 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
3929 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
3930 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
3931 dimensions of C), GrB_NULL should be specified.

3932 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3933 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3934 specified.

3935 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
3936 operation. Depending on which type is passed, the following defines the binary
3937 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3938 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 3939 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-
 3940 nored.
 3941 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-
 3942 nary op and additive identity are ignored.

3943 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3944 operation.

3945 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3946 operation.

3947 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3948 should be specified. Non-default field/value pairs are listed as follows:
 3949

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.

3951 Return Values

3952 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3953 blocking mode, this indicates that the compatibility tests on di-
 3954 mensions and domains for the input arguments passed successfully.
 3955 Either way, output matrix C is ready to be used in the next method
 3956 of the sequence.

3957 GrB_PANIC Unknown internal error.

3958 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3959 GraphBLAS objects (input or output) is in an invalid state caused
 3960 by a previous execution error. Call GrB_error() to access any error
 3961 messages generated by the implementation.

3962 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3963 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3964 a call to new (or Matrix_dup for matrix parameters).

3965 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3966 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3967 corresponding domains of the binary operator (op) or accumulation
 3968 operator, or the mask's domain is not compatible with `bool` (in the
 3969 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3970 Description

3971 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:
 3972 $C = A \oplus B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \oplus B)$.
 3973 Logically, this operation occurs in three steps:

3974 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3975 and dimensions are tested for compatibility.

3976 **Compute** The indicated computations are carried out.

3977 **Output** The result is written into the output matrix, possibly under control of a mask.

3978 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3979 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3980 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)
- 3981 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3982 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3983 The argument matrices, the “sum” operator (op), and the accumulation operator (if provided) are
 3984 tested for domain compatibility as follows:

- 3985 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3986 must be from one of the pre-defined types of Table 3.2.
- 3987 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3988 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3989 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3990 5. $\mathbf{D}(A)$ and $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3991 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3992 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3993 the accumulation operator.

3994 Two domains are compatible with each other if values from one domain can be cast to values in
 3995 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3996 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3997 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch
 3998 error listed above is returned.

3999 From the argument matrices, the internal matrices and mask used in the computation are formed
 4000 (\leftarrow denotes copy):

- 4001 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4002 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 4003 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 4004 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 4005 (b) If `Mask \neq GrB_NULL`,
 - 4006 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 4007 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 4008 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 4009 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 4010 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4011 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 4012 4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

4013 The internal matrices and masks are checked for dimension compatibility. The following conditions
 4014 must hold:

- 4015 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 4016 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

4017 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 4018 mismatch error listed above is returned.

4019 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4020 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4021 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 4022 We describe this in terms of two intermediate matrices:

- 4023 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise sum of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 4024 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4025 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$
 4026 is created. The value of each of its elements is computed by

$$\begin{aligned} 4027 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\ 4028 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\ 4029 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \end{aligned}$$

4032 where the difference operator in the previous expressions refers to set difference.

4033 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4034 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 4035 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4036 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4037 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4038 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 4039 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4040 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4041 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \end{aligned}$$

4044 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4045 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4046 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4047 mask which acts as a “write mask”.

- 4048 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 4049 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4050 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4051 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 4052 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 4053 mask are unchanged:

$$4054 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4055 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4056 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4057 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 4058 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4059 sequence.

4060 4.3.6 extract: Selecting sub-graphs

4061 Extract a subset of a matrix or vector.

4062 4.3.6.1 extract: Standard vector variant

4063 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector
4064 whose size is equal to the number of indices.

4065 C Syntax

```
4066         GrB_Info GrB_extract(GrB_Vector          w,  
4067                             const GrB_Vector    mask,  
4068                             const GrB_BinaryOp   accum,  
4069                             const GrB_Vector    u,  
4070                             const GrB_Index     *indices,  
4071                             GrB_Index           nindices,  
4072                             const GrB_Descriptor desc);
```

4073 Parameters

4074 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4075 that may be accumulated with the result of the extract operation. On output, this
4076 vector holds the results of the operation.

4077 **mask** (IN) An optional “write” mask that controls which results from this operation are
4078 stored into the output vector **w**. The mask dimensions must match those of the
4079 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4080 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4081 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4082 dimensions of **w**), **GrB_NULL** should be specified.

4083 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4084 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4085 specified.

4086 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4087 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of
4088 elements from **u** that are extracted. If all elements of **u** are to be extracted in order
4089 from 0 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution
4090 mode and return value, this array may be manipulated by the caller after this
4091 operation returns without affecting any deferred computations for this operation.

4092 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4093 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4094 should be specified. Non-default field/value pairs are listed as follows:

4095

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 .

4096

4097 Return Values

4098 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 4099 blocking mode, this indicates that the compatibility tests on
 4100 dimensions and domains for the input arguments passed suc-
 4101 cessfully. Either way, output vector **w** is ready to be used in the
 4102 next method of the sequence.

4103 GrB_PANIC Unknown internal error.

4104 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 4105 opaque GraphBLAS objects (input or output) is in an invalid
 4106 state caused by a previous execution error. Call GrB_error() to
 4107 access any error messages generated by the implementation.

4108 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4109 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
 4110 by a call to **new** (or **dup** for vector parameters).

4111 GrB_INDEX_OUT_OF_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-
 4112 blocking mode, this error can be deferred.

4113 GrB_DIMENSION_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices** \neq **size(w)**.

4114 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each
 4115 other or the corresponding domains of the accumulation oper-
 4116 ator, or the mask's domain is not compatible with **bool** (in the
 4117 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4118 GrB_NULL_POINTER Argument **row_indices** is a NULL pointer.

4119 Description

4120 This variant of GrB_extract computes the result of extracting a subset of locations from a Graph-
 4121 BLAS vector in a specific order: **w** = **u(indices)**; or, if an optional binary accumulation operator

4122 (\odot) is provided, $w = w \odot u(\text{indices})$. More explicitly:

$$4123 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4124 Logically, this operation occurs in three steps:

4125 **Setup** The internal vectors and mask used in the computation are formed and their domains
4126 and dimensions are tested for compatibility.

4127 **Compute** The indicated computations are carried out.

4128 **Output** The result is written into the output vector, possibly under control of a mask.

4129 Up to three argument vectors are used in this GrB_extract operation:

- 4130 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4131 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4132 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4133 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4134 bility as follows:

- 4135 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4136 must be from one of the pre-defined types of Table 3.2.
- 4137 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(u)$.
- 4138 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4139 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4140 mulation operator.

4141 Two domains are compatible with each other if values from one domain can be cast to values in
4142 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4143 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4144 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
4145 error listed above is returned.

4146 From the arguments, the internal vectors, mask, and index array used in the computation are
4147 formed (\leftarrow denotes copy):

- 4148 1. Vector $\tilde{w} \leftarrow w$.
- 4149 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
4150 (a) If `mask = GrB_NULL`, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.

4151 (b) If $\text{mask} \neq \text{GrB_NULL}$,
4152 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
4153 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
4154 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
4155 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
4156 4. The internal index array, $\widetilde{\mathbf{I}}$, is computed from argument indices as follows:
4157 (a) If $\text{indices} = \text{GrB_ALL}$, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.
4158 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

4159 The internal vectors and mask are checked for dimension compatibility. The following conditions
4160 must hold:

- 4161 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4162 2. $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$.

4163 If any compatibility rule above is violated, execution of GrB_extract ends and the dimension mis-
4164 match error listed above is returned.

4165 From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with
4166 GrB_SUCCESS return code and defer any computation and/or execution error codes.

4167 We are now ready to carry out the extract and any additional associated operations. We describe
4168 this in terms of two intermediate vectors:

- 4169 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{w}}$.
- 4170 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4171 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4172 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4173 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
4174 GrB_extract ends and the index-out-of-bounds error listed above is generated. In GrB_NONBLOCKING
4175 mode, the error can be deferred until a sequence-terminating $\text{GrB_wait}()$ is called. Regardless, the
4176 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4177 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 4178 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 4179 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$4180 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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 $\tilde{\mathbf{t}}$.

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector \mathbf{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 \mathbf{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.

4.3.6.2 extract: Standard matrix variant

Extract a sub-matrix from a larger matrix as specified by a set of row indices and a set of column indices. The result is a matrix whose size is equal to size of the sets of indices.

C Syntax

```
GrB_Info GrB_extract(GrB_Matrix      C,
                    const GrB_Matrix  Mask,
                    const GrB_BinaryOp accum,
                    const GrB_Matrix  A,
                    const GrB_Index   *row_indices,
                    GrB_Index          nrows,
                    const GrB_Index   *col_indices,
                    GrB_Index          ncols,
                    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 extract operation. On output, the matrix holds the results of the operation.

Mask (IN) An optional “write” mask that controls which results from this operation are stored into the output matrix **C**. The mask dimensions must match those 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 **bool** or any of the predefined “built-in” types in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the dimensions of **C**), **GrB_NULL** should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing **C** entries. If assignment rather than accumulation is desired, **GrB_NULL** should be specified.

A (IN) The GraphBLAS matrix from which the subset is extracted.

row_indices (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **A** from which elements are extracted. If elements in all rows of **A** are to be extracted in order, **GrB_ALL** should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

nrows (IN) The number of values in the **row_indices** array. Must be equal to **nrows(C)**.

col_indices (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **A** from which elements are extracted. If elements in all columns of **A** are to be extracted in order, then **GrB_ALL** should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

ncols (IN) The number of values in the **col_indices** array. Must be equal to **ncols(C)**.

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.

4247 Return Values

4248	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
4249		blocking mode, this indicates that the compatibility tests on
4250		dimensions and domains for the input arguments passed suc-
4251		cessfully. Either way, output matrix C is ready to be used in the
4252		next method of the sequence.
4253	GrB_PANIC	Unknown internal error.
4254	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
4255		opaque GraphBLAS objects (input or output) is in an invalid
4256		state caused by a previous execution error. Call <code>GrB_error()</code> to
4257		access any error messages generated by the implementation.
4258	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
4259	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
4260		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4261	GrB_INDEX_OUT_OF_BOUNDS	A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or
4262		a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In
4263		non-blocking mode, this error can be deferred.
4264	GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, <code>nrows</code> \neq <code>nrows(C)</code> , or
4265		<code>ncols</code> \neq <code>ncols(C)</code> .
4266	GrB_DOMAIN_MISMATCH	The domains of the various matrices are incompatible with each
4267		other or the corresponding domains of the accumulation oper-
4268		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4269		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4270	GrB_NULL_POINTER	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4271		is a NULL pointer, or both.

4272 Description

4273 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified
 4274 rows and columns of a GraphBLAS matrix in a specific order: $C = A(\text{row_indices}, \text{col_indices})$; or,
 4275 if an optional binary accumulation operator (\odot) is provided, $C = C \odot A(\text{row_indices}, \text{col_indices})$.
 4276 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4278 Logically, this operation occurs in three steps:

4279 **Setup** The internal matrices and mask used in the computation are formed and their domains
 4280 and dimensions are tested for compatibility.

4281 **Compute** The indicated computations are carried out.

4282 **Output** The result is written into the output matrix, possibly under control of a mask.

4283 Up to three argument matrices are used in the `GrB_extract` operation:

- 4284 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
4285 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)
4286 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4287 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
4288 ibility as follows:

- 4289 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
4290 must be from one of the pre-defined types of Table 3.2.
4291 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
4292 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4293 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4294 mulation operator.

4295 Two domains are compatible with each other if values from one domain can be cast to values in
4296 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4297 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4298 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
4299 error listed above is returned.

4300 From the arguments, the internal matrices, mask, and index arrays used in the computation are
4301 formed (\leftarrow denotes copy):

- 4302 1. Matrix $\tilde{C} \leftarrow C$.
4303 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:
4304 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
4305 $j < \mathbf{ncols}(C)\} \rangle$.
4306 (b) If `Mask` \neq `GrB_NULL`,
4307 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
4308 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
4309 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
4310 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
4311 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
4312 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

- 4313 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4314 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 4315 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \text{nrows}$.
- 4316 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4317 (a) If `col_indices` = `GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$.
- 4318 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \text{ncols}$.

4319 The internal matrices and mask are checked for dimension compatibility. The following conditions
4320 must hold:

- 4321 1. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$.
- 4322 2. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$.
- 4323 3. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$.
- 4324 4. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$.

4325 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
4326 match error listed above is returned.

4327 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4328 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4329 We are now ready to carry out the extract and any additional associated operations. We describe
4330 this in terms of two intermediate matrices:

- 4331 • $\tilde{\mathbf{T}}$: The matrix holding the extraction from $\tilde{\mathbf{A}}$.
- 4332 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4333 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

4334
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{ (i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}}) \} \rangle.$$

4335 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 $\tilde{\mathbf{J}}$
4336 array is not in the range $[0, \text{ncols}(\tilde{\mathbf{A}}))$, the execution of `GrB_extract` ends and the index out-of-
4337 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
4338 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
4339 this point forward in the sequence.

4340 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4341 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

4342 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4343 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4344 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4345 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4346 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4347 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4349 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4351 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4352 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
4353 using what is called a *standard matrix mask and replace*. This is carried out under control of the
4354 mask which acts as a “write mask”.

4355 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
4356 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4357 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4358 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
4359 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
4360 mask are unchanged:

$$4361 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4362 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4363 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4364 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
4365 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4366 sequence.

4367 4.3.6.3 extract: Column (and row) variant

4368 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the
4369 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as
4370 well.

4371 C Syntax

```

4372         GrB_Info GrB_extract(GrB_Vector      w,
4373                             const GrB_Vector  mask,
4374                             const GrB_BinaryOp accum,
4375                             const GrB_Matrix  A,
4376                             const GrB_Index  *row_indices,
4377                             GrB_Index        nrows,
4378                             GrB_Index        col_index,
4379                             const GrB_Descriptor desc);

```

4380 Parameters

4381 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4382 that may be accumulated with the result of the extract operation. On output, this
4383 vector holds the results of the operation.

4384 **mask** (IN) An optional “write” mask that controls which results from this operation are
4385 stored into the output vector **w**. The mask dimensions must match those of the
4386 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4387 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4388 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4389 dimensions of **w**), **GrB_NULL** should be specified.

4390 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4391 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4392 specified.

4393 **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4394 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations
4395 within the specified column of **A** from which elements are extracted. If elements in
4396 all rows of **A** are to be extracted in order, **GrB_ALL** should be specified. Regardless
4397 of execution mode and return value, this array may be manipulated by the caller
4398 after this operation returns without affecting any deferred computations for this
4399 operation.

4400 **nrows** (IN) The number of indices in the **row_indices** array. Must be equal to **size(w)**.

4401 **col_index** (IN) The index of the column of **A** from which to extract values. It must be in the
4402 range $[0, \mathbf{ncols}(A))$.

4403 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4404 should be specified. Non-default field/value pairs are listed as follows:

4405

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.

Return Values

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.

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 operation.

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).

GrB_INVALID_INDEX **col_index** is outside the allowable range (i.e., greater than **ncols(A)**).

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.

GrB_DIMENSION_MISMATCH **mask** and **w** dimensions are incompatible, or **nrows** \neq **size(w)**.

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).

GrB_NULL_POINTER Argument **row_indices** is a NULL pointer.

Description

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

4433 an optional binary accumulation operator (\odot) is provided, $w = w \odot A(:, \text{col_index})(\text{row_indices})$.
 4434 More explicitly:

$$4435 \quad \begin{aligned} w(i) &= A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4436 Logically, this operation occurs in three steps:

4437 **Setup** The internal matrices, vectors, and mask used in the computation are formed and their
 4438 domains and dimensions are tested for compatibility.

4439 **Compute** The indicated computations are carried out.

4440 **Output** The result is written into the output vector, possibly under control of a mask.

4441 Up to three argument vectors and matrices are used in this GrB_extract operation:

- 4442 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4443 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4444 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4445 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain
 4446 compatibility as follows:

- 4447 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
 4448 must be from one of the pre-defined types of Table 3.2.
- 4449 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(A)$.
- 4450 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4451 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4452 mulation operator.

4453 Two domains are compatible with each other if values from one domain can be cast to values in
 4454 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4455 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4456 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
 4457 error listed above is returned.

4458 From the arguments, the internal vector, matrix, mask, and index array used in the computation
 4459 are formed (\leftarrow denotes copy):

- 4460 1. Vector $\tilde{w} \leftarrow w$.
- 4461 2. One-dimensional mask, \tilde{m} , is computed from argument **mask** as follows:
 4462 (a) If **mask** = GrB_NULL, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.

4463 (b) If $\text{mask} \neq \text{GrB_NULL}$,
 4464 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 4465 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 4466 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
 4467 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
 4468 4. The internal row index array, $\widetilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
 4469 (a) If `indices = GrB_ALL`, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
 4470 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$.

4471 The internal vector, `mask`, and index array are checked for dimension compatibility. The following
 4472 conditions must hold:

- 4473 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
 4474 2. $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$.

4475 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
 4476 match error listed above is returned.

4477 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4478 1. $0 \leq \text{col_index} < \text{ncols}(\mathbf{A})$

4479 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above
 4480 is returned.

4481 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4482 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4483 We are now ready to carry out the extract and any additional associated operations. We describe
 4484 this in terms of two intermediate vectors:

- 4485 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from a column of $\widetilde{\mathbf{A}}$.
- 4486 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4487 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4488 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4489 At this point, if any value in $\widetilde{\mathbf{I}}$ is not in the range $[0, \text{nrows}(\widetilde{\mathbf{A}}))$, the execution of `GrB_extract`
 4490 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,
 4491 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result
 4492 vector, \mathbf{w} , is invalid from this point forward in the sequence.

4493 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

4494 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

4495 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4496 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4497 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4498 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4499 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

4500

$$4501 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4502

$$4503 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4504 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4505 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
4506 using what is called a *standard vector mask and replace*. This is carried out under control of the
4507 mask which acts as a “write mask”.

4508 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
4509 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4510 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4511 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4512 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
4513 mask are unchanged:

$$4514 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4515 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4516 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4517 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
4518 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4519 sequence.

4520 4.3.7 assign: Modifying sub-graphs

4521 Assign the contents of a subset of a matrix or vector.

4522 4.3.7.1 assign: Standard vector variant

4523 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.

4524 The size of the input vector is the same size as the index array provided.

4525 C Syntax

```

4526         GrB_Info GrB_assign(GrB_Vector      w,
4527                             const GrB_Vector mask,
4528                             const GrB_BinaryOp accum,
4529                             const GrB_Vector u,
4530                             const GrB_Index *indices,
4531                             GrB_Index      nindices,
4532                             const GrB_Descriptor desc);

```

4533 Parameters

4534 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4535 that may be accumulated with the result of the assign operation. On output, this
4536 vector holds the results of the operation.

4537 **mask** (IN) An optional “write” mask that controls which results from this operation are
4538 stored into the output vector **w**. The mask dimensions must match those of the
4539 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4540 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4541 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4542 dimensions of **w**), **GrB_NULL** should be specified.

4543 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4544 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4545 specified.

4546 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4547 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4548 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0
4549 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4550 and return value, this array may be manipulated by the caller after this operation
4551 returns without affecting any deferred computations for this operation. If this
4552 array contains duplicate values, it implies in assignment of more than one value to
4553 the same location which leads to undefined results.

4554 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4555 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4556 should be specified. Non-default field/value pairs are listed as follows:

4557

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.

Return Values

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.

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 operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector parameters).

GrB_INDEX_OUT_OF_BOUNDS A value in **indices** is greater than or equal to **size(w)**. In non-blocking mode, this can be reported as an execution error.

GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or **nindices** \neq **size(u)**.

GrB_DOMAIN_MISMATCH The domains of the various vectors 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).

GrB_NULL_POINTER Argument indices is a NULL pointer.

Description

This variant of **GrB_assign** computes the result of assigning elements from a source GraphBLAS vector to a destination GraphBLAS vector in a specific order: $w(\text{indices}) = u$; or, if an optional binary accumulation operator (\odot) is provided, $w(\text{indices}) = w(\text{indices}) \odot u$. More explicitly:

$$\begin{aligned}
 w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
 \end{aligned}$$

4586 Logically, this operation occurs in three steps:

4587 **Setup** The internal vectors and mask used in the computation are formed and their domains
4588 and dimensions are tested for compatibility.

4589 **Compute** The indicated computations are carried out.

4590 **Output** The result is written into the output vector, possibly under control of a mask.

4591 Up to three argument vectors are used in the `GrB_assign` operation:

- 4592 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4593 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4594 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4595 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4596 bility as follows:

- 4597 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
4598 must be from one of the pre-defined types of Table 3.2.
- 4599 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 4600 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
4601 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
4602 mulation operator.

4603 Two domains are compatible with each other if values from one domain can be cast to values in
4604 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4605 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4606 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4607 error listed above is returned.

4608 From the arguments, the internal vectors, mask and index array used in the computation are formed
4609 (\leftarrow denotes copy):

- 4610 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 4611 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 4612 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 4613 (b) If `mask` \neq `GrB_NULL`,
 - 4614 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 4615 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 4616 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

4617 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4618 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:

4619 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.

4620 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

4621 The internal vector and mask are checked for dimension compatibility. The following conditions
4622 must hold:

4623 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4624 2. $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$.

4625 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4626 match error listed above is returned.

4627 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4628 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4629 We are now ready to carry out the assign and any additional associated operations. We describe
4630 this in terms of two intermediate vectors:

- 4631 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.
- 4632 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4633 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4634 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4635 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{w}}$, computation
4636 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`
4637 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4638 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4639 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4640 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4641 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4642 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4643 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
4644 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4645 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4646 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4647 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4648 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4650 where the difference operator refers to set difference.

4651 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4652 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4653 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4654 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 4655 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 4656 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4657 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4658 \quad & \\ 4659 \end{aligned}$$

4660 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4661 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
4662 using what is called a *standard vector mask and replace*. This is carried out under control of the
4663 mask which acts as a “write mask”.

4664 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
4665 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4666 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4667 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4668 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
4669 mask are unchanged:

$$4670 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4671 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4672 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4673 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
4674 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4675 sequence.

4676 4.3.7.2 assign: Standard matrix variant

4677 Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices.
4678 The dimensions of the input matrix are the same size as the row and column index arrays provided.

4679 C Syntax

```
4680      GrB_Info GrB_assign(GrB_Matrix      C,
4681                          const GrB_Matrix Mask,
4682                          const GrB_BinaryOp accum,
4683                          const GrB_Matrix A,
```

```

4684         const GrB_Index      *row_indices,
4685         GrB_Index             nrows,
4686         const GrB_Index      *col_indices,
4687         GrB_Index             ncols,
4688         const GrB_Descriptor desc);

```

4689 Parameters

4690 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
4691 that may be accumulated with the result of the assign operation. On output, the
4692 matrix holds the results of the operation.

4693 **Mask** (IN) An optional “write” mask that controls which results from this operation are
4694 stored into the output matrix **C**. The mask dimensions must match those of the
4695 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4696 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
4697 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4698 dimensions of **C**), **GrB_NULL** should be specified.

4699 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4700 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4701 specified.

4702 **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4703 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
4704 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
4705 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
4706 this array may be manipulated by the caller after this operation returns without
4707 affecting any deferred computations for this operation. If this array contains du-
4708 plicate values, it implies assignment of more than one value to the same location
4709 which leads to undefined results.

4710 **nrows** (IN) The number of values in the **row_indices** array. Must be equal to **nrows(A)**
4711 if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4712 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns
4713 of **C** that are assigned. If all columns of **C** are to be assigned in order from 0
4714 to **ncols** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4715 and return value, this array may be manipulated by the caller after this operation
4716 returns without affecting any deferred computations for this operation. If this
4717 array contains duplicate values, it implies assignment of more than one value to
4718 the same location which leads to undefined results.

4719 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **ncols(A)** if **A** is
4720 not transposed, or equal to **nrows(A)** if **A** is transposed.

4721
4722
4723

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:

4724

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.

4725 Return Values

4726
4727
4728
4729
4730

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.

4731

GrB_PANIC Unknown internal error.

4732
4733
4734
4735

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.

4736

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

4737
4738

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).

4739
4740
4741

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.

4742
4743

GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrow \neq nrow(A), or ncols \neq ncols(A).

4744
4745
4746
4747

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).

4748
4749

GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices is a NULL pointer, or both.

4750 Description

4751 This variant of `GrB_assign` computes the result of assigning the contents of **A** to a subset of rows
 4752 and columns in **C** in a specified order: $\mathbf{C}(\text{row_indices}, \text{col_indices}) = \mathbf{A}$; or, if an optional binary
 4753 accumulation operator (\odot) is provided, $\mathbf{C}(\text{row_indices}, \text{col_indices}) = \mathbf{C}(\text{row_indices}, \text{col_indices}) \odot$
 4754 **A**. More explicitly (not accounting for an optional transpose of **A**):

$$\begin{aligned} & \mathbf{C}(\text{row_indices}[i], \text{col_indices}[j]) = \mathbf{A}(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4755 & \mathbf{C}(\text{row_indices}[i], \text{col_indices}[j]) = \mathbf{C}(\text{row_indices}[i], \text{col_indices}[j]) \odot \mathbf{A}(i, j), \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4756 Logically, this operation occurs in three steps:

4757 **Setup** The internal matrices and mask used in the computation are formed and their domains
 4758 and dimensions are tested for compatibility.

4759 **Compute** The indicated computations are carried out.

4760 **Output** The result is written into the output matrix, possibly under control of a mask.

4761 Up to three argument matrices are used in the `GrB_assign` operation:

- 4762 1. $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \text{nrows}(\mathbf{C}), \text{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 4763 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \text{nrows}(\text{Mask}), \text{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 4764 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\mathbf{A}), \text{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

4765 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
 4766 ibility as follows:

- 4767 1. If **Mask** is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 4768 must be from one of the pre-defined types of Table 3.2.
- 4769 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}(\mathbf{A})$.
- 4770 3. If **accum** is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4771 of the accumulation operator and $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4772 mulation operator.

4773 Two domains are compatible with each other if values from one domain can be cast to values in
 4774 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4775 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4776 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4777 error listed above is returned.

4778 From the arguments, the internal matrices, mask, and index arrays used in the computation are
 4779 formed (\leftarrow denotes copy):

- 4780 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4781 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
- 4782 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
4783 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 4784 (b) If `Mask \neq GrB_NULL`,
- 4785 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
4786 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
- 4787 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
4788 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
- 4789 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4790 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 4791 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4792 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
- 4793 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 4794 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4795 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
- 4796 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4797 The internal matrices and mask are checked for dimension compatibility. The following conditions
4798 must hold:

- 4799 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 4800 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 4801 3. $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$.
- 4802 4. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$.

4803 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4804 match error listed above is returned.

4805 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4806 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4807 We are now ready to carry out the assign and any additional associated operations. We describe
4808 this in terms of two intermediate vectors:

- 4809 • $\tilde{\mathbf{T}}$: The matrix holding the contents from $\tilde{\mathbf{A}}$ in their destination locations relative to $\tilde{\mathbf{C}}$.
- 4810 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4811 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4812 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4813 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
 4814 $\tilde{\mathbf{J}}$ array is not in the range $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-
 4815 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
 4816 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
 4817 this point forward in the sequence.

4818 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 4819 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$4820 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4821 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4822 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 4823 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
 4824 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

4825 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4826 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4827 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4828 \\ 4829 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4830 where the difference operator refers to set difference.

- 4831 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4832 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4833 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4834 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4835 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4836 \\ 4837 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4838 \\ 4839 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4840 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4841 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4842 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4843 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

```
GrB_Info GrB_assign(GrB_Matrix      C,
                    const GrB_Vector  mask,
                    const GrB_BinaryOp accum,
                    const GrB_Vector  u,
                    const GrB_Index   *row_indices,
                    GrB_Index         nrows,
                    GrB_Index         col_index,
                    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

4877 bool or any of the predefined “built-in” types in Table 3.2. If the default mask
 4878 is desired (i.e., a mask that is all true with the dimensions of a column of C),
 4879 GrB_NULL should be specified.

4880 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 4881 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 4882 specified.

4883 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column
 4884 of C.

4885 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
 4886 the specified column of C that are to be assigned. If all elements of the column
 4887 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB_ALL should
 4888 be specified. Regardless of execution mode and return value, this array may be
 4889 manipulated by the caller after this operation returns without affecting any de-
 4890 ferred computations for this operation. If this array contains duplicate values, it
 4891 implies in assignment of more than one value to the same location which leads to
 4892 undefined results.

4893 **nrows** (IN) The number of values in **row_indices** array. Must be equal to **size(u)**.

4894 **col_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4895 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4896 should be specified. Non-default field/value pairs are listed as follows:

4897

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 mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

4898

4899 Return Values

4900 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 4901 blocking mode, this indicates that the compatibility tests on
 4902 dimensions and domains for the input arguments passed suc-
 4903 cessfully. Either way, output matrix C is ready to be used in the
 4904 next method of the sequence.

4905 **GrB_PANIC** Unknown internal error.

4937 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4938 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
4939 compatibility as follows:

- 4940 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4941 must be from one of the pre-defined types of Table 3.2.
- 4942 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 4943 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4944 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4945 mulation operator.

4946 Two domains are compatible with each other if values from one domain can be cast to values in
4947 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4948 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4949 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4950 error listed above is returned.

4951 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4952 1. $0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{C})$

4953 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
4954 is returned.

4955 From the arguments, the internal vectors, `mask`, and index array used in the computation are
4956 formed (\leftarrow denotes copy):

- 4957 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a column of \mathbf{C} as follows:

$$4958 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

- 4959 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

- 4960 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$.
- 4961 (b) If `mask` \neq `GrB_NULL`,
 - 4962 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,
 - 4963 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 4964 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

- 4965 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

- 4966 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:

- 4967 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.

4968 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i]$, $\forall i : 0 \leq i < \text{nrows}$.

4969 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
4970 conditions must hold:

- 4971 1. $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4972 2. $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$.

4973 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4974 match error listed above is returned.

4975 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4976 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4977 We are now ready to carry out the assign and any additional associated operations. We describe
4978 this in terms of two intermediate vectors:

- 4979 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 4980 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4981 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4982 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4983 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
4984 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
4985 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4986 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4987 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4988 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4989 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4990 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4991 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
4992 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4993 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4994 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$4995 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4996 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4998 where the difference operator refers to set difference.

4999 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5000 \quad (\mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\}).$$

5001 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5002 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5003 \quad z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5004 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5005 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5006 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5009 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
5010 result matrix, $\mathbf{C}(:, \text{col_index})$. This is carried out under control of the mask which acts as a “write
5011 mask”.

5012 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(:, \text{col_index})$ on input to this
5013 operation are deleted and the new contents of the column is given by:

$$5014 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5015 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
5016 copied into the column of the final result matrix, $\mathbf{C}(:, \text{col_index})$, and elements of this column
5017 that fall outside the set indicated by the mask are unchanged:

$$5018 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup$$

$$5019 \quad \{(i, \text{col_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5020 \quad \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5021 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5022 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5023 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may
5024 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5025 4.3.7.4 assign: Row variant

5026 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the
5027 output cannot be transposed, a different variant of `assign` is provided to assign to a column of a
5028 matrix.

5029 C Syntax

```
5030         GrB_Info GrB_assign(GrB_Matrix      C,  
5031                             const GrB_Vector mask,  
5032                             const GrB_BinaryOp accum,  
5033                             const GrB_Vector u,  
5034                             GrB_Index      row_index,  
5035                             const GrB_Index *col_indices,  
5036                             GrB_Index      ncols,  
5037                             const GrB_Descriptor desc);
```

5038 Parameters

5039 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values
5040 that may be accumulated with the result of the assign operation. On output, this
5041 matrix holds the results of the operation.

5042 **mask** (IN) An optional “write” mask that controls which results from this operation are
5043 stored into the specified row of the output matrix **C**. The mask dimensions must
5044 match those of a single row of the matrix **C**. If the **GrB_STRUCTURE** descriptor
5045 is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or
5046 any of the predefined “built-in” types in Table 3.2. If the default mask is desired
5047 (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB_NULL** should
5048 be specified.

5049 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5050 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5051 specified.

5052 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of
5053 **C**.

5054 **row_index** (IN) The index of the row in **C** to assign. Must be in the range $[0, \mathbf{nrows}(\mathbf{C})]$.

5055 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
5056 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to
5057 be assigned in order from index 0 to $\mathbf{ncols} - 1$, then **GrB_ALL** should be specified.
5058 Regardless of execution mode and return value, this array may be manipulated by
5059 the caller after this operation returns without affecting any deferred computations
5060 for this operation. If this array contains duplicate values, it implies in assignment
5061 of more than one value to the same location which leads to undefined results.

5062 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **size(u)**.

5063 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5064 should be specified. Non-default field/value pairs are listed as follows:
5065

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 mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask .

Return Values

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 operation.
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).
GrB_INVALID_INDEX	row_index is outside the allowable range (i.e., greater than nrows(C)).
GrB_INDEX_OUT_OF_BOUNDS	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.
GrB_DIMENSION_MISMATCH	mask size and number of columns in C are not the same, or ncols \neq size(u) .
GrB_DOMAIN_MISMATCH	The domains of the matrix and vector 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).
GrB_NULL_POINTER	Argument col_indices is a NULL pointer.

Description

This variant of **GrB_assign** computes the result of assigning a subset of locations in a row of a GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

5094 $C(\text{row_index}, :) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(\text{row_index}, :$
5095 $) = C(\text{row_index}, :) \odot u$. Taking order of `col_indices` into account it is more explicitly written as:

5096 $C(\text{row_index}, \text{col_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$
5096 $C(\text{row_index}, \text{col_indices}[j]) = C(\text{row_index}, \text{col_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$

5097 Logically, this operation occurs in three steps:

5098 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
5099 domains and dimensions are tested for compatibility.

5100 **Compute** The indicated computations are carried out.

5101 **Output** The result is written into the output matrix, possibly under control of a mask.

5102 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5103 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5104 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 5105 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5106 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
5107 compatibility as follows:

- 5108 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5109 must be from one of the pre-defined types of Table 3.2.
- 5110 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(u)$.
- 5111 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5112 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
5113 mulation operator.

5114 Two domains are compatible with each other if values from one domain can be cast to values in
5115 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5116 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5117 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
5118 error listed above is returned.

5119 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5120 1. $0 \leq \text{row_index} < \mathbf{nrows}(C)$

5121 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
5122 is returned.

5123 From the arguments, the internal vectors, mask, and index array used in the computation are
5124 formed (\leftarrow denotes copy):

5125 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a row of \mathbf{C} as follows:

$$5126 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5127 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

5128 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$.

5129 (b) If `mask \neq GrB_NULL`,

5130 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

5131 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

5132 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

5133 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5134 4. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:

5135 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.

5136 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

5137 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
5138 conditions must hold:

5139 1. $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5140 2. $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$.

5141 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
5142 match error listed above is returned.

5143 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5144 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5145 We are now ready to carry out the assign and any additional associated operations. We describe
5146 this in terms of two intermediate vectors:

- 5147 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 5148 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5149 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$5150 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5151 At this point, if any value of $\tilde{\mathbf{J}}[j]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
5152 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
5153 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
5154 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

5155 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5156 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}}$ is defined as

$$5157 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5158 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
5159 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
5160 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5161 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5162 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$5163 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5164 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5166 where the difference operator refers to set difference.

5167 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5168 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5169 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5170 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5171 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5172 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5173 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5176 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5177 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
5178 result matrix, $\mathbf{C}(\text{row_index}, :)$. This is carried out under control of the mask which acts as a “write
5179 mask”.

5180 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in $\mathbf{C}(\text{row_index}, :)$ on input to this
5181 operation are deleted and the new contents of the column is given by:

$$5182 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5183 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
5184 copied into the column of the final result matrix, $\mathbf{C}(\text{row_index}, :)$, and elements of this column
5185 that fall outside the set indicated by the mask are unchanged:

$$5186 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup$$

$$5187 \quad \{(\text{row_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5188 \quad \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5189 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
5190 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
5191 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but may
5192 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5193 **4.3.7.5 assign: Constant vector variant**[Scott: NEW CONTENT]

5194 Assign the same value to a specified subset of vector elements. With the use of GrB_ALL, the entire
5195 destination vector can be filled with the constant.

5196 **C Syntax**

```
5197         GrB_Info GrB_assign(GrB_Vector          w,  
5198                             const GrB_Vector    mask,  
5199                             const GrB_BinaryOp    accum,  
5200                             <type>              val,  
5201                             const GrB_Index      *indices,  
5202                             GrB_Index            nindices,  
5203                             const GrB_Descriptor desc);
```

```
5204         GrB_Info GrB_assign(GrB_Vector          w,  
5205                             const GrB_Vector    mask,  
5206                             const GrB_BinaryOp    accum,  
5207                             const GrB_Scalar      s,  
5208                             const GrB_Index      *indices,  
5209                             GrB_Index            nindices,  
5210                             const GrB_Descriptor desc);
```

5211 **Parameters**

5212 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5213 that may be accumulated with the result of the assign operation. On output, this
5214 vector holds the results of the operation.

5215 **mask** (IN) An optional “write” mask that controls which results from this operation are
5216 stored into the output vector w. The mask dimensions must match those of the
5217 vector w. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5218 of the mask vector must be of type bool or any of the predefined “built-in” types
5219 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5220 dimensions of w), GrB_NULL should be specified.

5221 **accum** (IN) An optional binary operator used for accumulating entries into existing w
5222 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5223 specified.

5224 **val** (IN) Scalar value to assign to (a subset of) w.

5225 **s** (IN) Scalar value to assign to (a subset of) w.

5226 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
5227 w that are to be assigned. If all elements of w are to be assigned in order from 0

5228 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode
5229 and return value, this array may be manipulated by the caller after this operation
5230 returns without affecting any deferred computations for this operation. In this
5231 variant, the specific order of the values in the array has no effect on the result.
5232 Unlike other variants, if there are duplicated values in this array the result is still
5233 defined.

5234 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If
5235 `nindices` is zero, the operation becomes a NO-OP.

5236 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
5237 should be specified. Non-default field/value pairs are listed as follows:

5238

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.
5239 <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> .

5240 Return Values

5241 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5242 blocking mode, this indicates that the compatibility tests on
5243 dimensions and domains for the input arguments passed suc-
5244 cessfully. Either way, output vector `w` is ready to be used in the
5245 next method of the sequence.

5246 **GrB_PANIC** Unknown internal error.

5247 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
5248 opaque GraphBLAS objects (input or output) is in an invalid
5249 state caused by a previous execution error. Call `GrB_error()` to
5250 access any error messages generated by the implementation.

5251 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5252 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized
5253 by a call to `new` (or `dup` for vector parameters).

5254 **GrB_INDEX_OUT_OF_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-
5255 blocking mode, this can be reported as an execution error.

5256 **GrB_DIMENSION_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less
5257 than `size(w)`.

5288 4. If **accum** is not **GrB_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the
 5289 method, must be compatible with **D_{in2}(accum)** of the accumulation operator.

5290 Two domains are compatible with each other if values from one domain can be cast to values in
 5291 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 5292 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5293 any compatibility rule above is violated, execution of **GrB_assign** ends and the domain mismatch
 5294 error listed above is returned.

5295 From the arguments, the internal vectors, mask and index array used in the computation are formed
 5296 (\leftarrow denotes copy):

- 5297 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5298 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument **mask** as follows:
 - 5299 (a) If **mask** = **GrB_NULL**, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 5300 (b) If **mask** \neq **GrB_NULL**,
 - 5301 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 5302 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 5303 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5304 3. Scalar $\tilde{s} \leftarrow s$ (**GrB_Scalar** version only).
- 5305 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument **indices** as follows:
 - 5306 (a) If **indices** = **GrB_ALL**, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$.
 - 5307 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$.

5308 The internal vector and mask are checked for dimension compatibility. The following conditions
 5309 must hold:

- 5310 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5311 2. $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$.

5312 If any compatibility rule above is violated, execution of **GrB_assign** ends and the dimension mis-
 5313 match error listed above is returned.

5314 From this point forward, in **GrB_NONBLOCKING** mode, the method can optionally exit with
 5315 **GrB_SUCCESS** return code and defer any computation and/or execution error codes.

5316 We are now ready to carry out the assign and any additional associated operations. We describe
 5317 this in terms of two intermediate vectors:

- 5318 • $\tilde{\mathbf{t}}$: The vector holding the copies of the scalar, either **val** or \tilde{s} , in their destination locations
 5319 relative to $\tilde{\mathbf{w}}$.

5320 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5321 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows. If a non-opaque scalar \mathbf{val} is provided:

$$5322 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5323 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 1$):

$$5324 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5325 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 0$):

$$5326 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5327 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
 5328 is not in the range $[0, \mathbf{size}(\tilde{\mathbf{w}}))$, the execution of `GrB_assign` ends and the index out-of-bounds
 5329 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a
 5330 sequence-terminating `GrB_wait()` is called. Regardless, the result vector, \mathbf{w} , is invalid from this
 5331 point forward in the sequence.

5332 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5333 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$5334 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5335 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 5336 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
 5337 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5338 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5339 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5340 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5341 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5343 where the difference operator refers to set difference. We note that in this case of assigning
 5344 a constant, $\{\tilde{\mathbf{I}}[k], \forall k\}$ and $\mathbf{ind}(\tilde{\mathbf{t}})$ are identical.

5345 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5346 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5347 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5348 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5349 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$5350 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5351 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5354 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5355 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 5356 using what is called a *standard vector mask and replace*. This is carried out under control of the
 5357 mask which acts as a “write mask”.

- 5358 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 5359 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$5360 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5361 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 5362 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 5363 mask are unchanged:

$$5364 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5365 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 5366 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 5367 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 5368 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 5369 sequence.

5370 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5371 Assign the same value to a specified subset of matrix elements. With the use of GrB_ALL, the
 5372 entire destination matrix can be filled with the constant.

5373 C Syntax

```
5374      GrB_Info GrB_assign(GrB_Matrix      C,
5375                          const GrB_Matrix Mask,
5376                          const GrB_BinaryOp accum,
5377                          <type>         val,
5378                          const GrB_Index *row_indices,
5379                          GrB_Index      nrows,
5380                          const GrB_Index *col_indices,
5381                          GrB_Index      ncols,
5382                          const GrB_Descriptor desc);
```

```
5383      GrB_Info GrB_assign(GrB_Matrix      C,
5384                          const GrB_Matrix Mask,
5385                          const GrB_BinaryOp accum,
5386                          const GrB_Scalar s,
5387                          const GrB_Index *row_indices,
5388                          GrB_Index      nrows,
```

```

5389         const GrB_Index      *col_indices,
5390         GrB_Index             ncols,
5391         const GrB_Descriptor  desc);

```

5392 Parameters

5393 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5394 that may be accumulated with the result of the assign operation. On output, the
5395 matrix holds the results of the operation.

5396 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5397 stored into the output matrix **C**. The mask dimensions must match those of the
5398 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5399 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
5400 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5401 dimensions of **C**), **GrB_NULL** should be specified.

5402 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5403 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5404 specified.

5405 **val** (IN) Scalar value to assign to (a subset of) **C**.

5406 **s** (IN) Scalar value to assign to (a subset of) **C**.

5407 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
5408 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
5409 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
5410 this array may be manipulated by the caller after this operation returns without
5411 affecting any deferred computations for this operation. Unlike other variants, if
5412 there are duplicated values in this array the result is still defined.

5413 **nrows** (IN) The number of values in **row_indices** array. Must be in the range: [0, **nrows**(**C**)].
5414 If **nrows** is zero, the operation becomes a NO-OP.

5415 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**
5416 that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** – 1,
5417 then **GrB_ALL** should be specified. Regardless of execution mode and return value,
5418 this array may be manipulated by the caller after this operation returns without
5419 affecting any deferred computations for this operation. Unlike other variants, if
5420 there are duplicated values in this array the result is still defined.

5421 **ncols** (IN) The number of values in **col_indices** array. Must be in the range: [0, **ncols**(**C**)].
5422 If **ncols** is zero, the operation becomes a NO-OP.

5423 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5424 should be specified. Non-default field/value pairs are listed as follows:

5425

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.

Return Values

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 <code>GrB_error()</code> 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 <code>new</code> (or <code>dup</code> for vector parameters).
GrB_INDEX_OUT_OF_BOUNDS	A value in <code>row_indices</code> is greater than or equal to <code>nrows(C)</code> , or a value in <code>col_indices</code> is greater than or equal to <code>ncols(C)</code> . In non-blocking mode, this can be reported as an execution error.
GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, <code>nrows</code> is not less than <code>nrows(C)</code> , or <code>ncols</code> is not less than <code>ncols(C)</code> .
GrB_DOMAIN_MISMATCH	The domains of the matrix and scalar are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with <code>bool</code> (in the case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
GrB_NULL_POINTER	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code> is a NULL pointer, or both.

Description

This variant of `GrB_assign` computes the result of assigning a constant scalar value – either `val` or `s` – to locations in a destination GraphBLAS matrix: Either `C(row_indices, col_indices) = val`

5455 or $C(\text{row_indices}, \text{col_indices}) = s$ is performed. If an optional binary accumulation operator
 5456 (\odot) is provided, then either $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot \text{val}$ or
 5457 $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot s$ is performed. More explicitly, if a
 5458 non-opaque value val is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = \text{val}, \text{ or} \\ 5459 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5460 Correspondingly, if a `GrB_Scalar` s is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = s, \text{ or} \\ 5461 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5462 Logically, this operation occurs in three steps:

5463 Setup The internal vectors and mask used in the computation are formed and their domains
 5464 and dimensions are tested for compatibility.

5465 Compute The indicated computations are carried out.

5466 Output The result is written into the output matrix, possibly under control of a mask.

5467 Up to two argument matrices are used in the `GrB_assign` operation:

- 5468 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5469 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \text{nrows}(\text{Mask}), \text{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)

5470 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain
 5471 compatibility as follows:

- 5472 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 5473 must be from one of the pre-defined types of Table 3.2.
- 5474 2. $\mathbf{D}(C)$ must be compatible with either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5475 method.
- 5476 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5477 of the accumulation operator.
- 5478 4. If `accum` is not `GrB_NULL`, then either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5479 method, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch error listed above is returned.

From the arguments, the internal matrices, index arrays, and mask used in the computation are formed (\leftarrow denotes copy):

1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
 - (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - (b) If `Mask \neq GrB_NULL`,
 - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$.
 - (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
3. Scalar $\tilde{s} \leftarrow s$ (`GrB_Scalar` version only).
4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
 - (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
 - (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
 - (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
 - (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

The internal matrix and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
3. $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$.
4. $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$.

If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch error listed above is returned.

5512 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5513 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5514 We are now ready to carry out the assign and any additional associated operations. We describe
 5515 this in terms of two intermediate matrices:

- 5516 • $\tilde{\mathbf{T}}$: The matrix holding the copies of the scalar, either `val` or \tilde{s} , in their destination locations
 5517 relative to $\tilde{\mathbf{C}}$.
- 5518 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5519 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows. If a non-opaque scalar `val` is provided:

$$\begin{aligned} 5520 \quad \tilde{\mathbf{T}} = & \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5521 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 1):

$$\begin{aligned} 5522 \quad \tilde{\mathbf{T}} = & \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5523 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 0):

$$5524 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5525 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
 5526 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
 5527 $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-bounds error listed above is
 5528 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating
 5529 `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from this point forward in the
 5530 sequence.

5531 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 5532 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$\begin{aligned} 5533 \quad \tilde{\mathbf{Z}} = & \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5534 \quad & \{(i, j, Z_{ij}) \mid (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle. \end{aligned}$$

5535 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 5536 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
 5537 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

5538 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 5539 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 5540 \quad Z_{ij} = & \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5541 \quad & \\ 5542 \quad & \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \end{aligned}$$

5543 where the difference operator refers to set difference. We note that, in this particular case of
 5544 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.

5545 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5546 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5547 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5548 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5549 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

5550

$$5551 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5552

$$5553 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5554 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5555 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5556 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5557 mask which acts as a “write mask”.

5558 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5559 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5560 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5561 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
5562 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
5563 mask are unchanged:

$$5564 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5565 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5566 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5567 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
5568 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
5569 sequence.

5570 4.3.8 apply: Apply a function to the elements of an object

5571 Computes the transformation of the values of the elements of a vector or a matrix using a unary
5572 function, or a binary function where one argument is bound to a scalar.

5573 4.3.8.1 apply: Vector variant

5574 Computes the transformation of the values of the elements of a vector using a unary function.

C Syntax

```
GrB_Info GrB_apply(GrB_Vector      w,
                  const GrB_Vector  mask,
                  const GrB_BinaryOp accum,
                  const GrB_UnaryOp  op,
                  const GrB_Vector  u,
                  const GrB_Descriptor desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the apply operation. On output, this vector holds the results of the operation.

mask (IN) An optional “write” mask that controls which results from this operation are stored into the output vector **w**. The mask dimensions must match those of the vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain of the mask vector must be of type **bool** or any of the predefined “built-in” types in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the dimensions of **w**), **GrB_NULL** should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing **w** entries. If assignment rather than accumulation is desired, **GrB_NULL** should be specified.

op (IN) A unary operator applied to each element of input vector **u**.

u (IN) The GraphBLAS vector to which the unary function is applied.

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
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 .

Return Values

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.

5605 Either way, output vector w is ready to be used in the next method
 5606 of the sequence.

5607 **GrB_PANIC** Unknown internal error.

5608 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 5609 GraphBLAS objects (input or output) is in an invalid state caused
 5610 by a previous execution error. Call **GrB_error()** to access any error
 5611 messages generated by the implementation.

5612 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5613 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 5614 a call to **new** (or **dup** for vector parameters).

5615 **GrB_DIMENSION_MISMATCH** $mask$, w and/or u dimensions are incompatible.

5616 **GrB_DOMAIN_MISMATCH** The domains of the various vectors are incompatible with the corre-
 5617 sponding domains of the accumulation operator or unary function,
 5618 or the mask's domain is not compatible with **bool** (in the case where
 5619 $desc[GrB_MASK].GrB_STRUCTURE$ is not set).

5620 **Description**

5621 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
 5622 GraphBLAS vector: $w = f(u)$; or, if an optional binary accumulation operator (\odot) is provided,
 5623 $w = w \odot f(u)$.

5624 Logically, this operation occurs in three steps:

5625 **Setup** The internal vectors and mask used in the computation are formed and their domains
 5626 and dimensions are tested for compatibility.

5627 **Compute** The indicated computations are carried out.

5628 **Output** The result is written into the output vector, possibly under control of a mask.

5629 Up to three argument vectors are used in this **GrB_apply** operation:

- 5630 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5631 2. $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$ (optional)
- 5632 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5633 The argument vectors, unary operator and the accumulation operator (if provided) are tested for
 5634 domain compatibility as follows:

- 5635 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5636 must be from one of the pre-defined types of Table 3.2.
- 5637 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5638 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5639 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5640 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5641 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in}(\text{op})$.

5642 Two domains are compatible with each other if values from one domain can be cast to values in
5643 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5644 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5645 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
5646 error listed above is returned.

5647 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
5648 denotes copy):

- 5649 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5650 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 5651 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 5652 (b) If `mask` \neq `GrB_NULL`,
 - 5653 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 5654 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 5655 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5656 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5657 The internal vectors and masks are checked for dimension compatibility. The following conditions
5658 must hold:

- 5659 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5660 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

5661 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5662 error listed above is returned.

5663 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5664 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5665 We are now ready to carry out the apply and any additional associated operations. We describe
5666 this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$: The vector holding the result from applying the unary operator to the input vector $\tilde{\mathbf{u}}$.
- $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle,$$

where $f = \mathbf{f}(\text{op})$.

The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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 $\tilde{\mathbf{t}}$.

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{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 \mathbf{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.

4.3.8.2 apply: Matrix variant

Computes the transformation of the values of the elements of a matrix using a unary function.

C Syntax

```
GrB_Info GrB_apply(GrB_Matrix      C,
                  const GrB_Matrix  Mask,
                  const GrB_BinaryOp accum,
                  const GrB_UnaryOp  op,
                  const GrB_Matrix  A,
                  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.

Mask (IN) An optional “write” mask that controls which results from this operation are stored into the output matrix C. The mask dimensions must match those 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 `bool` or any of the predefined “built-in” types in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the dimensions of C), `GrB_NULL` should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, `GrB_NULL` should be specified.

op (IN) A unary operator applied to each element of input matrix A.

A (IN) The GraphBLAS matrix to which the unary function is applied.

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	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output matrix C 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 matrix. The stored values are not examined.
Mask	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of Mask .
A	<code>GrB_INP0</code>	<code>GrB_TRAN</code>	Use transpose of A for the operation.

5727 Return Values

5728 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 5729 blocking mode, this indicates that the compatibility tests on
 5730 dimensions and domains for the input arguments passed suc-
 5731 cessfully. Either way, output matrix **C** is ready to be used in the
 5732 next method of the sequence.

5733 **GrB_PANIC** Unknown internal error.

5734 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
 5735 opaque GraphBLAS objects (input or output) is in an invalid
 5736 state caused by a previous execution error. Call **GrB_error()** to
 5737 access any error messages generated by the implementation.

5738 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

5739 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized
 5740 by a call to **new** (or **Matrix_dup** for matrix parameters).

5741 **GrB_DIMENSION_MISMATCH** Mask and **C** dimensions are incompatible, **nrows** \neq **nrows**(**C**), or
 5742 **ncols** \neq **ncols**(**C**).

5743 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
 5744 corresponding domains of the accumulation operator or unary
 5745 function, or the mask's domain is not compatible with **bool** (in
 5746 the case where **desc**[**GrB_MASK**].**GrB_STRUCTURE** is not set).

5747 Description

5748 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
 5749 GraphBLAS matrix: $C = f(A)$; or, if an optional binary accumulation operator (\odot) is provided,
 5750 $C = C \odot f(A)$.

5751 Logically, this operation occurs in three steps:

5752 **Setup** The internal matrices and mask used in the computation are formed and their domains
 5753 and dimensions are tested for compatibility.

5754 **Compute** The indicated computations are carried out.

5755 **Output** The result is written into the output matrix, possibly under control of a mask.

5756 Up to three argument matrices are used in the **GrB_apply** operation:

- 5757 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5758 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)

5759 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5760 The argument matrices, unary operator and the accumulation operator (if provided) are tested for
5761 domain compatibility as follows:

- 5762 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\text{Mask})$
5763 must be from one of the pre-defined types of Table 3.2.
- 5764 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5765 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5766 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5767 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5768 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in}(\text{op})$ of the unary operator.

5769 Two domains are compatible with each other if values from one domain can be cast to values in
5770 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5771 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5772 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
5773 error listed above is returned.

5774 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
5775 are formed (\leftarrow denotes copy):

- 5776 1. Matrix $\tilde{C} \leftarrow C$.
- 5777 2. Two-dimensional mask, \tilde{M} , is computed from argument **Mask** as follows:
 - 5778 (a) If **Mask** = **GrB_NULL**, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
5779 $j < \mathbf{ncols}(C)\} \rangle$.
 - 5780 (b) If **Mask** \neq **GrB_NULL**,
 - 5781 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
5782 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - 5783 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
5784 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - 5785 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
- 5786 3. Matrix $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$.

5787 The internal matrices and mask are checked for dimension compatibility. The following conditions
5788 must hold:

- 5789 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.
- 5790 2. $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$.
- 5791 3. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$.

5792 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

5793 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5794 error listed above is returned.

5795 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5796 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5797 We are now ready to carry out the apply and any additional associated operations. We describe
5798 this in terms of two intermediate matrices:

- 5799 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the unary operator to the input matrix $\tilde{\mathbf{A}}$.
- 5800 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5801 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$5802 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5803 where $f = \mathbf{f}(\mathbf{op})$.

5804 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 5805 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 5806 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5807 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5808 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5809 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5810 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5811 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5812 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5813 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5816 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5817 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5818 mask which acts as a “write mask”.

- 5819 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5820 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5821 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 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, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\text{ind}(\mathbf{C}) \cap \text{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\text{ind}(\tilde{\mathbf{Z}}) \cap \text{ind}(\tilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix \mathbf{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 \mathbf{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[Scott: NEW CONTENT]

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

```

5839 // bind-first + scalar value
5840 GrB_Info GrB_apply(GrB_Vector          w,
5841                   const GrB_Vector      mask,
5842                   const GrB_BinaryOp     accum,
5843                   const GrB_BinaryOp     op,
5844                   <type>                 val,
5845                   const GrB_Vector      u,
5846                   const GrB_Descriptor   desc);

5847 // bind-first + GraphBLAS scalar
5848 GrB_Info GrB_apply(GrB_Vector          w,
5849                   const GrB_Vector      mask,
5850                   const GrB_BinaryOp     accum,
5851                   const GrB_BinaryOp     op,
5852                   const GrB_Scalar       s,
5853                   const GrB_Vector      u,
5854                   const GrB_Descriptor   desc);

5855 // bind-second + scalar value
5856 GrB_Info GrB_apply(GrB_Vector          w,
5857                   const GrB_Vector      mask,
```

```

5858             const GrB_BinaryOp      accum,
5859             const GrB_BinaryOp      op,
5860             const GrB_Vector        u,
5861             <type>                  val,
5862             const GrB_Descriptor     desc);

5863 // bind-second + GraphBLAS scalar
5864 GrB_Info GrB_apply(GrB_Vector        w,
5865                   const GrB_Vector    mask,
5866                   const GrB_BinaryOp  accum,
5867                   const GrB_BinaryOp  op,
5868                   const GrB_Vector    u,
5869                   const GrB_Scalar     s,
5870                   const GrB_Descriptor desc);

```

5871 Parameters

5872 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5873 that may be accumulated with the result of the apply operation. On output, this
5874 vector holds the results of the operation.

5875 **mask** (IN) An optional “write” mask that controls which results from this operation are
5876 stored into the output vector **w**. The mask dimensions must match those of the
5877 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5878 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
5879 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5880 dimensions of **w**), **GrB_NULL** should be specified.

5881 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5882 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5883 specified.

5884 **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar
5885 value, **val**.

5886 **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as
5887 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
5888 argument in the *bind-second* variant.

5889 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
5890 argument in the *bind-first* variant, or the right-hand (second) argument in the
5891 *bind-second* variant.

5892 **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand
5893 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
5894 the *bind-second* variant. It must not be empty.

5895 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 5896 should be specified. Non-default field/value pairs are listed as follows:

5897

	Param	Field	Value	Description
	w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
5898	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 .

5899 Return Values

5900 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 5901 blocking mode, this indicates that the compatibility tests on di-
 5902 mensions and domains for the input arguments passed successfully.
 5903 Either way, output vector **w** is ready to be used in the next method
 5904 of the sequence.

5905 GrB_PANIC Unknown internal error.

5906 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 5907 GraphBLAS objects (input or output) is in an invalid state caused
 5908 by a previous execution error. Call GrB_error() to access any error
 5909 messages generated by the implementation.

5910 GrB_OUT_OF_MEMORY Not enough memory available for operation.

5911 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 5912 a call to new (or dup for vector parameters).

5913 GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

5914 GrB_DOMAIN_MISMATCH The domains of the various vectors and scalar are incompatible with
 5915 the corresponding domains of the binary operator or accumulation
 5916 operator, or the mask's domain is not compatible with bool (in the
 5917 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

5918 GrB_EMPTY_OBJECT The GrB_Scalar **s** used in the call is empty (**nvals(s) = 0**) and
 5919 therefore a value cannot be passed to the binary operator.

5920 Description

5921 This variant of GrB_apply computes the result of applying a binary operator to the elements of a
 5922 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5923 bind-first: $w = f(\text{val}, u)$ or $w = f(s, u)$

5924 bind-second: $w = f(u, \text{val})$ or $w = f(u, s)$,

5925 or if an optional binary accumulation operator (\odot) is provided:

5926 bind-first: $w = w \odot f(\text{val}, u)$ or $w = w \odot f(s, u)$

5927 bind-second: $w = w \odot f(u, \text{val})$ or $w = w \odot f(u, s)$.

5928 Logically, this operation occurs in three steps:

5929 **Setup** The internal vectors and mask used in the computation are formed and their domains
5930 and dimensions are tested for compatibility.

5931 **Compute** The indicated computations are carried out.

5932 **Output** The result is written into the output vector, possibly under control of a mask.

5933 Up to three argument vectors are used in this GrB_apply operation:

- 5934 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
5935 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
5936 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5937 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are
5938 tested for domain compatibility as follows:

- 5939 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
5940 must be from one of the pre-defined types of Table 3.2.
- 5941 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.
- 5942 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5943 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
5944 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5945 4. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.
- 5946 5. If bind-first:
- 5947 (a) $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.
- 5948 (b) If the non-opaque scalar **val** is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
5949 of the binary operator.
- 5950 (c) If the GrB_Scalar **s** is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
5951 binary operator.

- 5952 6. If bind-second:
- 5953 (a) $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the binary operator.
- 5954 (b) If the non-opaque scalar \mathbf{val} is provided, then $\mathbf{D}(\mathbf{val})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
- 5955 of the binary operator.
- 5956 (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
- 5957 binary operator.

5958 Two domains are compatible with each other if values from one domain can be cast to values in

5959 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5960 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5961 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5962 error listed above is returned.

5963 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow

5964 denotes copy):

- 5965 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5966 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 5967 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 5968 (b) If `mask \neq GrB_NULL`,
- 5969 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 5970 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 5971 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5972 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 5973 4. Scalar $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$ (GraphBLAS scalar case).

5974 The internal vectors and masks are checked for dimension compatibility. The following conditions

5975 must hold:

- 5976 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5977 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

5978 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5979 error listed above is returned.

5980 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5981 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5982 If an empty `GrB_Scalar` $\tilde{\mathbf{s}}$ is provided ($\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

5983 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

5984 `val` with the same domain as $\tilde{\mathbf{s}}$ and set `val = val($\tilde{\mathbf{s}}$)`.

5985 We are now ready to carry out the apply and any additional associated operations. We describe

5986 this in terms of two intermediate vectors:

- 5987 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the binary operator to the input vector $\tilde{\mathbf{u}}$.
- 5988 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5989 The intermediate vector, $\tilde{\mathbf{t}}$, is created as one of the following:

$$\begin{aligned}
 5990 \quad \text{bind-first:} \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\text{val}, \tilde{\mathbf{u}}(i))) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \\
 5991 \quad \text{bind-second:} \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i), \text{val})) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,
 \end{aligned}$$

5992 where $f = \mathbf{f}(\text{op})$.

5993 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 5994 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 5995 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5996 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5997 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5998 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned}
 5999 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\
 6000 \\
 6001 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 6002 \\
 6003 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),
 \end{aligned}$$

6004 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6005 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 6006 using what is called a *standard vector mask and replace*. This is carried out under control of the
 6007 mask which acts as a “write mask”.

- 6008 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 6009 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$6010 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 6011 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 6012 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 6013 mask are unchanged:

$$6014 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

6015 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 6016 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 6017 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
 6018 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 6019 sequence.

6020 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

6021 Computes the transformation of the values of the stored elements of a matrix using a binary
6022 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the
6023 first argument to the binary operator and stored elements of the matrix are passed as the second
6024 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument
6025 and the specified scalar value is passed as the second argument. The scalar can be passed either as
6026 a non-opaque variable or as a GrB_Scalar object.

6027 C Syntax

```
6028 // bind-first + scalar value
6029 GrB_Info GrB_apply(GrB_Matrix      C,
6030                   const GrB_Matrix  Mask,
6031                   const GrB_BinaryOp accum,
6032                   const GrB_BinaryOp op,
6033                   <type>            val,
6034                   const GrB_Matrix  A,
6035                   const GrB_Descriptor desc);
```

```
6036 // bind-first + GraphBLAS scalar
6037 GrB_Info GrB_apply(GrB_Matrix      C,
6038                   const GrB_Matrix  Mask,
6039                   const GrB_BinaryOp accum,
6040                   const GrB_BinaryOp op,
6041                   const GrB_Scalar   s,
6042                   const GrB_Matrix  A,
6043                   const GrB_Descriptor desc);
```

```
6044 // bind-second + scalar value
6045 GrB_Info GrB_apply(GrB_Matrix      C,
6046                   const GrB_Matrix  Mask,
6047                   const GrB_BinaryOp accum,
6048                   const GrB_BinaryOp op,
6049                   const GrB_Matrix  A,
6050                   <type>            val,
6051                   const GrB_Descriptor desc);
```

```
6052 // bind-second + GraphBLAS scalar
6053 GrB_Info GrB_apply(GrB_Matrix      C,
6054                   const GrB_Matrix  Mask,
6055                   const GrB_BinaryOp accum,
6056                   const GrB_BinaryOp op,
6057                   const GrB_Matrix  A,
```

```

6058         const GrB_Scalar      s,
6059         const GrB_Descriptor desc);

```

6060 Parameters

6061 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6062 that may be accumulated with the result of the apply operation. On output, the
6063 matrix holds the results of the operation.

6064 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6065 stored into the output matrix C. The mask dimensions must match those of the
6066 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
6067 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
6068 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
6069 dimensions of C), `GrB_NULL` should be specified.

6070 **accum** (IN) An optional binary operator used for accumulating entries into existing C
6071 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
6072 specified.

6073 **op** (IN) A binary operator applied to each element of input matrix, A, with the element
6074 of the input matrix used as the left-hand argument, and the scalar value, `val`, used
6075 as the right-hand argument.

6076 **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as
6077 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
6078 argument in the *bind-second* variant.

6079 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
6080 argument in the *bind-first* variant, or the right-hand (second) argument in the
6081 *bind-second* variant.

6082 **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand
6083 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
6084 the *bind-second* variant. It must not be empty.

6085 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
6086 should be specified. Non-default field/value pairs are listed as follows:

6087

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).

Return Values

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_INDEX_OUT_OF_BOUNDS	A value in row_indices is greater than or equal to nrows(A), or a value in col_indices is greater than or equal to ncols(A). In non-blocking mode, this can be reported as an execution error.
GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, nrows \neq nrows(C), or ncols \neq ncols(C).
GrB_DOMAIN_MISMATCH	The domains of the various matrices 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).
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.

6115 Description

6116 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a
 6117 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6118 bind-first: $C = f(\text{val}, A)$ or $C = f(s, A)$

6119 bind-second: $C = f(A, \text{val})$ or $C = f(A, s)$,

6120 or if an optional binary accumulation operator (\odot) is provided:

6121 bind-first: $C = C \odot f(\text{val}, A)$ or $C = C \odot f(s, A)$

6122 bind-second: $C = C \odot f(A, \text{val})$ or $C = C \odot f(A, s)$.

6123 Logically, this operation occurs in three steps:

6124 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6125 and dimensions are tested for compatibility.

6126 **Compute** The indicated computations are carried out.

6127 **Output** The result is written into the output matrix, possibly under control of a mask.

6128 Up to three argument matrices are used in the `GrB_apply` operation:

- 6129 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6130 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)
- 6131 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6132 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are
 6133 tested for domain compatibility as follows:

- 6134 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 6135 must be from one of the pre-defined types of Table 3.2.
- 6136 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.
- 6137 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6138 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
 6139 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6140 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.
- 6141 5. If bind-first:
 6142 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

6143 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
 6144 of the binary operator.

6145 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
 6146 binary operator.

6147 6. If `bind-second`:

6148 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

6149 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$
 6150 of the binary operator.

6151 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the
 6152 binary operator.

6153 Two domains are compatible with each other if values from one domain can be cast to values in
 6154 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6155 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6156 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
 6157 error listed above is returned.

6158 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
 6159 are formed (\leftarrow denotes copy):

6160 1. Matrix $\tilde{C} \leftarrow C$.

6161 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:

6162 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
 6163 $j < \mathbf{ncols}(C)\} \rangle$.

6164 (b) If `Mask` \neq `GrB_NULL`,

6165 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
 6166 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,

6167 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
 6168 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.

6169 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.

6170 3. Matrix \tilde{A} is computed from argument `A` as follows:

6171 `bind-first:` $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? A^T : A$

6172 `bind-second:` $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$

6173 4. Scalar $\tilde{s} \leftarrow s$ (`GraphBLAS` scalar case).

6174 The internal matrices and mask are checked for dimension compatibility. The following conditions
 6175 must hold:

6176 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.

6177 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.

6178 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6179 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6180 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6181 error listed above is returned.

6182 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6183 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6184 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6185 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6186 \mathbf{val} with the same domain as \tilde{s} and set $\mathbf{val} = \mathbf{val}(\tilde{s})$.

6187 We are now ready to carry out the apply and any additional associated operations. We describe
6188 this in terms of two intermediate matrices:

- 6189 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the binary operator to the input matrix $\tilde{\mathbf{A}}$.
- 6190 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6191 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as one of the following:

6192 bind-first: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$,

6193 bind-second: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$,

6194 where $f = \mathbf{f}(\mathbf{op})$.

6195 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6196 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6197 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6198 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6199 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6200 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6201 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6202 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6203 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6206 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6207 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6208 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6209 mask which acts as a “write mask”.

- 6210 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
6211 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6212 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6213 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6214 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6215 mask are unchanged:

$$6216 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6217 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
6218 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
6219 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
6220 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6221 sequence.

6222 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6223 Computes the transformation of the values of the stored elements of a vector using an index unary
6224 operator that is a function of the stored value, its location indices, and an user provided scalar
6225 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6226 C Syntax

```
6227     GrB_Info GrB_apply(GrB_Vector          w,
6228                       const GrB_Vector    mask,
6229                       const GrB_BinaryOp   accum,
6230                       const GrB_IndexUnaryOp op,
6231                       const GrB_Vector    u,
6232                       <type>              val,
6233                       const GrB_Descriptor desc);
```

```
6234     GrB_Info GrB_apply(GrB_Vector          w,
6235                       const GrB_Vector    mask,
6236                       const GrB_BinaryOp   accum,
6237                       const GrB_IndexUnaryOp op,
6238                       const GrB_Vector    u,
6239                       const GrB_Scalar    s,
6240                       const GrB_Descriptor desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the apply operation. On output, this vector holds the results of the operation.

mask (IN) An optional “write” mask that controls which results from this operation are stored into the output vector **w**. The mask dimensions must match those of the vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain of the **mask** vector must be of type **bool** or any of the predefined “built-in” types in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the dimensions of **w**), **GrB_NULL** should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing **w** entries. If assignment rather than accumulation is desired, **GrB_NULL** should be specified.

op (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\mathbf{GrB_Index}), D_{in_2}, f_i \rangle$, applied to each element stored in the input vector, **u**. It is a function of the stored element’s value, its location index, and a user supplied scalar value (either **s** or **val**).

u (IN) The GraphBLAS vector whose elements are passed to the index unary operator.

val (IN) An additional scalar value that is passed to the index unary operator.

s (IN) An additional GraphBLAS scalar that is passed to the index unary operator. It must not be empty.

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
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 .

Return Values

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.

6272 GrB_PANIC Unknown internal error.

6273 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6274 opaque GraphBLAS objects (input or output) is in an invalid
6275 state caused by a previous execution error. Call `GrB_error()` to
6276 access any error messages generated by the implementation.

6277 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6278 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6279 by a call to `new` (or another constructor).

6280 GrB_DIMENSION_MISMATCH `mask`, `w` and/or `u` dimensions are incompatible.

6281 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
6282 responding domains of the accumulation operator or index unary
6283 operator, or the mask's domain is not compatible with `bool` (in
6284 the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6285 GrB_EMPTY_OBJECT The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and
6286 therefore a value cannot be passed to the index unary operator.

6287 Description

6288 This variant of `GrB_apply` computes the result of applying an index unary operator to the elements
6289 of a GraphBLAS vector each composed with the element's index and a scalar constant, `val` or `s`:

$$6290 \qquad w = f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = f_i(u, \mathbf{ind}(u), 0, s),$$

6291 or if an optional binary accumulation operator (\odot) is provided:

$$6292 \qquad w = w \odot f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = w \odot f_i(u, \mathbf{ind}(u), 0, s).$$

6293 Logically, this operation occurs in three steps:

6294 **Setup** The internal vectors and mask used in the computation are formed and their domains
6295 and dimensions are tested for compatibility.

6296 **Compute** The indicated computations are carried out.

6297 **Output** The result is written into the output vector, possibly under control of a mask.

6298 Up to three argument vectors are used in this `GrB_apply` operation:

- 6299 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6300 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

6301 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6302 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6303 are tested for domain compatibility as follows:

- 6304 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
6305 must be from one of the pre-defined types of Table 3.2.
- 6306 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the index unary operator.
- 6307 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6308 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be compatible
6309 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6310 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6311 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
6312 the index unary operator.
- 6313 6. If the `GrB_Scalar s` is provided, then $\mathbf{D}(\mathbf{s})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the index
6314 unary operator.

6315 Two domains are compatible with each other if values from one domain can be cast to values in
6316 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6317 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6318 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
6319 error listed above is returned.

6320 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6321 denotes copy):

- 6322 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6323 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 6324 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 6325 (b) If `mask \neq GrB_NULL`,
 - 6326 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,
 - 6327 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 6328 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6329 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 6330 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6331 The internal vectors and masks are checked for dimension compatibility. The following conditions
6332 must hold:

6333 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6334 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

6335 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6336 error listed above is returned.

6337 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6338 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6339 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6340 If a non-empty `GrB_Scalar`, \tilde{s} , is provided ($\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable `val`
6341 with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6342 We are now ready to carry out the apply and any additional associated operations. We describe
6343 this in terms of two intermediate vectors:

- 6344 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6345 $\tilde{\mathbf{u}}$.
- 6346 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6347 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6348 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6349 where $f_i = \mathbf{f}(\text{op})$.

6350 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6351 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 6352 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6353 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6354 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6355 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6356 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6357 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6358 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6359 \quad & \\ 6360 \end{aligned}$$

6361 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6362 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
6363 using what is called a *standard vector mask and replace*. This is carried out under control of the
6364 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,

$$L(w) = \{(i, z_i) : i \in (\text{ind}(\tilde{z}) \cap \text{ind}(\tilde{m}))\}.$$

- 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:

$$L(w) = \{(i, w_i) : i \in (\text{ind}(w) \cap \text{ind}(\neg\tilde{m}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{z}) \cap \text{ind}(\tilde{m}))\}.$$

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.

6377 4.3.8.6 apply: Matrix index unary operator variant[Scott: NEW CONTENT]

6378 Computes the transformation of the values of the stored elements of a matrix using an index unary
6379 operator that is a function of the stored value, its location indices, and an user provided scalar
6380 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6381 C Syntax

```
6382     GrB_Info GrB_apply(GrB_Matrix      C,
6383                       const GrB_Matrix Mask,
6384                       const GrB_BinaryOp accum,
6385                       const GrB_IndexUnaryOp op,
6386                       const GrB_Matrix A,
6387                       <type>          val,
6388                       const GrB_Descriptor desc);
```

```
6389     GrB_Info GrB_apply(GrB_Matrix      C,
6390                       const GrB_Matrix Mask,
6391                       const GrB_BinaryOp accum,
6392                       const GrB_IndexUnaryOp op,
6393                       const GrB_Matrix A,
6394                       const GrB_Scalar s,
6395                       const GrB_Descriptor desc);
```

6396 Parameters

6397 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6398 that may be accumulated with the result of the apply operation. On output, the
6399 matrix holds the results of the operation.

6400 Mask (IN) An optional “write” mask that controls which results from this operation are
6401 stored into the output matrix C. The mask dimensions must match those of the
6402 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
6403 of the Mask matrix must be of type **bool** or any of the predefined “built-in” types
6404 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6405 dimensions of C), GrB_NULL should be specified.

6406 accum (IN) An optional binary operator used for accumulating entries into existing C
6407 entries. If assignment rather than accumulation is desired, GrB_NULL should be
6408 specified.

6409 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
6410 to each element stored in the input matrix, A. It is a function of the stored element’s
6411 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6412 A (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6413 ator.

6414 val (IN) An additional scalar value that is passed to the index unary operator.

6415 s (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6416 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
6417 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.

6420 Return Values

6421 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
6422 blocking mode, this indicates that the compatibility tests on di-
6423 mensions and domains for the input arguments passed successfully.
6424 Either way, output matrix C is ready to be used in the next method
6425 of the sequence.

6426 GrB_PANIC Unknown internal error.

6427 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
6428 GraphBLAS objects (input or output) is in an invalid state caused

6429 by a previous execution error. Call `GrB_error()` to access any error
 6430 messages generated by the implementation.

6431 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

6432 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 6433 a call to `new` (or another constructor).

6434 **GrB_DIMENSION_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6435 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
 6436 corresponding domains of the accumulation operator or index unary
 6437 operator, or the mask's domain is not compatible with `bool` (in the
 6438 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6439 **GrB_EMPTY_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and
 6440 therefore a value cannot be passed to the index unary operator.

6441 Description

6442 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements
 6443 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar
 6444 constant, `val` or `s`:

$$6445 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6446 or if an optional binary accumulation operator (\odot) is provided:

$$6447 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6448 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
 6449 indices, respectively.

6450 Logically, this operation occurs in three steps:

6451 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6452 and dimensions are tested for compatibility.

6453 **Compute** The indicated computations are carried out.

6454 **Output** The result is written into the output matrix, possibly under control of a mask.

6455 Up to three argument matrices are used in the `GrB_apply` operation:

- 6456 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6457 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)

6458 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6459 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6460 are tested for domain compatibility as follows:

- 6461 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\mathbf{Mask})$
6462 must be from one of the pre-defined types of Table 3.2.
- 6463 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator.
- 6464 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6465 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be compatible
6466 with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accumulation operator.
- 6467 4. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
- 6468 5. If the non-opaque scalar **val** is provided, then $\mathbf{D}(\mathbf{val})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of
6469 the index unary operator.
- 6470 6. If the **GrB_Scalar** **s** is provided, then $\mathbf{D}(\mathbf{s})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the index
6471 unary operator.

6472 Two domains are compatible with each other if values from one domain can be cast to values in
6473 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6474 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6475 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
6476 error listed above is returned.

6477 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation
6478 are formed (\leftarrow denotes copy):

- 6479 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 6480 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument **Mask** as follows:
 - 6481 (a) If **Mask** = **GrB_NULL**, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
6482 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 6483 (b) If **Mask** \neq **GrB_NULL**,
 - 6484 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
6485 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 6486 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
6487 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 6488 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 6489 3. Matrix $\tilde{\mathbf{A}}$ is computed from argument **A** as follows:

$$6490 \quad \tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$$
- 6491 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6492 The internal matrices and mask are checked for dimension compatibility. The following conditions
6493 must hold:

- 6494 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 6495 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 6496 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 6497 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6498 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6499 error listed above is returned.

6500 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6501 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6502 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6503 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6504 \mathbf{val} with the same domain as \tilde{s} and set $\mathbf{val} = \mathbf{val}(\tilde{s})$.

6505 We are now ready to carry out the apply and any additional associated operations. We describe
6506 this in terms of two intermediate matrices:

- 6507 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6508 $\tilde{\mathbf{A}}$.
- 6509 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6510 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6511 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6512 where $f_i = \mathbf{f}(\mathbf{op})$.

6513 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6514 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6515 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6516 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6517 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6518 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 6519 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6520 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6521 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6522 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6523 \end{aligned}$$

6524 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6525 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6526 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6527 mask which acts as a “write mask”.

- 6528 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
6529 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6530 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6531 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6532 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6533 mask are unchanged:

$$6534 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6535 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
6536 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
6537 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
6538 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6539 sequence.

6540 4.3.9 select:

6541 Apply a select operator to the stored elements of an object to determine whether or not to keep
6542 them.

6543 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6544 Apply a select operator (an index unary operator) to the elements of a vector.

6545 C Syntax

```
6546 // scalar value variant
6547 GrB_Info GrB_select(GrB_Vector          w,
6548                    const GrB_Vector     mask,
6549                    const GrB_BinaryOp   accum,
6550                    const GrB_IndexUnaryOp op,
6551                    const GrB_Vector     u,
6552                    <type>               val,
6553                    const GrB_Descriptor desc);
6554
6555 // GraphBLAS scalar variant
6556 GrB_Info GrB_select(GrB_Vector          w,
6557                    const GrB_Vector     mask,
```

```

6558         const GrB_BinaryOp      accum,
6559         const GrB_IndexUnaryOp  op,
6560         const GrB_Vector        u,
6561         const GrB_Scalar        s,
6562         const GrB_Descriptor    desc);
6563

```

6564 Parameters

6565 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6566 that may be accumulated with the result of the select operation. On output, this
6567 vector 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 vector **w**. The mask dimensions must match those of the
6570 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6571 of the **mask** vector 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 **w**), **GrB_NULL** should be specified.

6574 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6575 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6576 specified.

6577 **op** (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\mathbf{GrB_Index}), D_{in_2}, f_i \rangle$, applied
6578 to each element stored in the input vector, **u**. It is a function of the stored element’s
6579 value, its location index, and a user supplied scalar value (either **s** or **val**).

6580 **u** (IN) The GraphBLAS vector 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:

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 .

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 success-
 6593 fully. Either way, output vector **w** is ready to be used in the next
 6594 method of the sequence.

6595 **GrB_PANIC** Unknown internal error.

6596 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
 6597 opaque GraphBLAS objects (input or output) is in an invalid
 6598 state caused by a previous execution error. Call **GrB_error()** to
 6599 access any error 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
 6602 by a call to one of its constructors.

6603 **GrB_DIMENSION_MISMATCH** **mask**, **w** and/or **u** dimensions are incompatible.

6604 **GrB_DOMAIN_MISMATCH** The domains of the various vectors are incompatible with the cor-
 6605 responding domains of the accumulation operator or index unary
 6606 operator, or the mask's domain is not compatible with **bool** (in
 6607 the 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 vector. The operator takes, as input, the value of each stored
 6613 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding
 6614 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.
 6615 This acts like a functional mask on the input vector as follows:

$$6616 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6617 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6618 Correspondingly, if a **GrB_Scalar s**, is provided:

$$6619 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6620 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6621 Logically, this operation occurs in three steps:

6622 **Setup** The internal vectors and mask used in the computation are formed and their domains
6623 and dimensions are tested for compatibility.

6624 **Compute** The indicated computations are carried out.

6625 **Output** The result is written into the output vector, possibly under control of a mask.

6626 Up to three argument vectors are used in this `GrB_select` operation:

- 6627 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6628 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 6629 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6630 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6631 are tested for domain compatibility as follows:

- 6632 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
6633 must be from one of the pre-defined types of Table 3.2.
- 6634 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 6635 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6636 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
6637 mulation operator.
- 6638 4. $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
6639 i.e., castable to `bool`.
- 6640 5. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
- 6641 6. $\mathbf{D}(\mathbf{val})$ or $\mathbf{D}(\mathbf{s})$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
6642 of the index unary operator.

6643 Two domains are compatible with each other if values from one domain can be cast to values in
6644 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6645 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6646 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch
6647 error listed above is returned.

6648 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6649 denotes copy):

- 6650 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6651 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

6652 (a) If $\text{mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
6653 (b) If $\text{mask} \neq \text{GrB_NULL}$,
6654 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
6655 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
6656 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
6657 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
6658 4. Scalar $\widetilde{s} \leftarrow s$ (GrB_Scalar version only).

6659 The internal vectors and masks are checked for dimension compatibility. The following conditions
6660 must hold:

- 6661 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6662 2. $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$.

6663 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch
6664 error listed above is returned.

6665 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6666 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6667 If an empty `GrB_Scalar` \widetilde{s} is provided (i.e., $\text{nvals}(\widetilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6668 If a non-empty `GrB_Scalar`, \widetilde{s} , is provided (i.e., $\text{nvals}(\widetilde{s}) = 1$), we then create an internal variable
6669 `val` with the same domain as \widetilde{s} and set $\text{val} = \text{val}(\widetilde{s})$.

6670 We are now ready to carry out the `select` and any additional associated operations. We describe
6671 this in terms of two intermediate vectors:

- 6672 • $\widetilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6673 $\widetilde{\mathbf{u}}$.
- 6674 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6675 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$6676 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6677 where $f_i = \mathbf{f}(\text{op})$.

6678 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6679 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 6680 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$6681 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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 $\tilde{\mathbf{t}}$.

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector \mathbf{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 \mathbf{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.

4.3.9.2 select: Matrix variant[Scott: NEW CONTENT]

Apply a select operator (an index unary operator) to the elements of a matrix.

C Syntax

```
// scalar value variant
GrB_Info GrB_select(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);
```

```

6717 // GraphBLAS scalar variant
6718 GrB_Info GrB_select(GrB_Matrix          C,
6719                    const GrB_Matrix     Mask,
6720                    const GrB_BinaryOp    accum,
6721                    const GrB_IndexUnaryOp op,
6722                    const GrB_Matrix      A,
6723                    const GrB_Scalar      s,
6724                    const GrB_Descriptor  desc);

```

6725 Parameters

6726 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6727 that may be accumulated with the result of the select operation. On output, the
6728 matrix holds the results of the operation.

6729 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6730 stored into the output matrix **C**. The mask dimensions must match those of the
6731 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6732 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
6733 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6734 dimensions of **C**), **GrB_NULL** should be specified.

6735 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
6736 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6737 specified.

6738 **op** (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\mathbf{GrB_Index}), D_{in_2}, f_i \rangle$, applied
6739 to each element stored in the input matrix, **A**. It is a function of the stored element’s
6740 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6741 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6742 ator.

6743 **val** (IN) An additional scalar value that is passed to the index unary operator.

6744 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6745 not be empty.

6746 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6747 should be specified. Non-default field/value pairs are listed as follows:

6748

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.

Return Values

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 operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to one of its constructors.

GrB_DIMENSION_MISMATCH Mask, C and/or A dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the corresponding domains of the accumulation operator or index unary operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB_MASK].GrB_STRUCTURE** is not set).

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 index unary operator.

Description

This variant of **GrB_select** computes the result of applying a index unary operator to select the elements of the input GraphBLAS matrix. The operator takes, as input, the value of each stored element, along with the element's row and column indices and a scalar constant – from either **val** or **s**. The corresponding element of the input matrix is selected (kept) if the function evaluates to **true** when cast to **bool**. This acts like a functional mask on the input matrix as follows when specifying a transparent scalar value:

6778 $C = A \langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \rangle$, or
6779 $C = C \odot A \langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \rangle$.

6780 Correspondingly, if a GrB_Scalar, s, is provided:

6781 $C = A \langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s) \rangle$, or
6782 $C = C \odot A \langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s) \rangle$.

6783 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
6784 indices, respectively.

6785 Logically, this operation occurs in three steps:

6786 **Setup** The internal matrices and mask used in the computation are formed and their domains
6787 and dimensions are tested for compatibility.

6788 **Compute** The indicated computations are carried out.

6789 **Output** The result is written into the output matrix, possibly under control of a mask.

6790 Up to three argument matrices are used in the GrB_select operation:

- 6791 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6792 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)
- 6793 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6794 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6795 are tested for domain compatibility as follows:

- 6796 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
6797 must be from one of the pre-defined types of Table 3.2.
- 6798 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 6799 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6800 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
6801 mulation operator.
- 6802 4. $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
6803 i.e., castable to bool.
- 6804 5. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6805 6. $\mathbf{D}(\mathbf{val})$ or $\mathbf{D}(s)$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\text{op})$
6806 of the index unary operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch error listed above is returned.

From the argument matrices, the internal matrices, mask, and index arrays used in the computation are formed (\leftarrow denotes copy):

1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - (b) If `Mask \neq GrB_NULL`,
 - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
3. Matrix $\tilde{\mathbf{A}}$ is computed from argument `A` as follows: $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$
4. Scalar $\tilde{s} \leftarrow s$ (`GrB_Scalar` version only).

The internal matrices and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch error listed above is returned.

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.

If an empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`. If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable `val` with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

We are now ready to carry out the `select` and any additional associated operations. We describe this in terms of two intermediate matrices:

- 6841 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6842 $\tilde{\mathbf{A}}$.
- 6843 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6844 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6845 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6846 where $f_i = \mathbf{f}(\text{op})$.

6847 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6848 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6849 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6850 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

6851 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6852 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6853 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6854 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6855 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6856 \quad 6857$$

6858 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6859 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6860 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6861 mask which acts as a “write mask”.

- 6862 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
6863 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6864 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6865 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6866 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6867 mask are unchanged:

$$6868 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6869 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6870 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6871 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
6872 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6873 sequence.

6874 4.3.10 reduce: Perform a reduction across the elements of an object

6875 Computes the reduction of the values of the elements of a vector or matrix.

6876 4.3.10.1 reduce: Standard matrix to vector variant

6877 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns
6878 is desired, the input matrix should be transposed using the descriptor.

6879 C Syntax

```
6880         GrB_Info GrB_reduce(GrB_Vector          w,  
6881                             const GrB_Vector    mask,  
6882                             const GrB_BinaryOp   accum,  
6883                             const GrB_Monoid     op,  
6884                             const GrB_Matrix    A,  
6885                             const GrB_Descriptor desc);  
6886  
6887         GrB_Info GrB_reduce(GrB_Vector          w,  
6888                             const GrB_Vector    mask,  
6889                             const GrB_BinaryOp   accum,  
6890                             const GrB_BinaryOp   op,  
6891                             const GrB_Matrix    A,  
6892                             const GrB_Descriptor desc);
```

6893 Parameters

6894 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6895 that may be accumulated with the result of the reduction operation. On output,
6896 this vector holds the results of the operation.

6897 **mask** (IN) An optional “write” mask that controls which results from this operation are
6898 stored into the output vector **w**. The mask dimensions must match those of the
6899 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6900 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6901 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6902 dimensions of **w**), **GrB_NULL** should be specified.

6903 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6904 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6905 specified.

6906 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.
6907 Depending on which type is passed, the following defines the binary operator with
6908 one domain, $F_b = \langle D, D, D, \oplus \rangle$, that is used:

6909 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 6910 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
 6911 monoid is ignored.

6912 If op is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in
 6913 both cases $\odot(\text{op})$ must be commutative and associative. Otherwise, the outcome
 6914 of the operation is undefined.

6915 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

6916 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 6917 should be specified. Non-default field/value pairs are listed as follows:
 6918

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 .
A	<code>GrB_INP0</code>	<code>GrB_TRAN</code>	Use transpose of A for the operation.

6920 Return Values

6921 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 6922 blocking mode, this indicates that the compatibility tests on di-
 6923 mensions and domains for the input arguments passed successfully.
 6924 Either way, output vector **w** is ready to be used in the next method
 6925 of the sequence.

6926 **GrB_PANIC** Unknown internal error.

6927 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 6928 GraphBLAS objects (input or output) is in an invalid state caused
 6929 by a previous execution error. Call `GrB_error()` to access any error
 6930 messages generated by the implementation.

6931 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

6932 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 6933 a call to `new` (or `dup` for vector parameters).

6934 **GrB_DIMENSION_MISMATCH** **mask**, **w** and/or **u** dimensions are incompatible.

6935 **GrB_DOMAIN_MISMATCH** Either the domains of the various vectors and matrices are incom-
 6936 patible with the corresponding domains of the accumulation oper-
 6937 ator or reduce function, or the domains of the GraphBLAS binary

operator `op` are not all the same, or the mask's domain is not compatible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6941 Description

6942 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows
 6943 of an input matrix: $w(i) = \bigoplus A(i, :) \forall i$; or, if an optional binary accumulation operator is provided,
 6944 $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$, where $\bigoplus = \odot(F_b)$ and $\odot = \odot(\text{accum})$.

6945 Logically, this operation occurs in three steps:

6946 **Setup** The internal vector, matrix and mask used in the computation are formed and their
 6947 domains and dimensions are tested for compatibility.

6948 **Compute** The indicated computations are carried out.

6949 **Output** The result is written into the output vector, possibly under control of a mask.

6950 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6951 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6952 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 6953 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6954 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested
 6955 for domain compatibility as follows:

- 6956 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 6957 must be from one of the pre-defined types of Table 3.2.
- 6958 2. $\mathbf{D}(w)$ must be compatible with the domain of the reduction binary operator, $\mathbf{D}(F_b)$.
- 6959 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6960 of the accumulation operator and $\mathbf{D}(F_b)$, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 6961 mulation operator.
- 6962 4. $\mathbf{D}(A)$ must be compatible with the domain of the binary reduction operator, $\mathbf{D}(F_b)$.

6963 Two domains are compatible with each other if values from one domain can be cast to values in
 6964 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6965 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6966 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 6967 error listed above is returned.

6968 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 6969 denotes copy):

- 6970 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6971 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 6972 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 6973 (b) If `mask \neq GrB_NULL`,
- 6974 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 6975 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 6976 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6977 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

6978 The internal vectors and masks are checked for dimension compatibility. The following conditions
6979 must hold:

- 6980 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6981 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6982 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-
6983 match error listed above is returned.

6984 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6985 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6986 We carry out the reduce and any additional associated operations. We describe this in terms of
6987 two intermediate vectors:

- 6988 • $\tilde{\mathbf{t}}$: The vector holding the result from reducing along the rows of input matrix $\tilde{\mathbf{A}}$.
- 6989 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6990 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6991 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6992 The value of each of its elements is computed by

$$6993 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6994 where $\bigoplus = \odot(F_b)$.

6995 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6996 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

6997 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6998 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6999 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
7000 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 7001 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 7002 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 7003 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 7004 \quad & \\ 7005 \end{aligned}$$

7006 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7007 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
7008 using what is called a *standard vector mask and replace*. This is carried out under control of the
7009 mask which acts as a “write mask”.

7010 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
7011 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$7012 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7013 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
7014 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
7015 mask are unchanged:

$$7016 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7017 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
7018 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
7019 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
7020 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7021 sequence.

7022 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

7023 Reduce all stored values into a single scalar.

7024 C Syntax

```
7025 // scalar value + monoid (only)
7026 GrB_Info GrB_reduce(<type>          *val,
7027                      const GrB_BinaryOp accum,
7028                      const GrB_Monoid  op,
7029                      const GrB_Vector  u,
```

```

7030             const GrB_Descriptor desc);
7031
7032 // GraphBLAS Scalar + monoid
7033 GrB_Info GrB_reduce(GrB_Scalar      s,
7034                   const GrB_BinaryOp accum,
7035                   const GrB_Monoid  op,
7036                   const GrB_Vector  u,
7037                   const GrB_Descriptor desc);
7038
7039 // GraphBLAS Scalar + binary operator
7040 GrB_Info GrB_reduce(GrB_Scalar      s,
7041                   const GrB_BinaryOp accum,
7042                   const GrB_BinaryOp op,
7043                   const GrB_Vector  u,
7044                   const GrB_Descriptor desc);

```

7045 Parameters

7046 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
7047 a value that may be accumulated (optionally) with the result of the reduction
7048 operation. On output, this scalar holds the results of the operation.

7049 **accum** (IN) An optional binary operator used for accumulating entries into an exist-
7050 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,
7051 **GrB_NULL** should be specified.

7052 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
7053 the reduction operation. The \oplus operator must be commutative and associative;
7054 otherwise, the outcome of the operation is undefined.

7055 **u** (IN) The GraphBLAS vector on which reduction will be performed.

7056 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
7057 should be specified. Non-default field/value pairs are listed as follows:

7059 Param	Field	Value	Description
------------	-------	-------	-------------

7060 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
7061 tions. There are currently no non-default field/value pairs that can be set for this
7062 operation.

7063 Return Values

7064 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
7065 cessfully, and the output scalar (**s** or **val**) is ready to be used in the
7066 next method of the sequence.

7067 GrB_PANIC Unknown internal error.

7068 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
7069 GraphBLAS objects (input or output) is in an invalid state caused
7070 by a previous execution error. Call GrB_error() to access any error
7071 messages generated by the implementation.

7072 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7073 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
7074 a call to a respective constructor.

7075 GrB_NULL_POINTER val pointer is NULL.

7076 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
7077 the corresponding domains of the accumulation operator, or reduce
7078 operator.

7079 Description

7080 This variant of GrB_reduce computes the result of performing a reduction across all of the stored
7081 elements of an input vector storing the result into either s or val. This corresponds to (shown here
7082 for the scalar value case only):

$$7083 \quad \text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[\bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7084 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7085 Logically, this operation occurs in three steps:

7086 **Setup** The internal vector used in the computation is formed and its domain is tested for
7087 compatibility.

7088 **Compute** The indicated computations are carried out.

7089 **Output** The result is written into the output scalar.

7090 One vector argument is used in this GrB_reduce operation:

- 7091 1. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7092 The output scalar, argument vector, reduction operator and accumulation operator (if provided)
7093 are tested for domain compatibility as follows:

- 7094 1. If accum is GrB_NULL, then $\mathbf{D}(\text{val})$ or $\mathbf{D}(\mathbf{s})$ must be compatible with $\mathbf{D}(\text{op})$ from M (or with
7095 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

- 7096 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
 7097 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 7098 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 7099 3. $\mathbf{D}(\text{u})$ 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).

7100 Two domains are compatible with each other if values from one domain can be cast to values in
 7101 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7102 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7103 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 7104 error listed above is returned.

7105 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one
 7106 of the following occurs depending on the output variant:

$$7107 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7108 or

$$7109 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7110 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7111 For all other cases, the internal vector and scalar used in the computation is formed (\leftarrow denotes
 7112 copy):

- 7113 1. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 7114 2. Scalar $\tilde{s} \leftarrow \text{s}$ (GraphBLAS scalar case).

7115 We are now ready to carry out the reduction and any additional associated operations. An inter-
 7116 mediate scalar result t is computed as follows:

$$7117 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

7118 where $\oplus = \odot(\text{op})$.

7119 The final reduction value is computed as follows:

$$7120 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7121 or

$$7122 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7123 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
7124 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7125 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7126 Reduce all stored values into a single scalar.

7127 C Syntax

```
7128 // scalar value + monoid (only)
7129 GrB_Info GrB_reduce(<type>          *val,
7130                    const GrB_BinaryOp accum,
7131                    const GrB_Monoid  op,
7132                    const GrB_Matrix  A,
7133                    const GrB_Descriptor desc);
7134
7135 // GraphBLAS Scalar + monoid
7136 GrB_Info GrB_reduce(GrB_Scalar      s,
7137                    const GrB_BinaryOp accum,
7138                    const GrB_Monoid  op,
7139                    const GrB_Matrix  A,
7140                    const GrB_Descriptor desc);
7141
7142 // GraphBLAS Scalar + binary operator
7143 GrB_Info GrB_reduce(GrB_Scalar      s,
7144                    const GrB_BinaryOp accum,
7145                    const GrB_BinaryOp op,
7146                    const GrB_Matrix  A,
7147                    const GrB_Descriptor desc);
```

7148 Parameters

7149 val or s (INOUT) Scalar to store final reduced value into. On input, the scalar provides
7150 a value that may be accumulated (optionally) with the result of the reduction
7151 operation. On output, this scalar holds the results of the operation.

7152 accum (IN) An optional binary operator used for accumulating entries into existing (s or
7153 val) value. If assignment rather than accumulation is desired, GrB_NULL should
7154 be specified.

7155 op (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
7156 the reduction operation. The \oplus operator must be commutative and associative;
7157 otherwise, the outcome of the operation is undefined.

7158 A (IN) The GraphBLAS matrix on which the reduction will be performed.

7159 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 7160 should be specified. Non-default field/value pairs are listed as follows:

7161

Param	Field	Value	Description
-------	-------	-------	-------------

7163 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
 7164 tions. There are currently no non-default field/value pairs that can be set for this
 7165 operation.

7166 Return Values

7167 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
 7168 cessfully, and the output scalar (s or val) is ready to be used in the
 7169 next method of the sequence.

7170 GrB_PANIC Unknown internal error.

7171 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 7172 GraphBLAS objects (input or output) is in an invalid state caused
 7173 by a previous execution error. Call GrB_error() to access any error
 7174 messages generated by the implementation.

7175 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7176 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 7177 a call to a respective constructor.

7178 GrB_NULL_POINTER val pointer is NULL.

7179 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
 7180 the corresponding domains of the accumulation operator, or reduce
 7181 operator.

7182 Description

7183 This variant of GrB_reduce computes the result of performing a reduction across all of the stored
 7184 elements of an input matrix storing the result into either s or val. This corresponds to (shown here
 7185 for the scalar value case only):

$$7186 \quad \text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[\bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7187 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7188 Logically, this operation occurs in three steps:

7189 **Setup** The internal matrix used in the computation is formed and its domain is tested for
 7190 compatibility.

7191 **Compute** The indicated computations are carried out.

7192 **Output** The result is written into the output scalar.

7193 One matrix argument is used in this GrB_reduce operation:

7194 1. $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7195 The output scalar, argument matrix, reduction operator and accumulation operator (if provided)
 7196 are tested for domain compatibility as follows:

7197 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
 7198 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

7199 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
 7200 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 7201 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

7202 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).

7203 Two domains are compatible with each other if values from one domain can be cast to values in
 7204 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7205 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7206 any compatibility rule above is violated, execution of GrB_reduce ends and the domain mismatch
 7207 error listed above is returned.

7208 The number of values stored in the input, A , is checked. If there are no stored values in A , then
 7209 one of the following occurs depending on the output variant:

$$7210 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7211 or

$$7212 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7213 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with GrB_SUCCESS.

7214 For all other cases, the internal matrix and scalar used in the computation is formed (\leftarrow denotes
 7215 copy):

7216 1. Matrix $\tilde{A} \leftarrow A$.

7217 2. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

7218 We are now ready to carry out the reduce and any additional associated operations. An intermediate
 7219 scalar result t is computed as follows:

$$7220 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7221 where $\oplus = \odot(\text{op})$.

7222 The final reduction value is computed as follows:

$$7223 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7224 or

$$7225 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7226 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
 7227 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7228 4.3.11 transpose: Transpose rows and columns of a matrix

7229 This version computes a new matrix that is the transpose of the source matrix.

7230 C Syntax

```
7231      GrB_Info GrB_transpose(GrB_Matrix      C,
7232                           const GrB_Matrix Mask,
7233                           const GrB_BinaryOp accum,
7234                           const GrB_Matrix A,
7235                           const GrB_Descriptor desc);
```

7236 Parameters

7237 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 7238 that may be accumulated with the result of the transpose operation. On output,
 7239 the matrix holds the results of the operation.

7240 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 7241 stored into the output matrix C. The mask dimensions must match those of the
 7242 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 7243 of the Mask matrix must be of type bool or any of the predefined “built-in” types
 7244 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
 7245 dimensions of C), GrB_NULL should be specified.

7246 **accum** (IN) An optional binary operator used for accumulating entries into existing C
7247 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
7248 specified.

7249 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7250 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
7251 should be specified. Non-default field/value pairs are listed as follows:

7252

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.

7253

7254 Return Values

7255 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
7256 blocking mode, this indicates that the compatibility tests on di-
7257 mensions and domains for the input arguments passed successfully.
7258 Either way, output matrix C is ready to be used in the next method
7259 of the sequence.

7260 **GrB_PANIC** Unknown internal error.

7261 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
7262 GraphBLAS objects (input or output) is in an invalid state caused
7263 by a previous execution error. Call **GrB_error()** to access any error
7264 messages generated by the implementation.

7265 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

7266 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
7267 a call to **new** (or **Matrix_dup** for matrix parameters).

7268 **GrB_DIMENSION_MISMATCH** mask, C and/or A dimensions are incompatible.

7269 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the cor-
7270 responding domains of the accumulation operator, or the mask's do-
7271 main is not compatible with **bool** (in the case where **desc[GrB_MASK].GrB_STRUCTURE**
7272 is not set).

7273 Description

7274 GrB_transpose computes the result of performing a transpose of the input matrix: $C = A^T$; or, if an
 7275 optional binary accumulation operator (\odot) is provided, $C = C \odot A^T$. We note that the input matrix
 7276 A can itself be optionally transposed before the operation, which would cause either an assignment
 7277 from A to C or an accumulation of A into C.

7278 Logically, this operation occurs in three steps:

7279 **Setup** The internal matrix and mask used in the computation are formed and their domains
 7280 and dimensions are tested for compatibility.

7281 **Compute** The indicated computations are carried out.

7282 **Output** The result is written into the output matrix, possibly under control of a mask.

7283 Up to three matrix arguments are used in this GrB_transpose operation:

- 7284 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7285 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)
- 7286 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7287 The argument matrices and accumulation operator (if provided) are tested for domain compatibility
 7288 as follows:

- 7289 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
 7290 must be from one of the pre-defined types of Table 3.2.
- 7291 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$ of the input matrix.
- 7292 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 7293 of the accumulation operator and $\mathbf{D}(A)$ of the input matrix must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 7294 of the accumulation operator.

7295 Two domains are compatible with each other if values from one domain can be cast to values in
 7296 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7297 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7298 any compatibility rule above is violated, execution of GrB_transpose ends and the domain mismatch
 7299 error listed above is returned.

7300 From the argument matrices, the internal matrices and mask used in the computation are formed
 7301 (\leftarrow denotes copy):

- 7302 1. Matrix $\tilde{C} \leftarrow C$.
- 7303 2. Two-dimensional mask, \tilde{M} , is computed from argument Mask as follows:

- 7304 (a) If $\text{Mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
7305 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 7306 (b) If $\text{Mask} \neq \text{GrB_NULL}$,
- 7307 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
7308 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
- 7309 ii. Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
7310 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
- 7311 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$.
- 7312 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

7313 The internal matrices and masks are checked for dimension compatibility. The following conditions
7314 must hold:

- 7315 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
- 7316 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
- 7317 3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.
- 7318 4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

7319 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension
7320 mismatch error listed above is returned.

7321 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
7322 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7323 We are now ready to carry out the matrix transposition and any additional associated operations.
7324 We describe this in terms of two intermediate matrices:

- 7325 • $\widetilde{\mathbf{T}}$: The matrix holding the transpose of $\widetilde{\mathbf{A}}$.
- 7326 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

7327 The intermediate matrix

$$7328 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7329 is created.

7330 The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7331 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- 7332 • If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$7333 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

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 $\tilde{\mathbf{T}}$.

$$\begin{aligned}
 7336 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 7337 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7338 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7339 \quad & \\
 7340 \quad &
 \end{aligned}$$

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 \mathbf{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 \mathbf{C} on input to this operation are
 7346 deleted and the content of the new output matrix, \mathbf{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, \mathbf{C} , and elements of \mathbf{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 \mathbf{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 \mathbf{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.

7357 4.3.12 kronecker: Kronecker product of two matrices

7358 Computes the Kronecker product of two matrices. The result is a matrix.

7359 C Syntax

```

7360      GrB_Info GrB_kronecker(GrB_Matrix      C,
7361                             const GrB_Matrix Mask,
7362                             const GrB_BinaryOp accum,
7363                             const GrB_Semiring op,
7364                             const GrB_Matrix A,
7365                             const GrB_Matrix B,
7366                             const GrB_Descriptor desc);
7367
```

```

7368     GrB_Info GrB_kronecker(GrB_Matrix      C,
7369                           const GrB_Matrix Mask,
7370                           const GrB_BinaryOp accum,
7371                           const GrB_Monoid  op,
7372                           const GrB_Matrix  A,
7373                           const GrB_Matrix  B,
7374                           const GrB_Descriptor desc);
7375
7376     GrB_Info GrB_kronecker(GrB_Matrix      C,
7377                           const GrB_Matrix Mask,
7378                           const GrB_BinaryOp accum,
7379                           const GrB_BinaryOp op,
7380                           const GrB_Matrix  A,
7381                           const GrB_Matrix  B,
7382                           const GrB_Descriptor desc);

```

7383 Parameters

7384 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
7385 that may be accumulated with the result of the Kronecker product. On output,
7386 the matrix holds the results of the operation.

7387 **Mask** (IN) An optional “write” mask that controls which results from this operation are
7388 stored into the output matrix C. The mask dimensions must match those of the
7389 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
7390 of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types
7391 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
7392 dimensions of C), `GrB_NULL` should be specified.

7393 **accum** (IN) An optional binary operator used for accumulating entries into existing C
7394 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
7395 specified.

7396 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
7397 operation. Depending on which type is passed, the following defines the binary
7398 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

7399 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

7400 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-
7401 nored.

7402 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
7403 is ignored.

7404 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
7405 product.

7406 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
7407 product.

7408 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
7409 should be specified. Non-default field/value pairs are listed as follows:
7410

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.

7412 Return Values

7413 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
7414 blocking mode, this indicates that the compatibility tests on di-
7415 mensions and domains for the input arguments passed successfully.
7416 Either way, output matrix C is ready to be used in the next method
7417 of the sequence.

7418 GrB_PANIC Unknown internal error.

7419 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
7420 GraphBLAS objects (input or output) is in an invalid state caused
7421 by a previous execution error. Call GrB_error() to access any error
7422 messages generated by the implementation.

7423 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7424 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
7425 a call to new (or Matrix_dup for matrix parameters).

7426 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

7427 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
7428 corresponding domains of the binary operator (op) or accumulation
7429 operator, or the mask's domain is not compatible with bool (in the
7430 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

7431 Description

7432 GrB_kronecker computes the Kronecker product $C = A \otimes B$ or, if an optional binary accumulation
7433 operator (\odot) is provided, $C = C \odot (A \otimes B)$ (where matrices A and B can be optionally transposed).

7434 The Kronecker product is defined as follows:

7435

$$7436 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

7437 where $A : \mathbb{S}^{m_A \times n_A}$, $B : \mathbb{S}^{m_B \times n_B}$, and $C : \mathbb{S}^{m_A m_B \times n_A n_B}$. More explicitly, the elements of the
7438 Kronecker product are defined as

$$7439 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7440 where \otimes is the multiplicative operator specified by the `op` parameter.

7441 Logically, this operation occurs in three steps:

7442 **Setup** The internal matrices and mask used in the computation are formed and their domains
7443 and dimensions are tested for compatibility.

7444 **Compute** The indicated computations are carried out.

7445 **Output** The result is written into the output matrix, possibly under control of a mask.

7446 Up to four argument matrices are used in the `GrB_kronecker` operation:

- 7447 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7448 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)
- 7449 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 7450 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

7451 The argument matrices, the "product" operator (`op`), and the accumulation operator (if provided)
7452 are tested for domain compatibility as follows:

- 7453 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
7454 must be from one of the pre-defined types of Table 3.2.
- 7455 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 7456 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 7457 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 7458 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
7459 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
7460 the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of `GrB_kronecker` ends and the domain mismatch error listed above is returned.

From the argument matrices, the internal matrices and mask used in the computation are formed (\leftarrow denotes copy):

1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - (b) If `Mask \neq GrB_NULL`,
 - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

The internal matrices and masks are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) \cdot \mathbf{nrows}(\tilde{\mathbf{B}})$.
4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) \cdot \mathbf{ncols}(\tilde{\mathbf{B}})$.

If any compatibility rule above is violated, execution of `GrB_kronecker` ends and the dimension mismatch error listed above is returned.

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.

We are now ready to carry out the Kronecker product and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\tilde{\mathbf{T}}$: The matrix holding the Kronecker product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

7494 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$
7495 $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$ is created. The value of
7496 each of its elements is computed by

$$7497 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7498 where \otimes is the multiplicative operator specified by the `op` parameter.

7499 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7500 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 7501 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$7502 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7503 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
7504 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$7505 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7506 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7507 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7508 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7511 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
7512 using what is called a *standard matrix mask and replace*. This is carried out under control of the
7513 mask which acts as a “write mask”.

- 7514 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
7515 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7516 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7517 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
7518 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
7519 mask are unchanged:

$$7520 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7521 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
7522 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
7523 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
7524 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7525 sequence. s

Chapter 5

Nonpolymorphic interface[Scott: NEW CONTENT]

Each polymorphic GraphBLAS method (those with multiple parameter signatures under the same name) has a corresponding set of long-name forms that are specific to each parameter signature. 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).[\[Scott: NEW CONTENT\]](#)

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).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>	<code>GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>	<code>GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>	<code>GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>	<code>GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>	<code>GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>	<code>GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>	<code>GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>	<code>GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>	<code>GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>	<code>GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>	<code>GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>	<code>GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)</code>	<code>GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>	<code>GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>	<code>GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>	<code>GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>	<code>GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>	<code>GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>	<code>GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>	<code>GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>	<code>GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>	<code>GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>	<code>GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>	<code>GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>	<code>GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)</code>	<code>GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)</code>

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,...)

Table 5.12: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_get(GrB_Scalar,...,GrB_Scalar)	GrB_Scalar_get_Scalar(GrB_Scalar,...,GrB_Scalar)
GrB_get(GrB_Scalar,...,char*)	GrB_Scalar_get_String(GrB_Scalar,...,char*)
GrB_get(GrB_Scalar,...,int*)	GrB_Scalar_get_ENUM(GrB_Scalar,...,int*)
GrB_get(GrB_Scalar,...,void*)	GrB_Scalar_get_VOID(GrB_Scalar,...,void*)
GrB_get(GrB_Vector,...,GrB_Scalar)	GrB_Vector_get_Scalar(GrB_Vector,...,GrB_Scalar)
GrB_get(GrB_Vector,...,char*)	GrB_Vector_get_String(GrB_Vector,...,char*)
GrB_get(GrB_Vector,...,int*)	GrB_Matrix_get_ENUM(GrB_Vector,...,int*)
GrB_get(GrB_Vector,...,void*)	GrB_Vector_get_VOID(GrB_Vector,...,void*)
GrB_get(GrB_Matrix,...,GrB_Scalar)	GrB_Matrix_get_Scalar(GrB_Matrix,...,GrB_Scalar)
GrB_get(GrB_Matrix,...,char*)	GrB_Matrix_get_String(GrB_Matrix,...,char*)
GrB_get(GrB_Matrix,...,int*)	GrB_Matrix_get_ENUM(GrB_Matrix,...,int*)
GrB_get(GrB_Matrix,...,void*)	GrB_Matrix_get_VOID(GrB_Matrix,...,void*)
GrB_get(GrB_UnaryOp,...,GrB_Scalar)	GrB_UnaryOp_get_Scalar(GrB_UnaryOp,...,GrB_Scalar)
GrB_get(GrB_UnaryOp,...,char*)	GrB_UnaryOp_get_String(GrB_UnaryOp,...,char*)
GrB_get(GrB_UnaryOp,...,int*)	GrB_UnaryOp_get_ENUM(GrB_UnaryOp,...,int*)
GrB_get(GrB_UnaryOp,...,void*)	GrB_UnaryOp_get_VOID(GrB_UnaryOp,...,void*)
GrB_get(GrB_IndexUnaryOp,...,GrB_Scalar)	GrB_IndexUnaryOp_get_Scalar(GrB_IndexUnaryOp,...,GrB_Scalar)
GrB_get(GrB_IndexUnaryOp,...,char*)	GrB_IndexUnaryOp_get_String(GrB_IndexUnaryOp,...,char*)
GrB_get(GrB_IndexUnaryOp,...,int*)	GrB_IndexUnaryOp_get_ENUM(GrB_IndexUnaryOp,...,int*)
GrB_get(GrB_IndexUnaryOp,...,void*)	GrB_IndexUnaryOp_get_VOID(GrB_IndexUnaryOp,...,void*)
GrB_get(GrB_BinaryOp,...,GrB_Scalar)	GrB_BinaryOp_get_Scalar(GrB_BinaryOp,...,GrB_Scalar)
GrB_get(GrB_BinaryOp,...,char*)	GrB_BinaryOp_get_String(GrB_BinaryOp,...,char*)
GrB_get(GrB_BinaryOp,...,int*)	GrB_BinaryOp_get_ENUM(GrB_BinaryOp,...,int*)
GrB_get(GrB_BinaryOp,...,void*)	GrB_BinaryOp_get_VOID(GrB_BinaryOp,...,void*)
GrB_get(GrB_Monoid,...,GrB_Scalar)	GrB_Monoid_get_Scalar(GrB_Monoid,...,GrB_Scalar)
GrB_get(GrB_Monoid,...,char*)	GrB_Monoid_get_String(GrB_Monoid,...,char*)
GrB_get(GrB_Monoid,...,int*)	GrB_Monoid_get_ENUM(GrB_Monoid,...,int*)
GrB_get(GrB_Monoid,...,void*)	GrB_Monoid_get_VOID(GrB_Monoid,...,void*)
GrB_get(GrB_Semiring,...,GrB_Scalar)	GrB_Semiring_get_Scalar(GrB_Semiring,...,GrB_Scalar)
GrB_get(GrB_Semiring,...,char*)	GrB_Semiring_get_String(GrB_Semiring,...,char*)
GrB_get(GrB_Semiring,...,int*)	GrB_Semiring_get_ENUM(GrB_Semiring,...,int*)
GrB_get(GrB_Semiring,...,void*)	GrB_Semiring_get_VOID(GrB_Semiring,...,void*)
GrB_get(GrB_Descriptor,...,GrB_Scalar)	GrB_Descriptor_get_Scalar(GrB_Descriptor,...,GrB_Scalar)
GrB_get(GrB_Descriptor,...,char*)	GrB_Descriptor_get_String(GrB_Descriptor,...,char*)
GrB_get(GrB_Descriptor,...,int*)	GrB_Descriptor_get_ENUM(GrB_Descriptor,...,int*)
GrB_get(GrB_Descriptor,...,void*)	GrB_Descriptor_get_VOID(GrB_Descriptor,...,void*)
GrB_get(GrB_Type,...,GrB_Scalar)	GrB_Type_get_Scalar(GrB_Type,...,GrB_Scalar)
GrB_get(GrB_Type,...,char*)	GrB_Type_get_String(GrB_Type,...,char*)
GrB_get(GrB_Type,...,int*)	GrB_Type_get_ENUM(GrB_Type,...,int*)
GrB_get(GrB_Type,...,void*)	GrB_Type_get_VOID(GrB_Type,...,void*)
GrB_get(...,GrB_Scalar)	GrB_Global_get_Scalar(...,GrB_Scalar)
GrB_get(...,char*)	GrB_Global_get_String(...,char*)
GrB_get(...,int*)	GrB_Global_get_ENUM(...,int*)
GrB_get(...,void*)	GrB_Global_get_VOID(...,void*)
GrB_getPreallocSize(GrB_Scalar,...,int*)	GrB_Scalar_getPreallocSize(GrB_Scalar,...,int*)
GrB_getPreallocSize(GrB_Vector,...,int*)	GrB_Matrix_getPreallocSize(GrB_Vector,...,int*)
GrB_getPreallocSize(GrB_Matrix,...,int*)	GrB_Matrix_getPreallocSize(GrB_Matrix,...,int*)
GrB_getPreallocSize(GrB_UnaryOp,...,int*)	GrB_UnaryOp_getPreallocSize(GrB_UnaryOp,...,int*)
GrB_getPreallocSize(GrB_IndexUnaryOp,...,int*)	GrB_IndexUnaryOp_getPreallocSize(GrB_IndexUnaryOp,...,int*)
GrB_getPreallocSize(GrB_BinaryOp,...,int*)	GrB_BinaryOp_getPreallocSize(GrB_BinaryOp,...,int*)
GrB_getPreallocSize(GrB_Monoid,...,int*)	GrB_Monoid_getPreallocSize(GrB_Monoid,...,int*)
GrB_getPreallocSize(GrB_Semiring,...,int*)	GrB_Semiring_getPreallocSize(GrB_Semiring,...,int*)
GrB_getPreallocSize(GrB_Descriptor,...,int*)	GrB_Descriptor_getPreallocSize(GrB_Descriptor,...,int*)
GrB_getPreallocSize(GrB_Type,...,int*)	GrB_Type_getPreallocSize(GrB_Type,...,int*)
GrB_getPreallocSize(...,int*)	GrB_Global_getPreallocSize(...,int*)

Table 5.13: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_set(GrB_Scalar,...,GrB_Scalar)	GrB_Scalar_set_Scalar(GrB_Scalar,...,GrB_Scalar)
GrB_set(GrB_Scalar,...,char*)	GrB_Scalar_set_String(GrB_Scalar,...,char*)
GrB_set(GrB_Scalar,...,int)	GrB_Scalar_set_ENUM(GrB_Scalar,...,int)
GrB_set(GrB_Scalar,...,void*,int)	GrB_Scalar_set_VOID(GrB_Scalar,...,void*,int)
GrB_set(GrB_Vector,...,GrB_Scalar)	GrB_Vector_set_Scalar(GrB_Vector,...,GrB_Scalar)
GrB_set(GrB_Vector,...,char*)	GrB_Vector_set_String(GrB_Vector,...,char*)
GrB_set(GrB_Vector,...,int)	GrB_Vector_set_ENUM(GrB_Vector,...,int)
GrB_set(GrB_Vector,...,void*,int)	GrB_Vector_set_VOID(GrB_Vector,...,void*,int)
GrB_set(GrB_Matrix,...,GrB_Scalar)	GrB_Matrix_set_Scalar(GrB_Matrix,...,GrB_Scalar)
GrB_set(GrB_Matrix,...,char*)	GrB_Matrix_set_String(GrB_Matrix,...,char*)
GrB_set(GrB_Matrix,...,int)	GrB_Matrix_set_ENUM(GrB_Matrix,...,int)
GrB_set(GrB_Matrix,...,void*,int)	GrB_Matrix_set_VOID(GrB_Matrix,...,void*,int)
GrB_set(GrB_UnaryOp,...,GrB_Scalar)	GrB_UnaryOp_set_Scalar(GrB_UnaryOp,...,GrB_Scalar)
GrB_set(GrB_UnaryOp,...,char*)	GrB_UnaryOp_set_String(GrB_UnaryOp,...,char*)
GrB_set(GrB_UnaryOp,...,int)	GrB_UnaryOp_set_ENUM(GrB_UnaryOp,...,int)
GrB_set(GrB_UnaryOp,...,void*,int)	GrB_UnaryOp_set_VOID(GrB_UnaryOp,...,void*,int)
GrB_set(GrB_IndexUnaryOp,...,GrB_Scalar)	GrB_IndexUnaryOp_set_Scalar(GrB_IndexUnaryOp,...,GrB_Scalar)
GrB_set(GrB_IndexUnaryOp,...,char*)	GrB_IndexUnaryOp_set_String(GrB_IndexUnaryOp,...,char*)
GrB_set(GrB_IndexUnaryOp,...,int)	GrB_IndexUnaryOp_set_ENUM(GrB_IndexUnaryOp,...,int)
GrB_set(GrB_IndexUnaryOp,...,void*,int)	GrB_IndexUnaryOp_set_VOID(GrB_IndexUnaryOp,...,void*,int)
GrB_set(GrB_BinaryOp,...,GrB_Scalar)	GrB_BinaryOp_set_Scalar(GrB_BinaryOp,...,GrB_Scalar)
GrB_set(GrB_BinaryOp,...,char*)	GrB_BinaryOp_set_String(GrB_BinaryOp,...,char*)
GrB_set(GrB_BinaryOp,...,int)	GrB_BinaryOp_set_ENUM(GrB_BinaryOp,...,int)
GrB_set(GrB_BinaryOp,...,void*,int)	GrB_BinaryOp_set_VOID(GrB_BinaryOp,...,void*,int)
GrB_set(GrB_Monoid,...,GrB_Scalar)	GrB_Monoid_set_Scalar(GrB_Monoid,...,GrB_Scalar)
GrB_set(GrB_Monoid,...,char*)	GrB_Monoid_set_String(GrB_Monoid,...,char*)
GrB_set(GrB_Monoid,...,int)	GrB_Monoid_set_ENUM(GrB_Monoid,...,int)
GrB_set(GrB_Monoid,...,void*,int)	GrB_Monoid_set_VOID(GrB_Monoid,...,void*,int)
GrB_set(GrB_Semiring,...,GrB_Scalar)	GrB_Semiring_set_Scalar(GrB_Semiring,...,GrB_Scalar)
GrB_set(GrB_Semiring,...,char*)	GrB_Semiring_set_String(GrB_Semiring,...,char*)
GrB_set(GrB_Semiring,...,int)	GrB_Semiring_set_ENUM(GrB_Semiring,...,int)
GrB_set(GrB_Semiring,...,void*,int)	GrB_Semiring_set_VOID(GrB_Semiring,...,void*,int)
GrB_set(GrB_Descriptor,...,GrB_Scalar)	GrB_Descriptor_set_Scalar(GrB_Descriptor,...,GrB_Scalar)
GrB_set(GrB_Descriptor,...,char*)	GrB_Descriptor_set_String(GrB_Descriptor,...,char*)
GrB_set(GrB_Descriptor,...,int)	GrB_Descriptor_set_ENUM(GrB_Descriptor,...,int)
GrB_set(GrB_Descriptor,...,void*,int)	GrB_Descriptor_set_VOID(GrB_Descriptor,...,void*,int)
GrB_set(GrB_Type,...,GrB_Scalar)	GrB_Type_set_Scalar(GrB_Type,...,GrB_Scalar)
GrB_set(GrB_Type,...,char*)	GrB_Type_set_String(GrB_Type,...,char*)
GrB_set(GrB_Type,...,int)	GrB_Type_set_ENUM(GrB_Type,...,int)
GrB_set(GrB_Type,...,void*,int)	GrB_Type_set_VOID(GrB_Type,...,void*,int)
GrB_set(...,GrB_Scalar)	GrB_Global_set_Scalar(...,GrB_Scalar)
GrB_set(...,char*)	GrB_Global_set_String(...,char*)
GrB_set(...,int)	GrB_Global_set_ENUM(...,int)
GrB_set(...,void*,int)	GrB_Global_set_VOID(...,void*,int)

Appendix A

Revision history

Changes in 2.0.1 (Released: ## Xxxxx 2022:

- (Issue GH-69) Fix error in description of contents of matrix constructed from `GrB_Matrix_diag`.

Changes in 2.0.0 (Released: 15 November 2021:

- Reorganized Chapters 2 and 3: Chapter 2 contains prose regarding the basic concepts captured in the API; Chapter 3 presents all of the enumerations, literals, data types, and predefined objects required by the API. Made short captions for the List of Tables.
- (Issue BB-49, BB-50) Updated and corrected language regarding multithreading and completion, and requirements regarding acquire-release memory orders. Methods that used to force complete no longer do.
- (Issue BB-74, BB-9) Assigned integer values to all return codes as well as all enumerations in the API to ensure run-time compatibility between libraries.
- (Issues BB-70, BB-67) Changed semantics and signature of `GrB_wait(obj, mode)`. Added wait modes for 'complete' or 'materialize' and removed `GrB_wait(void)`. **This breaks backward compatibility.**
- (Issue GH-51) Removed deprecated `GrB_SCMP` literal from descriptor values. **This breaks backward compatibility.**
- (Issues BB-8, BB-36) Added sparse `GrB_Scalar` object and its use in additional variants of `extract/setElement` methods, and `reduce`, `apply`, `assign` and `select` operations.
- (Issues BB-34, GH-33, GH-45) Added new `select` operation that uses an index unary operator. Added new variants of `apply` that take an index unary operator (matrix and vector variants).
- (Issues BB-68, BB-51) Added `serialize` and `deserialize` methods for matrices to/from implementation defined formats.

- 7556 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified
7557 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats
7558 have been deferred.
- 7559 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7560 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return
7561 error if duplicate locations encountered.
- 7562 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7563 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7564 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type `T` (not to
7565 be confused with the proposed unary operator).
- 7566 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added
7567 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to
7568 a method that is not supported by the implementation.
- 7569 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque
7570 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7571 • (Issue BB-45) Removed language about annihilators.
- 7572 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7573 • Updated a number algorithms in the appendix to use new operations and methods.
- 7574 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the
7575 specification.
- 7576 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and
7577 `GRB_SUBVERSION`.
- 7578 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7579 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection
7580 fixed to read `eWiseAdd`/set union.
- 7581 • Typographical error (`NEQ` \rightarrow `NE`) in Description of Table 3.8.

7582 Changes in 1.3.0 (Released: 25 September 2019):

- 7583 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque
7584 GraphBLAS object as an argument.
- 7585 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7586 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a
7587 scalar.

7588
7589

7590

7591
7592

7593
7594

7595

7596
7597

7598
7599

7600
7601

7602

7603
7604

7605

7606
7607

7608
7609

7610

7611
7612

7613

7614
7615

7616
7617

7618

7619

7620

- (Issue BB-59) Changed specification about how reductions to scalar (`GrB_reduce`) are to be performed (to minimize dependence on monoid identity).
- (Issue BB-24) Added methods to resize matrices and vectors (`GrB_Matrix_resize` and `GrB_Vector_resize`).
- (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.

- 7621 • Fixed miscellaneous typographical errors (such as \otimes , \oplus).
- 7622 Changes in 1.2.0:
- 7623 • Removed "provisional" clause.
- 7624 Changes in 1.1.0:
- 7625 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
 - 7626 `assign` operations.
 - 7627 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
 - 7628 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
 - 7629 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
 - 7630 from the domains of the binary operator provided.
 - 7631 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
 - 7632 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
 - 7633 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
 - 7634 which is returned when the capacities of the output arrays are insufficient to hold all of the
 - 7635 tuples.
 - 7636 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
 - 7637 face.
 - 7638 • Added replace flag (`z`) notation to Table 4.1.
 - 7639 • Updated the “Mathematical Description” of the `assign` operation in Table 4.1.
 - 7640 • Added triangle counting example.
 - 7641 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
 - 7642 sections of GraphBLAS operations when the respective text was the “standard” text (i.e.,
 - 7643 identical in a majority of the operations).
 - 7644 • Fixed typographical errors.
- 7645 Changes in 1.0.2:
- 7646 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
 - 7647 matrices and avoid casting issues in certain implementations.
 - 7648 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
 - 7649 erased outside the assigned area.
 - 7650 • Changes non-polymorphic interface:
 - 7651 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7652 – Renamed GrB_Vector_reduce_BinaryOp to GrB_Matrix_reduce_BinaryOp.
- 7653 – Renamed GrB_Vector_reduce_Monoid to GrB_Matrix_reduce_Monoid.
- 7654 • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7655 • Fixed numerous typographical errors.

Appendix B

Non-opaque data format definitions

B.1 GrB_Format: Specify the format for input/output of a GraphBLAS matrix.

In this section, the non-opaque matrix formats specified by GrB_Format and used in matrix import and export methods are defined.

B.1.1 GrB_CSR_FORMAT

The GrB_CSR_FORMAT format indicates that a matrix will be imported or exported using the compressed sparse row (CSR) format. `indptr` is a pointer to an array of GrB_Index of size `nrows+1` elements, where the `i`'th index will contain the starting index in the `values` and `indices` arrays corresponding to the `i`'th row of the matrix. `indices` is a pointer to an array of number of stored elements (each a GrB_Index), where each element contains the corresponding element's column index within a row of the matrix. `values` is a pointer to an array of number of stored elements (each the size of the scalar stored in the matrix) containing the corresponding value. The elements of each row are not required to be sorted by column index.

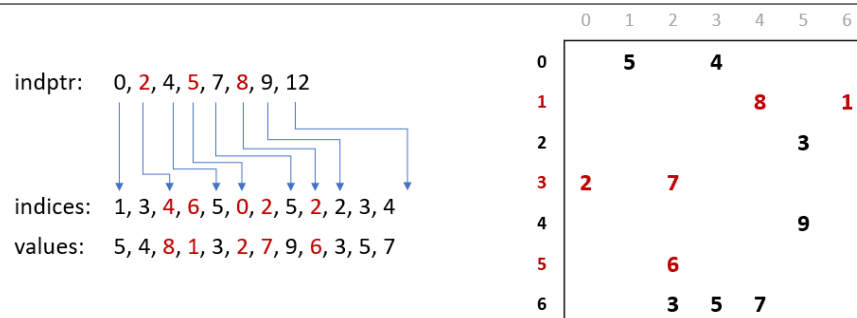


Figure B.1: Data layout for CSR format.

B.1.2 GrB_CSC_FORMAT

The GrB_CSC_FORMAT format indicates that a matrix will be imported or exported using the compressed sparse column (CSC) format. `indptr` is a pointer to an array of `GrB_Index` of size `ncols+1` elements, where the *i*'th index will contain the starting index in the `values` and `indices` arrays corresponding to the *i*'th column of the matrix. `indices` is a pointer to an array of number of stored elements (each a `GrB_Index`), where each element contains the corresponding element's row index within a column of the matrix. `values` is a pointer to an array of number of stored elements (each the size of the scalar stored in the matrix) containing the corresponding value. The elements of each column are not required to be sorted by row index.

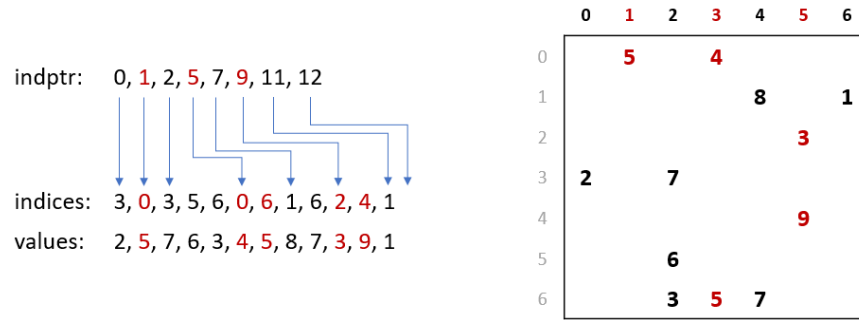


Figure B.2: Data layout for CSC format.

B.1.3 GrB_COO_FORMAT

The GrB_COO_FORMAT format indicates that a matrix will be imported or exported using the coordinate list (COO) format. `indptr` is a pointer to an array of `GrB_Index` of size number of stored elements, where each element contains the corresponding element's column index. `indices` will be a pointer to an array of `GrB_Index` of size number of stored elements, where each element contains the corresponding element's row index. `values` will be a pointer to an array of size number of stored elements (each the size of the scalar stored in the matrix) containing the corresponding value. Elements are not required to be sorted in any order.

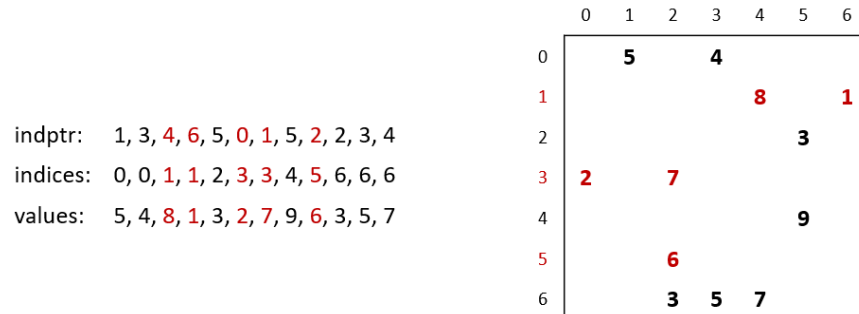


Figure B.3: Data layout for COO format.

7688 **Appendix C**

7689 **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;                          //  $d = \text{level in BFS traversal}$ 
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>  $q(n) = \text{false}$ 
22      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , 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); //  $v[q] = \text{level}$ 
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(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 vxm. 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;
19     GrB_Matrix_new(&sigma, GrB_INT32, n, n);      // Matrix<int32_t>  $\sigma(n,n)$ 
20                                                    //  $\sigma[d,k] = \#$  shortest paths to node  $k$  at level  $d$ 
21
22     GrB_Vector q;
23     GrB_Vector_new(&q, GrB_INT32, n);             // Vector<int32_t>  $q(n)$  of path counts
24     GrB_Vector_setElement(q, 1, s);               //  $q[s] = 1$ 
25
26     GrB_Vector p;
27     GrB_Vector_dup(&p, q);                        // Vector<int32_t>  $p(n)$  shortest path counts so far
28                                                    //  $p = q$ 
29
30     GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
31             q, A, GrB_DESC_RC);                  // get the first set of out neighbors
32
33     /*
34     * BFS phase
35     */
36     GrB_Index d = 0;                             // BFS level number
37     int32_t sum = 0;                             // sum == 0 when BFS phase is complete
38
39     do {
40         GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\sigma[d,:] = q$ 
41         GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
42         GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
43                 q, A, GrB_DESC_RC);              //  $q = \#$  paths to nodes reachable
44                                                    // from current level
45         GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
46         ++d;
47     } while (sum);
48
49     /*
50     * BC computation phase
51     * ( $t1, t2, t3, t4$ ) are temporary vectors
52     */
53     GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
54     GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
55     GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
56     GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
57
58     for(int i=d-1; i>0; i--)
59     {
60         GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t1 = 1 + \delta$ 
61         GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, t1, *delta, GrB_NULL);
62         GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t2 = \sigma[i,:]$ 
63         GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t2 = (1 + \delta) / \sigma[i,:]$ 
64         GrB_mvx(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
65                 // add contributions made by

```

```

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_mxm(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_mnv(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  $c = c \ominus !new$ 
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_mnv(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); //  $C \langle L \rangle = L +.* L$ 
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 }
```