

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 2022/12/12 at 11:03:49 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	String Handling	49
61	3.7.2	Hints	49
62	3.7.3	Input Types	49
63	3.8	GrB_Info return values	51
64	4	Methods	53
65	4.1	Context methods	53
66	4.1.1	init: Initialize a GraphBLAS context	53
67	4.1.2	finalize: Finalize a GraphBLAS context	54
68	4.1.3	getVersion: Get the version number of the standard.	55
69	4.2	Object methods	55
70	4.2.1	Get and Set methods	56
71	4.2.1.1	get: Query the value of an object	56
72	4.2.1.2	set: Set field of an object	57
73	4.2.2	Algebra methods	58
74	4.2.2.1	Type_new: Construct a new GraphBLAS (user-defined) type	58
75	4.2.2.2	UnaryOp_new: Construct a new GraphBLAS unary operator	59
76	4.2.2.3	BinaryOp_new: Construct a new GraphBLAS binary operator . . .	60
77	4.2.2.4	Monoid_new: Construct a new GraphBLAS monoid	62

78	4.2.2.5	Semiring_new: Construct a new GraphBLAS semiring	63
79	4.2.2.6	IndexUnaryOp_new: Construct a new GraphBLAS index unary op-	
80		erator [Scott: NEW CONTENT]	64
81	4.2.3	Scalar methods	66
82	4.2.3.1	Scalar_new: Construct a new scalar	66
83	4.2.3.2	Scalar_dup: Construct a copy of a GraphBLAS scalar	67
84	4.2.3.3	Scalar_clear: Clear/remove a stored value from a scalar	68
85	4.2.3.4	Scalar_nvals: Number of stored elements in a scalar	69
86	4.2.3.5	Scalar_setElement: Set the single element in a scalar	70
87	4.2.3.6	Scalar_extractElement: Extract a single element from a scalar. . . .	71
88	4.2.4	Vector methods	73
89	4.2.4.1	Vector_new: Construct new vector	73
90	4.2.4.2	Vector_dup: Construct a copy of a GraphBLAS vector	74
91	4.2.4.3	Vector_resize: Resize a vector	75
92	4.2.4.4	Vector_clear: Clear a vector	76
93	4.2.4.5	Vector_size: Size of a vector	77
94	4.2.4.6	Vector_nvals: Number of stored elements in a vector	77
95	4.2.4.7	Vector_build: Store elements from tuples into a vector	78
96	4.2.4.8	Vector_setElement: Set a single element in a vector	80
97	4.2.4.9	Vector_removeElement: Remove an element from a vector	82
98	4.2.4.10	Vector_extractElement: Extract a single element from a vector. . . .	83
99	4.2.4.11	Vector_extractTuples: Extract tuples from a vector	85
100	4.2.5	Matrix methods	86
101	4.2.5.1	Matrix_new: Construct new matrix	86
102	4.2.5.2	Matrix_dup: Construct a copy of a GraphBLAS matrix	88
103	4.2.5.3	Matrix_diag: Construct a diagonal GraphBLAS matrix	89
104	4.2.5.4	Matrix_resize: Resize a matrix	90
105	4.2.5.5	Matrix_clear: Clear a matrix	91
106	4.2.5.6	Matrix_nrows: Number of rows in a matrix	92
107	4.2.5.7	Matrix_ncols: Number of columns in a matrix	92
108	4.2.5.8	Matrix_nvals: Number of stored elements in a matrix	93

109	4.2.5.9	Matrix_build: Store elements from tuples into a matrix	94
110	4.2.5.10	Matrix_setElement: Set a single element in matrix	96
111	4.2.5.11	Matrix_removeElement: Remove an element from a matrix	98
112	4.2.5.12	Matrix_extractElement: Extract a single element from a matrix . . .	99
113	4.2.5.13	Matrix_extractTuples: Extract tuples from a matrix	101
114	4.2.5.14	Matrix_exportHint: Provide a hint as to which storage format might	
115		be most efficient for exporting a matrix	103
116	4.2.5.15	Matrix_exportSize: Return the array sizes necessary to export a	
117		GraphBLAS matrix object	104
118	4.2.5.16	Matrix_export: Export a GraphBLAS matrix to a pre-defined format	105
119	4.2.5.17	Matrix_import: Import a matrix into a GraphBLAS object	107
120	4.2.5.18	Matrix_serializeSize: Compute the serialize buffer size	109
121	4.2.5.19	Matrix_serialize: Serialize a GraphBLAS matrix.	110
122	4.2.5.20	Matrix_deserialize: Deserialize a GraphBLAS matrix.	111
123	4.2.6	Descriptor methods	112
124	4.2.6.1	Descriptor_new: Create new descriptor	112
125	4.2.6.2	Descriptor_set: Set content of descriptor	113
126	4.2.7	free: Destroy an object and release its resources	114
127	4.2.8	wait: Return once an object is either <i>complete</i> or <i>materialized</i>	116
128	4.2.9	error: Retrieve an error string	117
129	4.3	GraphBLAS operations	118
130	4.3.1	mxm: Matrix-matrix multiply	122
131	4.3.2	vxm: Vector-matrix multiply	127
132	4.3.3	mxv: Matrix-vector multiply	131
133	4.3.4	eWiseMult: Element-wise multiplication	135
134	4.3.4.1	eWiseMult: Vector variant	136
135	4.3.4.2	eWiseMult: Matrix variant	140
136	4.3.5	eWiseAdd: Element-wise addition	145
137	4.3.5.1	eWiseAdd: Vector variant	146
138	4.3.5.2	eWiseAdd: Matrix variant	150
139	4.3.6	extract: Selecting sub-graphs	156

140	4.3.6.1	extract: Standard vector variant	156
141	4.3.6.2	extract: Standard matrix variant	160
142	4.3.6.3	extract: Column (and row) variant	165
143	4.3.7	assign: Modifying sub-graphs	170
144	4.3.7.1	assign: Standard vector variant	170
145	4.3.7.2	assign: Standard matrix variant	175
146	4.3.7.3	assign: Column variant	181
147	4.3.7.4	assign: Row variant	186
148	4.3.7.5	assign: Constant vector variant[Scott: NEW CONTENT]	192
149	4.3.7.6	assign: Constant matrix variant[Scott: NEW CONTENT]	197
150	4.3.8	apply: Apply a function to the elements of an object	203
151	4.3.8.1	apply: Vector variant	203
152	4.3.8.2	apply: Matrix variant	208
153	4.3.8.3	apply: Vector-BinaryOp variants[Scott: NEW CONTENT]	212
154	4.3.8.4	apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]	218
155	4.3.8.5	apply: Vector index unary operator variant[Scott: NEW CONTENT]	224
156	4.3.8.6	apply: Matrix index unary operator variant[Scott: NEW CONTENT]	229
157	4.3.9	select:	234
158	4.3.9.1	select: Vector variant[Scott: NEW CONTENT]	234
159	4.3.9.2	select: Matrix variant[Scott: NEW CONTENT]	239
160	4.3.10	reduce: Perform a reduction across the elements of an object	245
161	4.3.10.1	reduce: Standard matrix to vector variant	245
162	4.3.10.2	reduce: Vector-scalar variant[Scott: NEW CONTENT]	249
163	4.3.10.3	reduce: Matrix-scalar variant[Scott: NEW CONTENT]	253
164	4.3.11	transpose: Transpose rows and columns of a matrix	256
165	4.3.12	kroncker: Kronecker product of two matrices	260
166	5	Nonpolymorphic interface[Scott: NEW CONTENT]	267
167	A	Revision history	281
168	B	Non-opaque data format definitions	287

169	B.1	GrB_Format: Specify the format for input/output of a GraphBLAS matrix.	287
170	B.1.1	GrB_CSR_FORMAT	287
171	B.1.2	GrB_CSC_FORMAT	288
172	B.1.3	GrB_COO_FORMAT	288
173	C	Examples	289
174	C.1	Example: Level breadth-first search (BFS) in GraphBLAS	290
175	C.2	Example: Level BFS in GraphBLAS using apply	291
176	C.3	Example: Parent BFS in GraphBLAS	292
177	C.4	Example: Betweenness centrality (BC) in GraphBLAS	293
178	C.5	Example: Batched BC in GraphBLAS	295
179	C.6	Example: Maximal independent set (MIS) in GraphBLAS	297
180	C.7	Example: Counting triangles in GraphBLAS	299

List of Tables

2.1	Types of GraphBLAS opaque objects.	23
2.2	Methods that forced completion prior to GraphBLAS v2.0.	28
3.1	Enumeration literals and corresponding values input to various GraphBLAS methods.	32
3.2	Predefined GrB_Type values.	33
3.3	Operator input for relevant GraphBLAS operations.	34
3.4	Properties and recipes for building GraphBLAS algebraic objects.	35
3.5	Predefined unary and binary operators for GraphBLAS in C.	37
3.6	Predefined index unary operators for GraphBLAS in C.	38
3.7	Predefined monoids for GraphBLAS in C.	40
3.8	Predefined “true” semirings for GraphBLAS in C.	41
3.9	Other useful predefined semirings for GraphBLAS in C.	42
3.10	GrB_Format enumeration literals and corresponding values for matrix import and export methods.	44
3.11	Descriptor types and literals for fields and values.	47
3.12	Predefined GraphBLAS descriptors.	48
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	50
3.14	Descriptions of select <i>field</i> , <i>value</i> pairs listed in 3.13	51
3.15	Enumeration literals and corresponding values returned by GraphBLAS methods and operations.	52

206	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in	
207		this specification.	119
208	5.1	Long-name, nonpolymorphic form of GraphBLAS methods.	267
209	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	268
210	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	269
211	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	270
212	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	271
213	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	272
214	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	273
215	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	274
216	5.9	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
217		CONTENT]	275
218	5.10	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
219		CONTENT]	276
220	5.11	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	277
221	5.12	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	278
222	5.13	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	279

223 List of Figures

224	3.1 Hierarchy of algebraic object classes in GraphBLAS.	43
225	4.1 Flowchart for the GraphBLAS operations.	120
226	B.1 Data layout for CSR format.	287
227	B.2 Data layout for CSC format.	288
228	B.3 Data layout for COO format.	288

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)
- Aydın 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):

- Aydın 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*

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

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

Notation	Description
$D_{out}, D_{in}, D_{in_1}, D_{in_2}$	Refers to output and input domains of various GraphBLAS operators.
$\mathbf{D}_{out}(*), \mathbf{D}_{in}(*), \mathbf{D}_{in_1}(*), \mathbf{D}_{in_2}(*)$	Evaluates to output and input domains of GraphBLAS operators (usually a unary or binary operator, or semiring).
$\mathbf{D}(*)$	Evaluates to the (only) domain of a GraphBLAS object (usually a monoid, vector, or matrix).
f	An arbitrary unary function, usually a component of a unary operator.
$\mathbf{f}(F_u)$	Evaluates to the unary function contained in the unary operator given as the argument.
\odot	An arbitrary binary function, usually a component of a binary operator.
$\odot(*)$	Evaluates to the binary function contained in the binary operator or monoid given as the argument.
\otimes	Multiplicative binary operator of a semiring.
\oplus	Additive binary operator of a semiring.
$\otimes(S)$	Evaluates to the multiplicative binary operator of the semiring given as the argument.
$\oplus(S)$	Evaluates to the additive binary operator of the semiring given as the argument.
$\mathbf{0}(*)$	The identity of a monoid, or the additive identity of a GraphBLAS semiring.
$\mathbf{L}(*)$	The contents (all stored values) of the vector or matrix GraphBLAS objects. For a vector, it is the set of (index, value) pairs, and for a matrix it is the set of (row, col, value) triples.
$\mathbf{v}(i)$ 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 3.15. 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 3.15(a) are non-negative and include `GrB_SUCCESS` (a value of 0) and `GrB_NO_VALUE`.

An API error (listed in Table 3.15(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 3.15(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

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

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

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

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

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

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

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

794 3.3 Types (domains)

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

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

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

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

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

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

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

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

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.

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 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

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

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

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

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

(a) Properties of algebraic objects.

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

(b) Recipes for algebraic objects.

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

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

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

840 D_{in_2} , and an operation $\odot : D_{in_1} \times D_{in_2} \rightarrow D_{out}$. For a given GraphBLAS binary operator $F_b =$
841 $\langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$, 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 $\odot(F_b) =$
842 \odot . Note that \odot could be used in place of either \oplus or \otimes in other methods and operations.

843 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
844 domains, D_{out} , D_{in_1} , D_{in_2} , the domain of GraphBLAS indices, and an operation $f_i : D_{in_1} \times I_{U64}^2 \times$
845 $D_{in_2} \rightarrow D_{out}$ (where I_{U64} corresponds to the domain of a `GrB_Index`). For a given GraphBLAS
846 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 $\mathbf{f}(F_i) = f_i$.

847 User-defined operators can be created with calls to `GrB_UnaryOp_new`, `GrB_BinaryOp_new`, and
848 `GrB_IndexUnaryOp_new`, respectively. See Section 4.2.2 for information on these methods. The
849 GraphBLAS C API predefines a number of these operators. These are listed in Tables 3.5 and 3.6.
850 Note that most entries in these tables represent a “family” of predefined operators for a set of
851 different types represented by the T , I , or F in their names. For example, the multiplicative
852 inverse (`GrB_MINV_F`) function is only defined for floating-point types ($F = \text{FP32}$ or FP64). The
853 division (`GrB_DIV_T`) function is defined for all types, but only if $y \neq 0$ for integral and floating
854 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 Name	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
	$\text{INT}x$	$\text{INT}x_MIN$	
	$\text{FP}x$	$-INFINITY$	

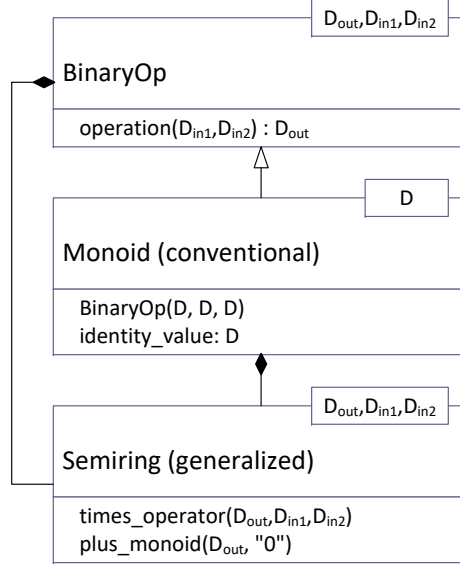


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

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 **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 **size**(\mathbf{m}) = N . The set **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 **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 **ncols**(\mathbf{M}) = N , and **nrows**(\mathbf{M}) = M . We also define the sets **indrow**(\mathbf{M}) = $\{i : \exists (i, j) \in \mathbf{ind}(\mathbf{M})\}$ and **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 **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 **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 **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 `get` and `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_NAME` field is a special case regarding writability. All objects which have a `GrB_NAME` field default to an empty string, `GrB_NAMESIZE` will be 0. Collections and `GrB_Descriptors` may have their `GrB_NAME` set at any time. Algebraic objects and `GrB_Types` may only have their

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

1005 GrB_NAME set once to a globally unique value. Attempting to set this field after it has already
1006 been set will return a GrB_ALREADY_SET error code.

1007 The GrB_Field enumeration is defined by the values in Table 3.13, and selected values are described
1008 in Table 3.14.

1009 3.7.1 String Handling

1010 When the input to GrB_<OBJ>_set is a char* the input array is null terminated. The GraphBLAS
1011 implementation must copy this array into internal data structures.

1012 When a char* is the output argument of GrB_<OBJ>_get the user must preallocate a properly
1013 sized buffer. The returned string is null terminated, so the buffer must be at least 1 larger than the
1014 size of the string. For instance the buffer for GrB_NAME must be of length GrB_NAMESIZE + 1.

1015 3.7.2 Hints

1016 Several fields are *hints* (marked H in Table 3.13). A GraphBLAS implementation is free to ignore
1017 a hint and return GrB_SUCCESS. For instance GrB_NTHREADS might be ignored by a sequential
1018 GraphBLAS implementation.

1019 3.7.3 Input Types

1020 By default arguments are passed in a GrB_Scalar, only char* and GraphBLAS objects like GrB_Type
1021 are passed in directly. Enums are passed as GrB_INT32 GrBScalars.

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	GrB_Desc_Value
GrB_MASK	W —	1	GrB_Descriptor	GrB_Desc_Value
GrB_INP0	W —	2	GrB_Descriptor	GrB_Desc_Value
GrB_INP1	W —	3	GrB_Descriptor	GrB_Desc_Value
GrB_NAMESIZE	— —	10	All, Global	GrB_INT32
GrB_NAME	*	11	All	Null terminated char* of size GrB_NAMESIZE
GrB_LIBRARY_NAME	— —	100	Global	Null terminated char* of size GrB_NAMESIZE
GrB_LIBRARY_VER_MAJOR	— —	101	Global	GrB_INT32
GrB_LIBRARY_VER_MINOR	— —	101	Global	GrB_INT32
GrB_LIBRARY_VER_PATCH	— —	101	Global	GrB_INT32
GrB_API_VER_MAJOR	— —	102	Global	GrB_INT32
GrB_API_VER_MINOR	— —	102	Global	GrB_INT32
GrB_API_VER_PATCH	— —	102	Global	GrB_INT32
GrB_BLOCKING_MODE	— —	103	Global	GrB_Mode
GrB_NTHREADS	W H	104	Global, GrB_Descriptor	GrB_INT32
GrB_STORAGE_ORIENTATION_HINT	W H	200	Global, Collection	GrB_ROWMAJOR, GrB_COLMAJOR
GrB_STORAGE_FORMAT_HINT	W H	201	Collection	GrB_Format
GrB_ELTYPE	— —	202	Collection	GrB_Type
GrB_INPUT1TYPE	— —	300	Algebraic	GrB_Type
GrB_INPUT2TYPE	— —	301	Algebraic	GrB_Type
GrB_OUTPUTTYPE	— —	302	Algebraic	GrB_Type
GrB_BINARYOP	— —	303	GrB_Monoid, GrB_Semiring	GrB_BinaryOp
GrB_MONOID	— —	304	GrB_Semiring	GrB_Monoid

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

Field Name	Description
GrB_NAMESIZE	The length of the GrB_NAME for a particular object.
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_NTHREADS	The number of threads for the library to use for a function call, may be ignored by the library.
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_FORMAT_HINT	Hint to the library that it should use a specific storage format.
GrB_ELTYPE	The element type of a collection.
GrB_INPUT1TYPE	The type of the first argument to an operator.
GrB_INPUT2TYPE	The type of the second argument to an operator.
GrB_OUTPUTTYPE	The type of the output of an operator.
GrB_BINARYOP	The binary operator of a semiring or monoid.
GrB_MONOID	The monoid of a semiring.

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 3.15.

Table 3.15: 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`.

1044 **Return Values**

1045 `GrB_SUCCESS` operation completed successfully.

1046 `GrB_PANIC` unknown internal error.

1047 `GrB_INVALID_VALUE` invalid mode specified, or method called multiple times.

1048 **Description**

1049 The `init` method creates and initializes a GraphBLAS C API context. The argument to `GrB_init`
1050 defines the mode for the context. The two available modes are:

- 1051 • `GrB_BLOCKING`: In this mode, each method in a sequence returns after its computations have
1052 completed and output arguments are available to subsequent statements in an application.
1053 When executing in `GrB_BLOCKING` mode, the methods execute in program order.
- 1054 • `GrB_NONBLOCKING`: In this mode, methods in a sequence may return after arguments in
1055 the method have been tested for dimension and domain compatibility within the method
1056 but potentially before their computations complete. Output arguments are available to sub-
1057 sequent GraphBLAS methods in an application. When executing in `GrB_NONBLOCKING`
1058 mode, the methods in a sequence may execute in any order that preserves the mathematical
1059 result defined by the sequence.

1060 An application can only create one context per execution instance. An application may only call
1061 `GrB_Init` once. Calling `GrB_Init` more than once results in undefined behavior.

1062 **4.1.2 finalize: Finalize a GraphBLAS context**

1063 Terminates and frees any internal resources created to support the GraphBLAS C API context.

1064 **C Syntax**

1065 `GrB_Info GrB_finalize();`

1066 **Return Values**

1067 `GrB_SUCCESS` operation completed successfully.

1068 `GrB_PANIC` unknown internal error.

1069 **Description**

1070 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS
1071 C API context. `GrB_finalize` may only be called after a context has been initialized by calling
1072 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-
1073 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined
1074 behavior.

1075 **4.1.3 getVersion: Get the version number of the standard.**

1076 Query the library for the version number of the standard that this library implements.

1077 **C Syntax**

```
1078         GrB_Info GrB_getVersion(unsigned int *version,  
1079                                unsigned int *subversion);
```

1080 **Parameters**

1081 version (OUT) On successful return will hold the value of the major version number.

1082 version (OUT) On successful return will hold the value of the subversion number.

1083 **Return Values**

1084 GrB_SUCCESS operation completed successfully.

1085 GrB_PANIC unknown internal error.

1086 **Description**

1087 The `getVersion` method is used to query the major and minor version number of the GraphBLAS
1088 C API specification that the library implements at runtime. To support compile time queries the
1089 following two macros shall also be defined by the library.

```
1090         #define GRB_VERSION      2  
1091         #define GRB_SUBVERSION  0
```

1092 **4.2 Object methods**

1093 This section describes methods that setup and operate on GraphBLAS opaque objects but are not
1094 part of the the GraphBLAS math specification.

1095 4.2.1 Get and Set methods

1096 The methods in this section query and, optionally, set internal fields of GraphBLAS objects.

1097 4.2.1.1 get: Query the value of an object

1098 C Syntax

```
1099     GrB_Info GrB_<OBJ>_get(GrB_<OBJ> o, GrB_Field field, <type> value);
1100
1101     GrB_Info GrB_Scalar_get(GrB_Scalar s, GrB_Field field, <type> value);
1102     GrB_Info GrB_Vector_get(GrB_Vector v, GrB_Field field, <type> value);
1103     GrB_Info GrB_Matrix_get(GrB_Matrix A, GrB_Field field, <type> value);
1104
1105     GrB_Info GrB_UnaryOp_get(GrB_UnaryOp op, GrB_Field field, <type> value);
1106     GrB_Info GrB_IndexUnaryOp_get(GrB_IndexUnaryOp op, GrB_Field field, <type> value);
1107     GrB_Info GrB_BinaryOp_get(GrB_BinaryOp op, GrB_Field field, <type> value);
1108     GrB_Info GrB_Monoid_get(GrB_Monoid op, GrB_Field field, <type> value);
1109     GrB_Info GrB_Semiring_get(GrB_Semiring op, GrB_Field field, <type> value);
1110
1111     GrB_Info GrB_Descriptor_get(GrB_Descriptor desc, GrB_Field field, <type> value);
1112     GrB_Info GrB_Type_get(GrB_Type type, GrB_Field field, <type> value);
1113
1114     GrB_Info GrB_Global_get(GrB_Field field, <type> value);
```

1115 Parameters

1116 OBJ (IN) An existing, valid GraphBLAS object which is being queried.

1117 field (IN) The field being queried.

1118 value (OUT) A pointer to or GrB_Scalar containing a value whose type is dependent on
1119 field which will be filled with the value of the field. type may be a char*, GrB_Scalar,
1120 GrB_Type, GrB_BinaryOp, GrB_Monoid, or void*.

1121 Return Value

1122 GrB_SUCCESS The method completed successfully.

1123 GrB_PANIC unknown internal error.

1124 GrB_OUT_OF_MEMORY not enough memory available for operation.

1125 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to new.

1126 GrB_INVALID_VALUE invalid value set on the field or invalid field.

1127 Description

1128 Queries a field of an existing GraphBLAS object. The type of the argument is uniquely determined
1129 by field. Fields marked as hints in Table 3.13 will return the hint when queried, not the true internal
1130 value.

1131 4.2.1.2 set: Set field of an object

1132 Set the content for a field for an existing GraphBLAS object.

1133 C Syntax

```
1134 GrB_Info GrB_<OBJ>_set(GrB_<OBJ> o, GrB_Field field, <type> value);
1135
1136 GrB_Info GrB_Scalar_set(GrB_Scalar s, GrB_Field field, <type> value);
1137 GrB_Info GrB_Vector_set(GrB_Vector v, GrB_Field field, <type> value);
1138 GrB_Info GrB_Matrix_set(GrB_Matrix A, GrB_Field field, <type> value);
1139
1140 GrB_Info GrB_UnaryOp_set(GrB_UnaryOp op, GrB_Field field, <type> value);
1141 GrB_Info GrB_IndexUnaryOp_set(GrB_IndexUnaryOp op, GrB_Field field, <type> value);
1142 GrB_Info GrB_BinaryOp_set(GrB_BinaryOp op, GrB_Field field, <type> value);
1143 GrB_Info GrB_Monoid_set(GrB_Monoid op, GrB_Field field, <type> value);
1144 GrB_Info GrB_Semiring_set(GrB_Semiring op, GrB_Field field, <type> value);
1145
1146 GrB_Info GrB_Descriptor_set(GrB_Descriptor op, GrB_Field field, <type> value);
1147 GrB_Info GrB_Type_set(GrB_Type op, GrB_Field field, <type> value);
1148
1149 GrB_Info GrB_Global_set(GrB_Field field, <type> value);
```

1150 Parameters

1151 OBJ (IN) The GraphBLAS object which is having field set.

1152 field (IN) The field being set.

1153 value (IN) A value whose type is dependent on field or a GrB_Scalar. type
1154 may be a char*, GrB_Scalar, GrB_Type, GrB_BinaryOp, GrB_Monoid,
1155 or void*.

1156 Return Values

1157 GrB_SUCCESS The method completed successfully.

1158 GrB_PANIC unknown internal error.

1159 GrB_OUT_OF_MEMORY not enough memory available for operation.

1160 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to new.

1161 GrB_INVALID_VALUE invalid value set on the field, invalid field, or field is read-only.

1162 GrB_ALREADY_SET this field has already been set, and may only be set once.

1163 **Description**

1164 Set a field of OBJ to a new value.

1165 **4.2.2 Algebra methods**

1166 **4.2.2.1 Type_new: Construct a new GraphBLAS (user-defined) type**

1167 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,
1168 monoids, semirings, vectors and matrices.

1169 **C Syntax**

```
1170       GrB_Info GrB_Type_new(GrB_Type *utype,  
1171                                       size_t     sizeof(ctype));
```

1172 **Parameters**

1173 utype (INOUT) On successful return, contains a handle to the newly created user-defined
1174 GraphBLAS type object.

1175 ctype (IN) A C type that defines the new GraphBLAS user-defined type.

1176 **Return Values**

1177 GrB_SUCCESS operation completed successfully.

1178 GrB_PANIC unknown internal error.

1179 GrB_OUT_OF_MEMORY not enough memory available for operation.

1180 GrB_NULL_POINTER utype pointer is NULL.

1181 Description

1182 Given a C type `ctype`, the `Type_new` method returns in `utype` a handle to a new GraphBLAS type
1183 that is equivalent to the C type. Variables of this `ctype` must be a struct, union, or fixed-size array.
1184 In particular, given two variables, `src` and `dst`, of type `ctype`, the following operation must be a
1185 valid way to copy the contents of `src` to `dst`:

```
1186             memcpy(&dst, &src, sizeof(ctype))
```

1187 A new, user-defined type `utype` should be destroyed with a call to `GrB_free(utype)` when no longer
1188 needed.

1189 It is not an error to call this method more than once on the same variable; however, the handle to
1190 the previously created object will be overwritten.

1191 4.2.2.2 UnaryOp_new: Construct a new GraphBLAS unary operator

1192 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types
1193 (domains).

1194 C Syntax

```
1195     GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1196                             void          (*unary_func)(void*, const void*),  
1197                             GrB_Type      d_out,  
1198                             GrB_Type      d_in);
```

1199 Parameters

1200 `unary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1201 unary operator object.

1202 `unary_func` (IN) a pointer to a user-defined function that takes one input parameter of `d_in`'s
1203 type and returns a value of `d_out`'s type, both passed as `void` pointers. Specifically
1204 the signature of the function is expected to be of the form:

```
1205             void func(void *out, const void *in);  
1206
```

1207 `d_out` (IN) The `GrB_Type` of the return value of the unary operator being created. Should
1208 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1209 BLAS type.

1210 `d_in` (IN) The `GrB_Type` of the input argument of the unary operator being created.
1211 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1212 GraphBLAS type.

1213 **Return Values**

1214 `GrB_SUCCESS` operation completed successfully.

1215 `GrB_PANIC` unknown internal error.

1216 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1217 `GrB_UNINITIALIZED_OBJECT` any `GrB_Type` parameter (for user-defined types) has not been ini-
1218 tialized by a call to `GrB_Type_new`.

1219 `GrB_NULL_POINTER` `unary_op` or `unary_func` pointers are `NULL`.

1220 **Description**

1221 The `UnaryOp_new` method creates a new GraphBLAS unary operator

1222 $f_u = \langle \mathbf{D}(\mathbf{d_out}), \mathbf{D}(\mathbf{d_in}), \text{unary_func} \rangle$

1223 and returns a handle to it in `unary_op`.

1224 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments
1225 are aliased. In other words, for all invocations of the function:

1226 `unary_func(out,in);`

1227 the value of `out` must be the same as if the following code was executed:

```
1228 D(d_in) *tmp = malloc(sizeof(D(d_in)));  
1229 memcpy(tmp,in,sizeof(D(d_in)));  
1230 unary_func(out,tmp);  
1231 free(tmp);
```

1232 It is not an error to call this method more than once on the same variable; however, the handle to
1233 the previously created object will be overwritten.

1234 **4.2.2.3 BinaryOp_new: Construct a new GraphBLAS binary operator**

1235 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types
1236 (domains).

1237 **C Syntax**

```
1238 GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
1239 void (*binary_func)(void*,
```

```

1240                                     const void*,
1241                                     const void*),
1242         GrB_Type          d_out,
1243         GrB_Type          d_in1,
1244         GrB_Type          d_in2);

```

1245 Parameters

1246 **binary_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1247 binary operator object.

1248 **binary_func** (IN) A pointer to a user-defined function that takes two input parameters of types
1249 **d_in1** and **d_in2** and returns a value of type **d_out**, all passed as void pointers.
1250 Specifically the signature of the function is expected to be of the form:

```

1251         void func(void *out, const void *in1, const void *in2);
1252

```

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

1256 **d_in1** (IN) The **GrB_Type** of the left hand argument of the binary operator being created.
1257 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1258 GraphBLAS type.

1259 **d_in2** (IN) The **GrB_Type** of the right hand argument of the binary operator being cre-
1260 ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-
1261 defined GraphBLAS type.

1262 Return Values

1263 **GrB_SUCCESS** operation completed successfully.

1264 **GrB_PANIC** unknown internal error.

1265 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1266 **GrB_UNINITIALIZED_OBJECT** the **GrB_Type** (for user-defined types) has not been initialized by a
1267 call to **GrB_Type_new**.

1268 **GrB_NULL_POINTER** **binary_op** or **binary_func** pointer is NULL.

1269 Description

1270 The **BinaryOp_new** methods creates a new GraphBLAS binary operator

1271 $f_b = \langle \mathbf{D}(d_out), \mathbf{D}(d_in1), \mathbf{D}(d_in2), \text{binary_func} \rangle$

1272 and returns a handle to it in `binary_op`.

1273 The implementation of `binary_func` must be such that it works even if any of the `d_out`, `d_in1`, and
1274 `d_in2` arguments are aliased to each other. In other words, for all invocations of the function:

1275 `binary_func(out, in1, in2);`

1276 the value of `out` must be the same as if the following code was executed:

```
1277 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));  
1278 D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));  
1279 memcpy(tmp1, in1, sizeof(D(d_in1)));  
1280 memcpy(tmp2, in2, sizeof(D(d_in2)));  
1281 binary_func(out, tmp1, tmp2);  
1282 free(tmp2);  
1283 free(tmp1);
```

1284 It is not an error to call this method more than once on the same variable; however, the handle to
1285 the previously created object will be overwritten.

1286 **4.2.2.4 Monoid_new: Construct a new GraphBLAS monoid**

1287 Creates a new monoid with specified binary operator and identity value.

1288 **C Syntax**

```
1289 GrB_Info GrB_Monoid_new(GrB_Monoid *monoid,  
1290                         GrB_BinaryOp binary_op,  
1291                         <type> identity);
```

1292 **Parameters**

1293 `monoid` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1294 `monoid` object.

1295 `binary_op` (IN) An existing GraphBLAS associative binary operator whose input and output
1296 types are the same.

1297 `identity` (IN) The value of the identity element of the monoid. Must be the same type as
1298 the type used by the `binary_op` operator.

1299 Return Values

1300 GrB_SUCCESS operation completed successfully.

1301 GrB_PANIC unknown internal error.

1302 GrB_OUT_OF_MEMORY not enough memory available for operation.

1303 GrB_UNINITIALIZED_OBJECT the GrB_BinaryOp (for user-defined operators) has not been initial-
1304 ized by a call to GrB_BinaryOp_new.

1305 GrB_NULL_POINTER monoid pointer is NULL.

1306 GrB_DOMAIN_MISMATCH all three argument types of the binary operator and the type of the
1307 identity value are not the same.

1308 Description

1309 The Monoid_new method creates a new monoid $M = \langle \mathbf{D}(\text{binary_op}), \text{binary_op}, \text{identity} \rangle$ and re-
1310 turns a handle to it in monoid.

1311 If binary_op is not associative, the results of GraphBLAS operations that require associativity of
1312 this monoid will be undefined.

1313 It is not an error to call this method more than once on the same variable; however, the handle to
1314 the previously created object will be overwritten.

1315 4.2.2.5 Semiring_new: Construct a new GraphBLAS semiring

1316 Creates a new semiring with specified domain, operators, and elements.

1317 C Syntax

```
1318           GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1319                                    GrB_Monoid     add_op,  
1320                                    GrB_BinaryOp   mul_op);
```

1321 Parameters

1322 semiring (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1323 semiring.

1324 add_op (IN) An existing GraphBLAS commutative monoid that specifies the addition op-
1325 erator and its identity.

1326 mul_op (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-
1327 plication operator. In addition, mul_op's output domain, $\mathbf{D}_{out}(\text{mul_op})$, must be
1328 the same as the add_op's domain $\mathbf{D}(\text{add_op})$.

1329 Return Values

1330 GrB_SUCCESS operation completed successfully.

1331 GrB_PANIC unknown internal error.

1332 GrB_OUT_OF_MEMORY not enough memory available for this method to complete.

1333 GrB_UNINITIALIZED_OBJECT the add_op (for user-define monoids) object has not been initialized
1334 with a call to GrB_Monoid_new or the mul_op (for user-defined
1335 operators) object has not been not been initialized by a call to
1336 GrB_BinaryOp_new.

1337 GrB_NULL_POINTER semiring pointer is NULL.

1338 GrB_DOMAIN_MISMATCH the output domain of mul_op does not match the domain of the
1339 add_op monoid.

1340 Description

1341 The Semiring_new method creates a new semiring:

1342
$$S = \langle \mathbf{D}_{out}(\text{mul_op}), \mathbf{D}_{in_1}(\text{mul_op}), \mathbf{D}_{in_2}(\text{mul_op}), \text{add_op}, \text{mul_op}, \mathbf{0}(\text{add_op}) \rangle$$

1343 and returns a handle to it in semiring. Note that $\mathbf{D}_{out}(\text{mul_op})$ must be the same as $\mathbf{D}(\text{add_op})$.

1344 If add_op is not commutative, then GraphBLAS operations using this semiring 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 4.2.2.6 IndexUnaryOp_new: Construct a new GraphBLAS index unary operator [Scott: 1348 NEW CONTENT]

1349 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its
1350 types (domains).

1351 C Syntax

```
1352 GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp *index_unary_op,  
1353                               void (*index_unary_func)(void*,  
1354                                                         const void*,  
1355                                                         GrB_Index,  
1356                                                         GrB_Index,  
1357                                                         const void*),  
1358                               GrB_Type d_out,  
1359                               GrB_Type d_in1,  
1360                               GrB_Type d_in2);
```


1361 Parameters

1362 `index_unary_op` (INOUT) On successful return, contains a handle to the newly created Graph-
1363 BLAS index unary operator object.

1364 `index_unary_func` (IN) A pointer to a user-defined function that takes input parameters of types
1365 `d_in1`, `GrB_Index`, `GrB_Index` and `d_in2` and returns a value of type `d_out`. Ex-
1366 cept for the `GrB_Index` parameters, all are passed as `void` pointers. Specifically
1367 the signature of the function is expected to be of the form:

```
1368         void func(void      *out,  
1369                   const void *in1,  
1370                   GrB_Index  row_index,  
1371                   GrB_Index  col_index,  
1372                   const void *in2);
```

1374 `d_out` (IN) The `GrB_Type` of the return value of the index unary operator being created.
1375 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1376 GraphBLAS type.

1377 `d_in1` (IN) The `GrB_Type` of the first input argument of the index unary operator being
1378 created and corresponds to the stored values of the `GrB_Vector` or `GrB_Matrix`
1379 being operated on. Should be one of the predefined GraphBLAS types in Ta-
1380 ble 3.2, or a user-defined GraphBLAS type.

1381 `d_in2` (IN) The `GrB_Type` of the last input argument of the index unary operator be-
1382 ing created and corresponds to a scalar provided by the GraphBLAS operation
1383 that uses this operator. Should be one of the predefined GraphBLAS types in
1384 Table 3.2, or a user-defined GraphBLAS type.

1385 Return Values

1386 `GrB_SUCCESS` operation completed successfully.

1387 `GrB_PANIC` unknown internal error.

1388 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1389 `GrB_UNINITIALIZED_OBJECT` the `GrB_Type` (for user-defined types) has not been initialized by a
1390 call to `GrB_Type_new`.

1391 `GrB_NULL_POINTER` `index_unary_op` or `index_unary_func` pointer is `NULL`.

1392 Description

1393 The `IndexUnaryOp_new` methods creates a new GraphBLAS index unary operator

1394 $f_i = \langle \mathbf{D}(\mathbf{d_out}), \mathbf{D}(\mathbf{d_in1}), \mathbf{D}(\mathbf{GrB_Index}), \mathbf{D}(\mathbf{GrB_Index}), \mathbf{D}(\mathbf{d_in2}), \text{index_unary_func} \rangle$

1395 and returns a handle to it in `index_unary_op`.

1396 The implementation of `index_unary_func` must be such that it works even if any of the `d_out`,
 1397 `d_in1`, and `d_in2` arguments are aliased to each other. In other words, for all invocations of the
 1398 function:

1399 `index_unary_func(out, in1, row_index, col_index, n, in2);`

1400 the value of `out` must be the same as if the following code was executed (shown here for matrices):

```
1401 GrB_Index row_index = ...;
1402 GrB_Index col_index = ...;
1403 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1404 D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1405 memcpy(tmp1, in1, sizeof(D(d_in1)));
1406 memcpy(tmp2, in2, sizeof(D(d_in2)));
1407 index_unary_func(out, tmp1, row_index, col_index, tmp2);
1408 free(tmp2);
1409 free(tmp1);
```

1410 It is not an error to call this method more than once on the same variable; however, the handle to
 1411 the previously created object will be overwritten.

1412 4.2.3 Scalar methods

1413 4.2.3.1 Scalar_new: Construct a new scalar

1414 Creates a new empty scalar with specified domain.

1415 C Syntax

```
1416 GrB_Info GrB_Scalar_new(GrB_Scalar *s,
1417                          GrB_Type d);
```

1418 Parameters

1419 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
 1420 scalar.

1421 **d** (IN) The type corresponding to the domain of the scalar being created. Can be
 1422 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
 1423 GraphBLAS type.

1424 Return Values

1425 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1426 blocking mode, this indicates that the API checks for the input
1427 arguments passed successfully. Either way, output scalar `s` is ready
1428 to be used in the next method of the sequence.

1429 GrB_PANIC Unknown internal error.

1430 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1431 GraphBLAS objects (input or output) is in an invalid state caused
1432 by a previous execution error. Call `GrB_error()` to access any error
1433 messages generated by the implementation.

1434 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1435 GrB_UNINITIALIZED_OBJECT The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`
1436 (nEEDED for user-defined types).

1437 GrB_NULL_POINTER The `s` pointer is NULL.

1438 Description

1439 Creates a new GraphBLAS scalar `s` of domain `D(d)` and empty `L(s)`. The method returns a handle
1440 to the new scalar in `s`.

1441 It is not an error to call this method more than once on the same variable; however, the handle to
1442 the previously created object will be overwritten.

1443 4.2.3.2 Scalar_dup: Construct a copy of a GraphBLAS scalar

1444 Creates a new scalar with the same domain and contents as another scalar.

1445 C Syntax

```
1446           GrB_Info GrB_Scalar_dup(GrB_Scalar           *t,  
1447                                   const GrB_Scalar   s);
```

1448 Parameters

1449 t (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1450 scalar.

1451 s (IN) The GraphBLAS scalar to be duplicated.

1452 Return Values

1453 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1454 blocking mode, this indicates that the API checks for the input
1455 arguments passed successfully. Either way, output scalar *t* is ready
1456 to be used in the next method of the sequence.

1457 GrB_PANIC Unknown internal error.

1458 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1459 GraphBLAS objects (input or output) is in an invalid state caused
1460 by a previous execution error. Call `GrB_error()` to access any error
1461 messages generated by the implementation.

1462 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1463 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to
1464 `Scalar_new` or `Scalar_dup`.

1465 GrB_NULL_POINTER The *t* pointer is NULL.

1466 Description

1467 Creates a new scalar *t* of domain $\mathbf{D}(\mathbf{s})$ and contents $\mathbf{L}(\mathbf{s})$. The method returns a handle to the new
1468 scalar in *t*.

1469 It is not an error to call this method more than once with the same output variable; however, the
1470 handle to the previously created object will be overwritten.

1471 4.2.3.3 Scalar_clear: Clear/remove a stored value from a scalar

1472 Removes the stored value from a scalar.

1473 C Syntax

1474 GrB_Info GrB_Scalar_clear(GrB_Scalar s);

1475 Parameters

1476 *s* (INOUT) An existing GraphBLAS scalar to clear.

1477 Return Values

1478 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1479 blocking mode, this indicates that the API checks for the input

1480 arguments passed successfully. Either way, output scalar `s` is ready
 1481 to be used in the next method of the sequence.

1482 `GrB_PANIC` Unknown internal error.

1483 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 1484 GraphBLAS objects (input or output) is in an invalid state caused
 1485 by a previous execution error. Call `GrB_error()` to access any error
 1486 messages generated by the implementation.

1487 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1488 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to
 1489 `Scalar_new` or `Scalar_dup`.

1490 Description

1491 Removes the stored value from an existing scalar. After the call, `L(s)` is empty. The size of the
 1492 scalar does not change.

1493 4.2.3.4 `Scalar_nvals`: Number of stored elements in a scalar

1494 Retrieve the number of stored elements in a scalar (either zero or one).

1495 C Syntax

```
1496 GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,  
1497                          const GrB_Scalar s);
```

1498 Parameters

1499 `nvals` (OUT) On successful return, this is set to the number of stored elements in the
 1500 scalar (zero or one).

1501 `s` (IN) An existing GraphBLAS scalar being queried.

1502 Return Values

1503 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
 1504 cessfully and the value of `nvals` has been set.

1505 `GrB_PANIC` Unknown internal error.

1506 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 1507 GraphBLAS objects (input or output) is in an invalid state caused
 1508 by a previous execution error. Call GrB_error() to access any error
 1509 messages generated by the implementation.

1510 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1511 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to
 1512 Scalar_new or Scalar_dup.

1513 GrB_NULL_POINTER The *nvals* pointer is NULL.

1514 Description

1515 Return *nvals(s)* in *nvals*. This is the number of stored elements in scalar *s*, which is the size of
 1516 *L(s)*, and can only be either zero or one (see Section 3.5.1).

1517 4.2.3.5 Scalar_setElement: Set the single element in a scalar

1518 Set the single element of a scalar to a given value.

1519 C Syntax

```
1520           GrB_Info GrB_Scalar_setElement(GrB_Scalar    s,
1521                                                           <type>    val);
```

1522 Parameters

1523 *s* (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1524 *val* (IN) Scalar value to assign. The type must be compatible with the domain of *s*.

1525 Return Values

1526 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 1527 blocking mode, this indicates that the compatibility tests on in-
 1528 dex/dimensions and domains for the input arguments passed suc-
 1529 cessfully. Either way, the output scalar *s* is ready to be used in the
 1530 next method of the sequence.

1531 GrB_PANIC Unknown internal error.

1532 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 1533 GraphBLAS objects (input or output) is in an invalid state caused

1534 by a previous execution error. Call `GrB_error()` to access any error
1535 messages generated by the implementation.

1536 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1537 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, `s`, has not been initialized by a call to
1538 `Scalar_new` or `Scalar_dup`.

1539 **GrB_DOMAIN_MISMATCH** The domains of `s` and `val` are incompatible.

1540 Description

1541 First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: **D**(`val`) must
1542 be compatible with **D**(`s`). Two domains are compatible with each other if values from one domain
1543 can be cast to values in the other domain as per the rules of the C language. In particular, domains
1544 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-
1545 patible with itself. If any compatibility rule above is violated, execution of `GrB_Scalar_setElement`
1546 ends and the domain mismatch error listed above is returned.

1547 We are now ready to carry out the assignment `val`; that is:

$$1548 \quad s(0) = val$$

1549 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1550 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1551 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1552 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be
1553 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1554 4.2.3.6 `Scalar_extractElement`: Extract a single element from a scalar.

1555 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

1556 C Syntax

```
1557      GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1558                                         const GrB_Scalar s);
```

1559 Parameters

1560 `val` (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain
1561 of scalar `s`. On successful return, `val` holds the result of the operation, and any
1562 previous value in `val` is overwritten.

1563 `s` (IN) The GraphBLAS scalar from which an element is extracted.

1594 4.2.4 Vector methods

1595 4.2.4.1 Vector_new: Construct new vector

1596 Creates a new vector with specified domain and size.

1597 C Syntax

```
1598         GrB_Info GrB_Vector_new(GrB_Vector *v,  
1599                                GrB_Type    d,  
1600                                GrB_Index   nsize);
```

1601 Parameters

1602 v (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1603 vector.

1604 d (IN) The type corresponding to the domain of the vector being created. Can be
1605 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1606 GraphBLAS type.

1607 nsize (IN) The size of the vector being created.

1608 Return Values

1609 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1610 blocking mode, this indicates that the API checks for the input
1611 arguments passed successfully. Either way, output vector v is ready
1612 to be used in the next method of the sequence.

1613 GrB_PANIC Unknown internal error.

1614 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1615 GraphBLAS objects (input or output) is in an invalid state caused
1616 by a previous execution error. Call GrB_error() to access any error
1617 messages generated by the implementation.

1618 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1619 GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to GrB_Type_new
1620 (needed for user-defined types).

1621 GrB_NULL_POINTER The v pointer is NULL.

1622 GrB_INVALID_VALUE nsize is zero or outside the range of the type GrB_Index.

1623 Description

1624 Creates a new vector \mathbf{v} of domain $\mathbf{D(d)}$, size \mathbf{nsz} , and empty $\mathbf{L(v)}$. The method returns a handle
1625 to the new vector in \mathbf{v} .

1626 It is not an error to call this method more than once on the same variable; however, the handle to
1627 the previously created object will be overwritten.

1628 4.2.4.2 Vector_dup: Construct a copy of a GraphBLAS vector

1629 Creates a new vector with the same domain, size, and contents as another vector.

1630 C Syntax

```
1631         GrB_Info GrB_Vector_dup(GrB_Vector      *w,  
1632                               const GrB_Vector  u);
```

1633 Parameters

1634 \mathbf{w} (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1635 vector.

1636 \mathbf{u} (IN) The GraphBLAS vector to be duplicated.

1637 Return Values

1638 $\mathbf{GrB_SUCCESS}$ In blocking mode, the operation completed successfully. In non-
1639 blocking mode, this indicates that the API checks for the input
1640 arguments passed successfully. Either way, output vector \mathbf{w} is ready
1641 to be used in the next method of the sequence.

1642 $\mathbf{GrB_PANIC}$ Unknown internal error.

1643 $\mathbf{GrB_INVALID_OBJECT}$ This is returned in any execution mode whenever one of the opaque
1644 GraphBLAS objects (input or output) is in an invalid state caused
1645 by a previous execution error. Call $\mathbf{GrB_error()}$ to access any error
1646 messages generated by the implementation.

1647 $\mathbf{GrB_OUT_OF_MEMORY}$ Not enough memory available for operation.

1648 $\mathbf{GrB_UNINITIALIZED_OBJECT}$ The GraphBLAS vector, \mathbf{u} , has not been initialized by a call to
1649 $\mathbf{Vector_new}$ or $\mathbf{Vector_dup}$.

1650 $\mathbf{GrB_NULL_POINTER}$ The \mathbf{w} pointer is \mathbf{NULL} .

1651 Description

1652 Creates a new vector \mathbf{w} of domain $\mathbf{D}(\mathbf{u})$, size $\mathbf{size}(\mathbf{u})$, and contents $\mathbf{L}(\mathbf{u})$. The method returns a
1653 handle to the new vector in \mathbf{w} .

1654 It is not an error to call this method more than once on the same variable; however, the handle to
1655 the previously created object will be overwritten.

1656 4.2.4.3 Vector_resize: Resize a vector

1657 Changes the size of an existing vector.

1658 C Syntax

```
1659         GrB_Info GrB_Vector_resize(GrB_Vector  w,  
1660                                   GrB_Index   nsize);
```

1661 Parameters

1662 \mathbf{w} (INOUT) An existing Vector object that is being resized.

1663 \mathbf{nsize} (IN) The new size of the vector. It can be smaller or larger than the current size.

1664 Return Values

1665 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1666 blocking mode, this indicates that the API checks for the input
1667 arguments passed successfully. Either way, output vector \mathbf{w} is ready
1668 to be used in the next method of the sequence.

1669 GrB_PANIC Unknown internal error.

1670 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1671 GraphBLAS objects (input or output) is in an invalid state caused
1672 by a previous execution error. Call `GrB_error()` to access any error
1673 messages generated by the implementation.

1674 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1675 GrB_NULL_POINTER The \mathbf{w} pointer is NULL.

1676 GrB_INVALID_VALUE \mathbf{nsize} is zero or outside the range of the type `GrB_Index`.

1677 Description

1678 Changes the size of w to nsz . The domain $\mathbf{D}(w)$ of vector w remains the same. The contents $\mathbf{L}(w)$
1679 are modified as described below.

1680 Let $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$ when the method is called. When the method returns, $w = \langle \mathbf{D}(w), nsz, \mathbf{L}'(w) \rangle$
1681 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
1682 than or equal to the new vector size (nsz) are dropped.

1683 4.2.4.4 Vector_clear: Clear a vector

1684 Removes all the elements (tuples) from a vector.

1685 C Syntax

1686 `GrB_Info GrB_Vector_clear(GrB_Vector v);`

1687 Parameters

1688 v (INOUT) An existing GraphBLAS vector to clear.

1689 Return Values

1690 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1691 blocking mode, this indicates that the API checks for the input
1692 arguments passed successfully. Either way, output vector v is ready
1693 to be used in the next method of the sequence.

1694 **GrB_PANIC** Unknown internal error.

1695 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1696 GraphBLAS objects (input or output) is in an invalid state caused
1697 by a previous execution error. Call `GrB_error()` to access any error
1698 messages generated by the implementation.

1699 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1700 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, v , has not been initialized by a call to
1701 `Vector_new` or `Vector_dup`.

1702 Description

1703 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,
1704 $\mathbf{L}(v) = \emptyset$. The size of the vector does not change.

1705 4.2.4.5 Vector_size: Size of a vector

1706 Retrieve the size of a vector.

1707 C Syntax

```
1708         GrB_Info GrB_Vector_size(GrB_Index      *nsize,  
1709                                const GrB_Vector  v);
```

1710 Parameters

1711 nsize (OUT) On successful return, is set to the size of the vector.

1712 v (IN) An existing GraphBLAS vector being queried.

1713 Return Values

1714 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1715 cessfully and the value of nsize has been set.

1716 GrB_PANIC Unknown internal error.

1717 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1718 GraphBLAS objects (input or output) is in an invalid state caused
1719 by a previous execution error. Call GrB_error() to access any error
1720 messages generated by the implementation.

1721 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to
1722 Vector_new or Vector_dup.

1723 GrB_NULL_POINTER nsize pointer is NULL.

1724 Description

1725 Return **size**(v) in nsize.

1726 4.2.4.6 Vector_nvals: Number of stored elements in a vector

1727 Retrieve the number of stored elements (tuples) in a vector.

1728 C Syntax

```
1729         GrB_Info GrB_Vector_nvals(GrB_Index      *nvals,  
1730                                const GrB_Vector  v);
```

1731 Parameters

1732 **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)
1733 in the vector.

1734 **v** (IN) An existing GraphBLAS vector being queried.

1735 Return Values

1736 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1737 cessfully and the value of **nvals** has been set.

1738 **GrB_PANIC** Unknown internal error.

1739 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1740 GraphBLAS objects (input or output) is in an invalid state caused
1741 by a previous execution error. Call **GrB_error()** to access any error
1742 messages generated by the implementation.

1743 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1744 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to
1745 **Vector_new** or **Vector_dup**.

1746 **GrB_NULL_POINTER** The **nvals** pointer is **NULL**.

1747 Description

1748 Return **nvals(v)** in **nvals**. This is the number of stored elements in vector **v**, which is the size of
1749 **L(v)** (see Section 3.5.2).

1750 4.2.4.7 Vector_build: Store elements from tuples into a vector

1751 C Syntax

```
1752            GrB_Info GrB_Vector_build(GrB_Vector            w,  
1753                                        const GrB_Index        *indices,  
1754                                        const <type>           *values,  
1755                                        GrB_Index                n,  
1756                                        const GrB_BinaryOp       dup);
```

1757 Parameters

1758 **w** (INOUT) An existing Vector object to store the result.

1759 **indices** (IN) Pointer to an array of indices.

1760 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of
 1761 vector **w**.

1762 **n** (IN) The number of entries contained in each array (the same for indices and values).

1763 **dup** (IN) An associative and commutative binary operator to apply when duplicate
 1764 values for the same location are present in the input arrays. All three domains of
 1765 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
 1766 then duplicate locations will result in an error.

1767 Return Values

1768 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 1769 blocking mode, this indicates that the API checks for the input
 1770 arguments passed successfully. Either way, output vector **w** is
 1771 ready to be used in the next method of the sequence.

1772 **GrB_PANIC** Unknown internal error.

1773 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
 1774 opaque GraphBLAS objects (input or output) is in an invalid
 1775 state caused by a previous execution error. Call **GrB_error()** to
 1776 access any error messages generated by the implementation.

1777 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1778 **GrB_UNINITIALIZED_OBJECT** Either **w** has not been initialized by a call to **GrB_Vector_new**
 1779 or by **GrB_Vector_dup**, or **dup** has not been initialized by a call
 1780 to **GrB_BinaryOp_new**.

1781 **GrB_NULL_POINTER** indices or values pointer is **NULL**.

1782 **GrB_INDEX_OUT_OF_BOUNDS** A value in indices is outside the allowed range for **w**.

1783 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
 1784 not all the same, or the domains of **values** and **w** are incompatible
 1785 with each other or D_{dup} .

1786 **GrB_OUTPUT_NOT_EMPTY** Output vector **w** already contains valid tuples (elements). In
 1787 other words, **GrB_Vector_nvals(C)** returns a positive value.

1788 **GrB_INVALID_VALUE** indices contains a duplicate location and **dup** is **GrB_NULL**.

1789 Description

1790 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
 1791 from **w** in its domain; otherwise, $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$.

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

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

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

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

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

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

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

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

4.2.4.8 Vector_setElement: Set a single element in a vector

Set one element of a vector to a given value.

C Syntax

```
// scalar value
GrB_Info GrB_Vector_setElement(GrB_Vector      w,
                               <type>         val,
                               GrB_Index       index);

// GraphBLAS scalar
GrB_Info GrB_Vector_setElement(GrB_Vector      w,
                               const GrB_Scalar s,
                               GrB_Index       index);
```

Parameters

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

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

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

1820 Return Values

1821 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1822 blocking mode, this indicates that the compatibility tests on in-
1823 dex/dimensions and domains for the input arguments passed suc-
1824 cessfully. Either way, the output vector **w** is ready to be used in
1825 the next method of the sequence.

1826 **GrB_PANIC** Unknown internal error.

1827 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1828 GraphBLAS objects (input or output) is in an invalid state caused
1829 by a previous execution error. Call **GrB_error()** to access any error
1830 messages generated by the implementation.

1831 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1832 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been
1833 initialized by a call to a respective constructor.

1834 **GrB_INVALID_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1835 **GrB_DOMAIN_MISMATCH** The domains of the vector and the scalar are incompatible.

1836 Description

1837 First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)**
1838 must be compatible with **D(w)**. Two domains are compatible with each other if values from
1839 one domain can be cast to values in the other domain as per the rules of the C language. In
1840 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1841 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1842 **GrB_Vector_setElement** ends and the domain mismatch error listed above is returned.

1843 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1844 \qquad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1845 If this condition is violated, execution of **GrB_Vector_setElement** ends and the invalid index error
1846 listed above is returned.

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

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

1847 In the case of a transparent scalar or if **L(s)** is not empty, then a value will be stored at the
1848 specified location in **w**, overwriting any value that may have been stored there before. In the case
1849 of a GraphBLAS scalar, if **L(s)** is empty, then any value stored at the specified location in **w** will
1850 be removed.

1851 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
 1852 of **w** is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
 1853 return value GrB_SUCCESS and the new contents of vector **w** is as defined above but may not be
 1854 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1855 4.2.4.9 Vector_removeElement: Remove an element from a vector

1856 Remove (annihilate) one stored element from a vector.

1857 C Syntax

```
1858      GrB_Info GrB_Vector_removeElement(GrB_Vector  w,
1859                                     GrB_Index    index);
```

1860 Parameters

1861 **w** (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1862 **index** (IN) The location of the element to be removed.

1863 Return Values

1864 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 1865 blocking mode, this indicates that the compatibility tests on in-
 1866 dex/dimensions and domains for the input arguments passed suc-
 1867 cessfully. Either way, the output vector **w** is ready to be used in
 1868 the next method of the sequence.

1869 **GrB_PANIC** Unknown internal error.

1870 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 1871 GraphBLAS objects (input or output) is in an invalid state caused
 1872 by a previous execution error. Call **GrB_error()** to access any error
 1873 messages generated by the implementation.

1874 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1875 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **w**, has not been initialized by a call to
 1876 **Vector_new** or **Vector_dup**.

1877 **GrB_INVALID_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1878 Description

1879 First, the `index` parameter is checked for a valid value where the following condition must hold:

$$1880 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1881 If this condition is violated, execution of `GrB_Vector_removeElement` ends and the invalid index
1882 error listed above is returned.

1883 We are now ready to carry out the removal of a value that may be stored at the location specified
1884 by `index`. If a value does not exist at the specified location in \mathbf{w} , no error is reported and the
1885 operation has no effect on the state of \mathbf{w} . In either case, the following will be true on return from
1886 the method: `index` \notin `ind(w)`.

1887 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1888 of \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1889 return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not be
1890 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1891 4.2.4.10 Vector_extractElement: Extract a single element from a vector.

1892 Extract one element of a vector into a scalar.

1893 C Syntax

```
1894 // scalar value
1895 GrB_Info GrB_Vector_extractElement(<type>          *val,
1896                                   const GrB_Vector u,
1897                                   GrB_Index         index);
1898
1899 // GraphBLAS scalar
1900 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1901                                   const GrB_Vector u,
1902                                   GrB_Index         index);
```

1903 Parameters

1904 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector
1905 `u`. On successful return, this scalar holds the result of the extract. Any previous
1906 value stored in `val` or `s` is overwritten.

1907 `u` (IN) The GraphBLAS vector from which an element is extracted.

1908 `index` (IN) The location in `u` to extract.

1909 Return Values

- 1910 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1911 cessfully. This indicates that the compatibility tests on dimensions
1912 and domains for the input arguments passed successfully, and the
1913 output scalar, **val** or **s**, has been computed and is ready to be used
1914 in the next method of the sequence.
- 1915 GrB_NO_VALUE When using the transparent scalar, **val**, this is returned when there
1916 is no stored value at specified location.
- 1917 GrB_PANIC Unknown internal error.
- 1918 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1919 GraphBLAS objects (input or output) is in an invalid state caused
1920 by a previous execution error. Call **GrB_error()** to access any error
1921 messages generated by the implementation.
- 1922 GrB_OUT_OF_MEMORY Not enough memory available for operation.
- 1923 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, **u**, or scalar, **s**, has not been initialized by
1924 a call to a corresponding constructor.
- 1925 GrB_NULL_POINTER **val** pointer is NULL.
- 1926 GrB_INVALID_INDEX **index** specifies a location that is outside the dimensions of **w**.
- 1927 GrB_DOMAIN_MISMATCH The domains of the vector and scalar are incompatible.

1928 Description

1929 First, the scalar and input vector are tested for domain compatibility as follows: **D(val)** or **D(s)**
1930 must be compatible with **D(u)**. Two domains are compatible with each other if values from
1931 one domain can be cast to values in the other domain as per the rules of the C language. In
1932 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1933 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1934 **GrB_Vector_extractElement** ends and the domain mismatch error listed above is returned.

1935 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1936 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1937 If this condition is violated, execution of **GrB_Vector_extractElement** ends and the invalid index
1938 error listed above is returned.

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

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

1939 If $\text{index} \in \mathbf{ind}(u)$, then the corresponding value from u is copied into s or val with casting as
 1940 necessary. If $\text{index} \notin \mathbf{ind}(u)$, then one of the follow occurs depending on output scalar type:

- 1941 • The GraphBLAS scalar, s , is cleared and `GrB_SUCCESS` is returned.
- 1942 • The non-opaque scalar, val , is unchanged, and `GrB_NO_VALUE` is returned.

1943 When using the non-opaque scalar variant (val) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
 1944 mode, the new contents of val are as defined above if the method exits with return value `GrB_SUCCESS`
 1945 or `GrB_NO_VALUE`.

1946 When using the GraphBLAS scalar variant (s) with a `GrB_SUCCESS` return value, the method
 1947 exits and the new contents of s is as defined above and fully computed in `GrB_BLOCKING` mode.
 1948 In `GrB_NONBLOCKING` mode, the new contents of s is as defined above but may not be fully
 1949 computed; however, it can be used in the next GraphBLAS method call in a sequence.

1950 4.2.4.11 Vector_extractTuples: Extract tuples from a vector

1951 Extract the contents of a GraphBLAS vector into non-opaque data structures.

1952 C Syntax

```
1953      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,
1954                                     <type>          *values,
1955                                     GrB_Index        *n,
1956                                     const GrB_Vector  v);
1957
```

1958 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored
 1959 values' indices.

1960 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
 1961 the stored values whose type is compatible with $\mathbf{D}(v)$.

1962 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the
 1963 values and indices arrays can hold. Upon return, it will contain the number of
 1964 values written to the arrays.

1965 **v** (IN) An existing GraphBLAS vector.

1966 Return Values

1967 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
 1968 cessfully. This indicates that the compatibility tests on the input
 1969 argument passed successfully, and the output arrays, **indices** and
 1970 **values**, have been computed.

1971 GrB_PANIC Unknown internal error.

1972 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1973 GraphBLAS objects (input or output) is in an invalid state caused
1974 by a previous execution error. Call GrB_error() to access any error
1975 messages generated by the implementation.

1976 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1977 GrB_INSUFFICIENT_SPACE Not enough space in `indices` and `values` (as indicated by the `n` pa-
1978 rameter) to hold all of the tuples that will be extracted.

1979 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to
1980 Vector_new or Vector_dup.

1981 GrB_NULL_POINTER `indices`, `values`, or `n` pointer is NULL.

1982 GrB_DOMAIN_MISMATCH The domains of the `v` vector or `values` array are incompatible with
1983 one another.

1984 Description

1985 This method will extract all the tuples from the GraphBLAS vector `v`. The values associated
1986 with those tuples are placed in the `values` array and the indices are placed in the `indices` array.
1987 Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least
1988 GrB_Vector_nvals(`v`) elements before calling this function.

1989 Upon return of this function, `n` will be set to the number of values (and indices) copied. Also, the
1990 entries of `indices` are unique, but not necessarily sorted. Each tuple (i, v_i) in `v` is unzipped and
1991 copied into a distinct k th location in output vectors:

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

1992 where $0 \leq k < \text{GrB_Vector_nvals}(v)$. No gaps in output vectors are allowed; that is, if `indices`[k]
1993 and `values`[k] exist upon return, so does `indices`[j] and `values`[j] for all j such that $0 \leq j < k$.

1994 Note that if the value in `n` on input is less than the number of values contained in the vector `v`,
1995 then a GrB_INSUFFICIENT_SPACE error is returned because it is undefined which subset of values
1996 would be extracted otherwise.

1997 In both GrB_BLOCKING mode GrB_NONBLOCKING mode if the method exits with return value
1998 GrB_SUCCESS, the new contents of the arrays `indices` and `values` are as defined above.

1999 4.2.5 Matrix methods

2000 4.2.5.1 Matrix_new: Construct new matrix

2001 Creates a new matrix with specified domain and dimensions.

2002 C Syntax

```
2003         GrB_Info GrB_Matrix_new(GrB_Matrix *A,  
2004                                 GrB_Type      d,  
2005                                 GrB_Index    nrows,  
2006                                 GrB_Index    ncols);
```

2007 Parameters

2008 **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2009 matrix.

2010 **d** (IN) The type corresponding to the domain of the matrix being created. Can be
2011 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
2012 GraphBLAS type.

2013 **nrows** (IN) The number of rows of the matrix being created.

2014 **ncols** (IN) The number of columns of the matrix being created.

2015 Return Values

2016 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2017 blocking mode, this indicates that the API checks for the input ar-
2018 guments passed successfully. Either way, output matrix **A** is ready
2019 to be used in the next method of the sequence.

2020 **GrB_PANIC** Unknown internal error.

2021 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2022 GraphBLAS objects (input or output) is in an invalid state caused
2023 by a previous execution error. Call **GrB_error()** to access any error
2024 messages generated by the implementation.

2025 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2026 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
2027 (needed for user-defined types).

2028 **GrB_NULL_POINTER** The **A** pointer is **NULL**.

2029 **GrB_INVALID_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB_Index**.

2030 Description

2031 Creates a new matrix **A** of domain **D**(**d**), size **nrows** \times **ncols**, and empty **L**(**A**). The method returns
2032 a handle to the new matrix in **A**.

2033 It is not an error to call this method more than once on the same variable; however, the handle to
2034 the previously created object will be overwritten.

2035 **4.2.5.2 Matrix_dup: Construct a copy of a GraphBLAS matrix**

2036 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

2037 **C Syntax**

```
2038         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
2039                                const GrB_Matrix  A);
```

2040 **Parameters**

2041 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2042 matrix.

2043 A (IN) The GraphBLAS matrix to be duplicated.

2044 **Return Values**

2045 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2046 blocking mode, this indicates that the API checks for the input
2047 arguments passed successfully. Either way, output matrix C is ready
2048 to be used in the next method of the sequence.

2049 GrB_PANIC Unknown internal error.

2050 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2051 GraphBLAS objects (input or output) is in an invalid state caused
2052 by a previous execution error. Call GrB_error() to access any error
2053 messages generated by the implementation.

2054 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2055 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2056 any matrix constructor.

2057 GrB_NULL_POINTER The C pointer is NULL.

2058 **Description**

2059 Creates a new matrix **C** of domain **D(A)**, size **nrows(A) × ncols(A)**, and contents **L(A)**. It returns
2060 a handle to it in C.

2061 It is not an error to call this method more than once on the same variable; however, the handle to
2062 the previously created object will be overwritten.

2063 4.2.5.3 Matrix_diag: Construct a diagonal GraphBLAS matrix

2064 Creates a new matrix with the same domain and contents as a GrB_Vector, and square dimensions
2065 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

2066 C Syntax

```
2067         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
2068                                 const GrB_Vector  v,  
2069                                 int64_t           k);
```

2070 Parameters

2071 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2072 matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

2073 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the
2074 matrix.

2075 k (IN) The diagonal to which the vector is assigned. k = 0 represents the main
2076 diagonal, k > 0 is above the main diagonal, and k < 0 is below.

2077 Return Values

2078 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2079 blocking mode, this indicates that the API checks for the input
2080 arguments passed successfully. Either way, output matrix C is ready
2081 to be used in the next method of the sequence.

2082 GrB_PANIC Unknown internal error.

2083 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2084 GraphBLAS objects (input or output) is in an invalid state caused
2085 by a previous execution error. Call GrB_error() to access any error
2086 messages generated by the implementation.

2087 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2088 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to
2089 Vector_new or Vector_dup.

2090 GrB_NULL_POINTER The C pointer is NULL.

2091 Description

2092 Creates a new matrix **C** of domain **D(v)**, size $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$, and contents

$$\begin{aligned} 2093 \quad \mathbf{L}(\mathbf{C}) &= \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or} \\ 2094 \quad \mathbf{L}(\mathbf{C}) &= \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0. \end{aligned}$$

2095 It returns a handle to it in **C**. It is not an error to call this method more than once on the same
2096 variable; however, the handle to the previously created object will be overwritten.

2097 4.2.5.4 Matrix_resize: Resize a matrix

2098 Changes the dimensions of an existing matrix.

2099 C Syntax

```
2100      GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
2101                                GrB_Index  nrows,  
2102                                GrB_Index  ncols);
```

2103 Parameters

2104 **C** (INOUT) An existing Matrix object that is being resized.

2105 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the
2106 current number of rows.

2107 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than
2108 the current number of columns.

2109 Return Values

2110 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2111 blocking mode, this indicates that the API checks for the input
2112 arguments passed successfully. Either way, output matrix **C** is ready
2113 to be used in the next method of the sequence.

2114 **GrB_PANIC** Unknown internal error.

2115 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2116 GraphBLAS objects (input or output) is in an invalid state caused
2117 by a previous execution error. Call **GrB_error()** to access any error
2118 messages generated by the implementation.

2119 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2120 GrB_NULL_POINTER The C pointer is NULL.

2121 GrB_INVALID_VALUE nrows or ncols is zero or outside the range of the type GrB_Index.

2122 **Description**

2123 Changes the number of rows and columns of C to nrows and ncols, respectively. The domain $\mathbf{D}(\mathbf{C})$
2124 of matrix C remains the same. The contents $\mathbf{L}(\mathbf{C})$ are modified as described below.

2125 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
2126 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
2127 greater than or equal to ncols are dropped.
2128

2129 **4.2.5.5 Matrix_clear: Clear a matrix**

2130 Removes all elements (tuples) from a matrix.

2131 **C Syntax**

2132 GrB_Info GrB_Matrix_clear(GrB_Matrix A);

2133 **Parameters**

2134 A (IN) An existing GraphBLAS matrix to clear.

2135 **Return Values**

2136 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2137 blocking mode, this indicates that the API checks for the input ar-
2138 guments passed successfully. Either way, output matrix A is ready
2139 to be used in the next method of the sequence.

2140 GrB_PANIC Unknown internal error.

2141 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2142 GraphBLAS objects (input or output) is in an invalid state caused
2143 by a previous execution error. Call GrB_error() to access any error
2144 messages generated by the implementation.

2145 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2146 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2147 any matrix constructor.

2148 **Description**

2149 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,
2150 $\mathbf{L}(\mathbf{A}) = \emptyset$. The dimensions of the matrix do not change.

2151 **4.2.5.6 Matrix_nrows: Number of rows in a matrix**

2152 Retrieve the number of rows in a matrix.

2153 **C Syntax**

```
2154         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2155                                   const GrB_Matrix A);
```

2156 **Parameters**

2157 nrows (OUT) On successful return, contains the number of rows in the matrix.

2158 A (IN) An existing GraphBLAS matrix being queried.

2159 **Return Values**

2160 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2161 cessfully and the value of `nrows` has been set.

2162 GrB_PANIC Unknown internal error.

2163 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2164 GraphBLAS objects (input or output) is in an invalid state caused
2165 by a previous execution error. Call `GrB_error()` to access any error
2166 messages generated by the implementation.

2167 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
2168 any matrix constructor.

2169 GrB_NULL_POINTER `nrows` pointer is NULL.

2170 **Description**

2171 Return `nrows(A)` in `nrows` (the number of rows).

2172 **4.2.5.7 Matrix_ncols: Number of columns in a matrix**

2173 Retrieve the number of columns in a matrix.

2174 C Syntax

```
2175         GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2176                                   const GrB_Matrix A);
```

2177 Parameters

2178 ncols (OUT) On successful return, contains the number of columns in the matrix.

2179 A (IN) An existing GraphBLAS matrix being queried.

2180 Return Values

2181 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2182 cessfully and the value of ncols has been set.

2183 GrB_PANIC Unknown internal error.

2184 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2185 GraphBLAS objects (input or output) is in an invalid state caused
2186 by a previous execution error. Call GrB_error() to access any error
2187 messages generated by the implementation.

2188 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2189 any matrix constructor.

2190 GrB_NULL_POINTER ncols pointer is NULL.

2191 Description

2192 Return **ncols(A)** in ncols (the number of columns).

2193 4.2.5.8 Matrix_nvals: Number of stored elements in a matrix

2194 Retrieve the number of stored elements (tuples) in a matrix.

2195 C Syntax

```
2196         GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2197                                   const GrB_Matrix A);
```

2198 **Parameters**

2199 **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in
2200 the matrix.

2201 **A** (IN) An existing GraphBLAS matrix being queried.

2202 **Return Values**

2203 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2204 cessfully and the value of **nvals** has been set.

2205 **GrB_PANIC** Unknown internal error.

2206 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2207 GraphBLAS objects (input or output) is in an invalid state caused
2208 by a previous execution error. Call **GrB_error()** to access any error
2209 messages generated by the implementation.

2210 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2211 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2212 any matrix constructor.

2213 **GrB_NULL_POINTER** The **nvals** pointer is **NULL**.

2214 **Description**

2215 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of
2216 **L(A)** (see Section 3.5.3).

2217 **4.2.5.9 Matrix_build: Store elements from tuples into a matrix**

2218 **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);
```

2219 **Parameters**

2220 **C** (INOUT) An existing Matrix object to store the result.

2221 **row_indices** (IN) Pointer to an array of row indices.

2222 **col_indices** (IN) Pointer to an array of column indices.

2223 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of
2224 matrix, **C**.

2225 **n** (IN) The number of entries contained in each array (the same for **row_indices**,
2226 **col_indices**, and **values**).

2227 **dup** (IN) An associative and commutative binary operator to apply when duplicate
2228 values for the same location are present in the input arrays. All three domains of
2229 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
2230 then duplicate locations will result in an error.

2231 Return Values

2232 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2233 blocking mode, this indicates that the API checks for the input
2234 arguments passed successfully. Either way, output matrix **C** is
2235 ready to be used in the next method of the sequence.

2236 **GrB_PANIC** Unknown internal error.

2237 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2238 opaque GraphBLAS objects (input or output) is in an invalid
2239 state caused by a previous execution error. Call **GrB_error()** to
2240 access any error messages generated by the implementation.

2241 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2242 **GrB_UNINITIALIZED_OBJECT** Either **C** has not been initialized by a call to any matrix construc-
2243 tor, or **dup** has not been initialized by a call to **GrB_BinaryOp_new**.

2244 **GrB_NULL_POINTER** **row_indices**, **col_indices** or **values** pointer is **NULL**.

2245 **GrB_INDEX_OUT_OF_BOUNDS** A value in **row_indices** or **col_indices** is outside the allowed range
2246 for **C**.

2247 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
2248 not all the same, or the domains of **values** and **C** are incompatible
2249 with each other or D_{dup} .

2250 **GrB_OUTPUT_NOT_EMPTY** Output matrix **C** already contains valid tuples (elements). In
2251 other words, **GrB_Matrix_nvals(C)** returns a positive value.

2252 **GrB_INVALID_VALUE** indices contains a duplicate location and **dup** is **GrB_NULL**.

2253 Description

2254 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
 2255 only differs from \mathbf{C} in its domain; otherwise, $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$.

2256 Each tuple $\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the
 2257 output in the form of

$$2258 \quad \tilde{\mathbf{C}}(\text{row_indices}[k], \text{col_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2259 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,
 2260 `dup` is used to reduce the values before assignment into $\tilde{\mathbf{C}}$ as follows:

$$2261 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row_indices}[k]=i \wedge \text{col_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2262 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{C}}$ is copied into \mathbf{C} via typecasting its
 2263 values to $\mathbf{D}(\mathbf{C})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

2264 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2265 It is an error to call this function on an output object with existing elements. In other words,
 2266 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2267 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,
 2268 `col_indices`, or `values`.

2269 4.2.5.10 Matrix_setElement: Set a single element in matrix

2270 Set one element of a matrix to a given value.

2271 C Syntax

```
2272 // scalar value
2273 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2274                               <type>         val,
2275                               GrB_Index        row_index,
2276                               GrB_Index        col_index);
2277
2278 // GraphBLAS scalar
2279 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2280                               const GrB_Scalar s,
2281                               GrB_Index        row_index,
2282                               GrB_Index        col_index);
```


2283 Parameters

2284 **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.
2285 **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of
2286 **C**.
2287 **row_index** (IN) Row index of element to be assigned
2288 **col_index** (IN) Column index of element to be assigned

2289 Return Values

2290 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2291 blocking mode, this indicates that the compatibility tests on in-
2292 dex/dimensions and domains for the input arguments passed suc-
2293 cessfully. Either way, the output matrix **C** is ready to be used in
2294 the next method of the sequence.
2295 **GrB_PANIC** Unknown internal error.
2296 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2297 GraphBLAS objects (input or output) is in an invalid state caused
2298 by a previous execution error. Call **GrB_error()** to access any error
2299 messages generated by the implementation.
2300 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
2301 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been
2302 initialized by a call to a respective constructor.
2303 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e., not less
2304 than **nrows(C)** or **ncols(C)**, respectively).
2305 **GrB_DOMAIN_MISMATCH** The domains of the matrix and the scalar are incompatible.

2306 Description

2307 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or
2308 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from
2309 one domain can be cast to values in the other domain as per the rules of the C language. In
2310 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
2311 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
2312 **GrB_Matrix_setElement** ends and the domain mismatch error listed above is returned.

2313 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2314 \quad & 0 \leq \text{row_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2315 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid
 2316 index error listed above is returned.

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

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

2317 In the case of a transparent scalar or if $\mathbf{L}(s)$ is not empty, then a value will be stored at the
 2318 specified location in C , overwriting any value that may have been stored there before. In the case
 2319 of a GraphBLAS scalar and if $\mathbf{L}(s)$ is empty, then any value stored at the specified location in C
 2320 will be removed.

2321 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
 2322 of C is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
 2323 return value `GrB_SUCCESS` and the new content of vector C is as defined above but may not be
 2324 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2325 **4.2.5.11 Matrix_removeElement: Remove an element from a matrix**

2326 Remove (annihilate) one stored element from a matrix.

2327 **C Syntax**

```
2328      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2329                                         GrB_Index   row_index,
2330                                         GrB_Index   col_index);
```

2331 **Parameters**

2332 C (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2333 `row_index` (IN) Row index of element to be removed

2334 `col_index` (IN) Column index of element to be removed

2335 **Return Values**

2336 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 2337 blocking mode, this indicates that the compatibility tests on in-
 2338 dex/dimensions and domains for the input arguments passed suc-
 2339 cessfully. Either way, the output matrix C is ready to be used in
 2340 the next method of the sequence.

2341 `GrB_PANIC` Unknown internal error.

2342 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 2343 GraphBLAS objects (input or output) is in an invalid state caused
 2344 by a previous execution error. Call GrB_error() to access any error
 2345 messages generated by the implementation.

2346 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2347 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to
 2348 any matrix constructor.

2349 GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e., not less
 2350 than **nrows(C)** or **ncols(C)**, respectively).

2351 Description

2352 First, both index parameters are checked for valid values where following conditions must hold:

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

2354 If either of these conditions is violated, execution of GrB_Matrix_removeElement ends and the
 2355 invalid index error listed above is returned.

2356 We are now ready to carry out the removal of a value that may be stored at the location specified by
 2357 (row_index, col_index). If a value does not exist at the specified location in C, no error is reported
 2358 and the operation has no effect on the state of C. In either case, the following will be true on return
 2359 from this method: (row_index, col_index) \notin ind(C)

2360 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
 2361 of C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
 2362 return value GrB_SUCCESS and the new content of vector C is as defined above but may not be
 2363 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2364 4.2.5.12 Matrix_extractElement: Extract a single element from a matrix

2365 Extract one element of a matrix into a scalar.

2366 C Syntax

```

2367 // scalar value
2368 GrB_Info GrB_Matrix_extractElement(<type>          *val,
2369                                   const GrB_Matrix  A,
2370                                   GrB_Index         row_index,
2371                                   GrB_Index         col_index);
2372
2373 // GraphBLAS scalar

```

```

2374         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2375                                           const GrB_Matrix A,
2376                                           GrB_Index      row_index,
2377                                           GrB_Index      col_index);
2378

```

2379 Parameters

2380 **val or s (INOUT)** An existing scalar whose domain is compatible with the domain of matrix
2381 A. On successful return, this scalar holds the result of the extract. Any previous
2382 value stored in **val** or **s** is overwritten.

2383 **A (IN)** The GraphBLAS matrix from which an element is extracted.

2384 **row_index (IN)** The row index of location in **A** to extract.

2385 **col_index (IN)** The column index of location in **A** to extract.

2386 Return Values

2387 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2388 cessfully. This indicates that the compatibility tests on dimensions
2389 and domains for the input arguments passed successfully, and the
2390 output scalar, **val** or **s**, has been computed and is ready to be used
2391 in the next method of the sequence.

2392 **GrB_NO_VALUE** When using the transparent scalar, **val**, this is returned when there
2393 is no stored value at specified location.

2394 **GrB_PANIC** Unknown internal error.

2395 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2396 GraphBLAS objects (input or output) is in an invalid state caused
2397 by a previous execution error. Call **GrB_error()** to access any error
2398 messages generated by the implementation.

2399 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2400 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by
2401 a call to a corresponding constructor.

2402 **GrB_NULL_POINTER** **val** pointer is **NULL**.

2403 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e. less than
2404 zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-
2405 tively).

2406 **GrB_DOMAIN_MISMATCH** The domains of the matrix and scalar are incompatible.

2407 Description

2408 First, the scalar and input matrix are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ or $\mathbf{D}(\mathbf{s})$
 2409 must be compatible with $\mathbf{D}(\mathbf{A})$. Two domains are compatible with each other if values from
 2410 one domain can be cast to values in the other domain as per the rules of the C language. In
 2411 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
 2412 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
 2413 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2414 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2415 \quad & 0 \leq \text{row_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2416 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index
 2417 error listed above is returned.

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

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

2418 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 val
 2419 with casting as necessary. If $(\text{row_index}, \text{col_index}) \notin \mathbf{ind}(\mathbf{A})$, then one of the follow occurs
 2420 depending on output scalar type:

- 2421 • The GraphBLAS scalar, \mathbf{s} , is cleared and `GrB_SUCCESS` is returned.
- 2422 • The non-opaque scalar, val , is unchanged, and `GrB_NO_VALUE` is returned.

2423 When using the non-opaque scalar variant (val) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
 2424 mode, the new contents of val are as defined above if the method exits with return value `GrB_SUCCESS`
 2425 or `GrB_NO_VALUE`.

2426 When using the GraphBLAS scalar variant (\mathbf{s}) with a `GrB_SUCCESS` return value, the method
 2427 exits and the new contents of \mathbf{s} is as defined above and fully computed in `GrB_BLOCKING` mode.
 2428 In `GrB_NONBLOCKING` mode, the new contents of \mathbf{s} is as defined above but may not be fully
 2429 computed; however, it can be used in the next GraphBLAS method call in a sequence.

2430 4.2.5.13 Matrix_extractTuples: Extract tuples from a matrix

2431 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

2432 C Syntax

```
2433      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2434                                     GrB_Index      *col_indices,
```

```

2435                                     <type>          *values,
2436                                     GrB_Index         *n,
2437                                     const GrB_Matrix   A);

```

2438 Parameters

2439 **row_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the
2440 row indices.

2441 **col_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the
2442 column indices.

2443 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
2444 the stored values whose type is compatible with $\mathbf{D}(\mathbf{A})$.

2445 **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,
2446 **row_indices**, and **col_indices** arrays can hold. Upon return, it will contain the
2447 number of values written to the arrays.

2448 **A** (IN) An existing GraphBLAS matrix.

2449 Return Values

2450 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2451 cessfully. This indicates that the compatibility tests on the input
2452 argument passed successfully, and the output arrays, **indices** and
2453 **values**, have been computed.

2454 **GrB_PANIC** Unknown internal error.

2455 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2456 GraphBLAS objects (input or output) is in an invalid state caused
2457 by a previous execution error. Call **GrB_error()** to access any error
2458 messages generated by the implementation.

2459 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2460 **GrB_INSUFFICIENT_SPACE** Not enough space in **row_indices**, **col_indices**, and **values** (as indi-
2461 cated by the **n** parameter) to hold all of the tuples that will be
2462 extracted.

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

2465 **GrB_NULL_POINTER** **row_indices**, **col_indices**, **values** or **n** pointer is NULL.

2466 **GrB_DOMAIN_MISMATCH** The domains of the **A** matrix and **values** array are incompatible
2467 with one another.

2468 Description

2469 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with
2470 those tuples are placed in the **values** array, the column indices are placed in the **col_indices** array,
2471 and the row indices are placed in the **row_indices** array. These output arrays are pre-allocated by
2472 the user before calling this function such that each output array has enough space to hold at least
2473 **GrB_Matrix_nvals(A)** elements.

2474 Upon return of this function, a pair of $\{\text{row_indices}[k], \text{col_indices}[k]\}$ are unique for every valid
2475 k , but they are not required to be sorted in any particular order. Each tuple (i, j, A_{ij}) in **A** is
2476 unzipped and copied into a distinct k th location in output vectors:

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

2477 where $0 \leq k < \text{GrB_Matrix_nvals}(v)$. No gaps in output vectors are allowed; that is, if **row_indices**[k],
2478 **col_indices**[k] and **values**[k] exist upon return, so does **row_indices**[j], **col_indices**[j] and **values**[j] for
2479 all j such that $0 \leq j < k$.

2480 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,
2481 then a **GrB_INSUFFICIENT_SPACE** error is returned since it is undefined which subset of values
2482 would be extracted.

2483 In both **GrB_BLOCKING** mode **GrB_NONBLOCKING** mode if the method exits with return value
2484 **GrB_SUCCESS**, the new contents of the arrays **row_indices**, **col_indices** and **values** are as defined
2485 above.

2486 4.2.5.14 Matrix_exportHint: Provide a hint as to which storage format might be most 2487 efficient for exporting a matrix

2488 C Syntax

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

2489 Parameters

2490 **hint** (OUT) Pointer to a value of type **GrB_Format**.

2491 **A** (IN) A GraphBLAS matrix object.

2492 Return Values

2493 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2494 cessfully and the value of **hint** has been set.

2495 **GrB_PANIC** Unknown internal error.

2496 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 2497 opaque GraphBLAS objects (input or output) is in an invalid
 2498 state caused by a previous execution error. Call GrB_error() to
 2499 access any error messages generated by the implementation.

2500 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2501 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
 2502 any matrix constructor.

2503 GrB_NULL_POINTER hint is NULL.

2504 GrB_NO_VALUE If the implementation does not have a preferred format, it may
 2505 return the value GrB_NO_VALUE.

2506 Description

2507 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for
 2508 exporting the matrix A. GraphBLAS implementations might return the current storage format of
 2509 the matrix, or the format to which it could most efficiently be exported. However, implementations
 2510 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is
 2511 free to refuse to provide a format hint, returning GrB_NO_VALUE.

2512 4.2.5.15 Matrix_exportSize: Return the array sizes necessary to export a GraphBLAS 2513 matrix object

2514 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);

```

2515 Parameters

2516 n_indptr (OUT) Pointer to a value of type GrB_Index.

2517 n_indices (OUT) Pointer to a value of type GrB_Index.

2518 n_values (OUT) Pointer to a value of type GrB_Index.

2519 format (IN) a value indicating the format in which the matrix will be exported, as defined
 2520 in Section 3.5.3.1.

2521 A (IN) A GraphBLAS matrix object.

2522 Return Values

2523 GrB_SUCCESS In blocking mode or non-blocking mode, the operation com-
2524 pleted successfully. This indicates that the API checks for the
2525 input arguments passed successfully, and the number of elements
2526 necessary for the export buffers have been written to `n_indptr`,
2527 `n_indices`, and `n_values`, respectively.

2528 GrB_PANIC Unknown internal error.

2529 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
2530 opaque GraphBLAS objects (input or output) is in an invalid
2531 state caused by a previous execution error. Call `GrB_error()` to
2532 access any error messages generated by the implementation.

2533 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2534 GrB_UNINITIALIZED_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to
2535 any matrix constructor.

2536 GrB_NULL_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

2537 Description

2538 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to
2539 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and
2540 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be
2541 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user
2542 as one of the values defined in Section 3.5.3.1.

2543 4.2.5.16 Matrix_export: Export a GraphBLAS matrix to a pre-defined format

2544 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);
```

2545 Parameters

2546 **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-
2547 dices, depending on the value of **format**. It must be large enough to hold at
2548 least **n_indptr** elements of type **GrB_Index**, where **n_indices** was returned from
2549 **GrB_Matrix_exportSize()** method.

2550 **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements
2551 in **values**, depending on the value of **format**. It must be large enough to hold at
2552 least **n_indices** elements of type **GrB_Index**, where **n_indices** was returned from
2553 **GrB_Matrix_exportSize()** method.

2554 **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-
2555 ment must match the type of the values stored in **A**. It must be large enough
2556 to hold at least **n_values** elements of that type, where **n_values** was returned from
2557 **GrB_Matrix_exportSize**.

2558 **n_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**
2559 array can hold. Upon return, it will contain the number of elements written to the
2560 array.

2561 **n_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**
2562 array can hold. Upon return, it will contain the number of elements written to the
2563 array.

2564 **n_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**
2565 array can hold. Upon return, it will contain the number of elements written to the
2566 array.

2567 **format** (IN) a value indicating the format in which the matrix will be exported, as defined
2568 in Section 3.5.3.1.

2569 **A** (IN) A GraphBLAS matrix object.

2570 Return Values

2571 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2572 cessfully. This indicates that the compatibility tests on the input
2573 argument passed successfully, and the output arrays, **indptr**, **in-**
2574 **dices** and **values**, have been computed.

2575 **GrB_PANIC** Unknown internal error.

2576 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2577 opaque GraphBLAS objects (input or output) is in an invalid
2578 state caused by a previous execution error. Call **GrB_error()** to
2579 access any error messages generated by the implementation.

2580 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2581 GrB_INSUFFICIENT_SPACE Not enough space in `indptr`, `indices`, and/or `values` (as indicated
2582 by the corresponding `n_*` parameter) to hold all of the corre-
2583 sponding elements that will be extracted.

2584 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
2585 any matrix constructor.

2586 GrB_NULL_POINTER `indptr`, `indices`, `values` `n_indptr`, `n_indices`, `n_values` pointer is
2587 NULL.

2588 GrB_DOMAIN_MISMATCH The domain of `A` does not match with the type of `values`.

2589 Description

2590 Given a matrix `A`, this method exports the contents of the matrix into one of the pre-defined
2591 `GrB_Format` formats from Section 3.5.3.1. The user-allocated arrays pointed to by `indptr`, `indices`,
2592 and `values` must be at least large enough to hold the corresponding number of elements returned
2593 by calling `GrB_Matrix_exportSize`. The value of `format` can be chosen arbitrarily, but a call to
2594 `GrB_Matrix_exportHint` may suggest a format that results in the most efficient export. Details
2595 of the contents of `indptr`, `indices`, and `values` corresponding to each supported format is given in
2596 Appendix B.

2597 4.2.5.17 Matrix_import: Import a matrix into a GraphBLAS object

2598 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);

```

2599 Parameters

2600 `A` (INOUT) On a successful return, contains a handle to the newly created Graph-
2601 BLAS matrix.

2602 `d` (IN) The type corresponding to the domain of the matrix being created. Can be
2603 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
2604 GraphBLAS type.

2605 **nrows** (IN) Integer value holding the number of rows in the matrix.

2606 **ncols** (IN) Integer value holding the number of columns in the matrix.

2607 **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the
2608 value of **format**.

2609 **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending
2610 on the value of **format**.

2611 **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2612 **n_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2613 **n_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2614 **n_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2615 **format** (IN) a value indicating the format of the matrix being imported, as defined in
2616 Section 3.5.3.1.

2617 **Return Values**

2618 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2619 blocking mode, this indicates that the API checks for the input
2620 arguments passed successfully and the input arrays have been
2621 consumed. Either way, output matrix **A** is ready to be used in
2622 the next method of the sequence.

2623 **GrB_PANIC** Unknown internal error.

2624 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2625 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
2626 (needed for user-defined types).

2627 **GrB_NULL_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2628 **GrB_INDEX_OUT_OF_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices
2629 in **A** and or the size of **values**, **n_values**, depending on the value
2630 of **format**.

2631 **GrB_INVALID_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB_Index**.

2632 **GrB_DOMAIN_MISMATCH** The domain given in parameter **d** does not match the element
2633 type of **values**.

2634 Description

2635 Creates a new matrix **A** of domain **D**(d) and dimension **nrows** \times **ncols**. The new GraphBLAS
2636 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.
2637 The method returns a handle to the new matrix in **A**. The structure of the data being imported is
2638 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of
2639 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2640 It is not an error to call this method more than once on the same output matrix; however, the
2641 handle to the previously created object will be overwritten.

2642 4.2.5.18 Matrix_serializeSize: Compute the serialize buffer size

2643 Compute the buffer size (in bytes) necessary to serialize a GrB_Matrix using GrB_Matrix_serialize.

2644 C Syntax

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

2645 Parameters

2646 **size** (OUT) Pointer to GrB_Index value where size in bytes of serialized object will be
2647 written.

2648 **A** (IN) A GraphBLAS matrix object.

2649 Return Values

2650 **GrB_SUCCESS** The operation completed successfully and the value pointed to
2651 by ***size** has been computed and is ready to use.

2652 **GrB_PANIC** Unknown internal error.

2653 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2654 **GrB_NULL_POINTER** **size** is NULL.

2655 Description

2656 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object **A**.
2657 Users may then allocate a buffer of **size** bytes to pass as a parameter to GrB_Matrix_serialize.

2658 **4.2.5.19 Matrix_serialize: Serialize a GraphBLAS matrix.**

2659 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2660 **C Syntax**

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

2661 **Parameters**

2662 **serialized_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be
2663 written.

2664 **serialized_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized_data**.
2665 On output, the number of bytes written to **serialized_data**.

2666 **A** (IN) A GraphBLAS matrix object.

2667 **Return Values**

2668 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2669 cessfully. This indicates that the compatibility tests on the in-
2670 put argument passed successfully, and the output buffer **serial-
2671 ized_data** and **serialized_size**, have been computed and are ready
2672 to use.

2673 **GrB_PANIC** Unknown internal error.

2674 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2675 opaque GraphBLAS objects (input or output) is in an invalid
2676 state caused by a previous execution error. Call **GrB_error()** to
2677 access any error messages generated by the implementation.

2678 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2679 **GrB_NULL_POINTER** **serialized_data** or **serialize_size** is NULL.

2680 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2681 any matrix constructor.

2682 **GrB_INSUFFICIENT_SPACE** The size of the buffer **serialized_data** (provided as an input **seri-
2683 alized_size**) was not large enough.

2684 Description

2685 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,
2686 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must
2687 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the
2688 serialized matrix written to `serialized_data` is provided upon completion as an output written to
2689 `serialized_size`.

2690 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created
2691 with one library implementation is not necessarily valid for deserialization with another implemen-
2692 tation.

2693 4.2.5.20 Matrix_deserialize: Deserialize a GraphBLAS matrix.

2694 Construct a new GraphBLAS matrix from a serialized object.

2695 C Syntax

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

2696 Parameters

2697 A (INOUT) On a successful return, contains a handle to the newly created Graph-
2698 BLAS matrix.

2699 d (IN) the type of the matrix that was serialized in `serialized_data`.

2700 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2701 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

2702 Return Values

2703 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2704 blocking mode, this indicates that the API checks for the input
2705 arguments passed successfully. Either way, output matrix A is
2706 ready to be used in the next method of the sequence.

2707 GrB_PANIC Unknown internal error.

2708 GrB_INVALID_OBJECT This is returned if `serialized_data` is invalid or corrupted.

2709 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2710 GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to GrB_Type_new
2711 (needed for user-defined types).

2712 GrB_NULL_POINTER serialized_data or A is NULL.

2713 GrB_DOMAIN_MISMATCH The type given in d does not match the type of the matrix
2714 serialized in serialized_data.

2715 Description

2716 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object
2717 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The
2718 domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized
2719 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2720 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix
2721 serialized in one library implementation can be deserialized by another.

2722 It is not an error to call this method more than once on the same output matrix; however, the
2723 handle to the previously created object will be overwritten.

2724 4.2.6 Descriptor methods

2725 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-
2726 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

2727 4.2.6.1 Descriptor_new: Create new descriptor

2728 Creates a new (empty or default) descriptor.

2729 C Syntax

2730 GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);

2731 Parameters

2732 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2733 descriptor.

2734 Return Value

2735 GrB_SUCCESS The method completed successfully.

2736 GrB_PANIC unknown internal error.

2737 GrB_OUT_OF_MEMORY not enough memory available for operation.

2738 GrB_NULL_POINTER desc pointer is NULL.

2739 **Description**

2740 Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can
2741 be populated by calls to Descriptor_set.

2742 It is not an error to call this method more than once on the same variable; however, the handle to
2743 the previously created object will be overwritten.

2744 **4.2.6.2 Descriptor_set: Set content of descriptor**

2745 Sets the content for a field for an existing descriptor.

2746 **C Syntax**

```
2747        GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2748                                    GrB_Desc_Field        field,  
2749                                    GrB_Desc_Value        val);
```

2750 **Parameters**

2751 desc (IN) An existing GraphBLAS descriptor to be modified.

2752 field (IN) The field being set.

2753 val (IN) New value for the field being set.

2754 **Return Values**

2755 GrB_SUCCESS operation completed successfully.

2756 GrB_PANIC unknown internal error.

2757 GrB_OUT_OF_MEMORY not enough memory available for operation.

2758 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to new.

2759 GrB_INVALID_VALUE invalid value set on the field, or invalid field.

2760 Description

2761 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor
2762 to set the value associated with that field. Valid values for the `field` parameter include the following:

2763 `GrB_OUTP` refers to the output parameter (result) of the operation.

2764 `GrB_MASK` refers to the mask parameter of the operation.

2765 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2766 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2767 Valid values for the `val` parameter are:

2768 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask
2769 (`GrB_MASK`) parameter.

2770 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-
2771 eter. When combined with `GrB_STRUCTURE`, the complement of the
2772 structure of the mask is used without evaluating the values stored.

2773 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input
2774 matrix parameters only).

2775 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear
2776 the matrix first (or clear the non-masked entries). The default behavior
2777 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`
2778 parameter only.

2779 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,
2780 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).
2781 A value for a given field may be set multiple times but will have no additional effect. Fields that
2782 have no values set result in their default behavior, as defined in Section 3.6.

2783 4.2.7 free: Destroy an object and release its resources

2784 Destroys a previously created GraphBLAS object and releases any resources associated with the
2785 object.

2786 C Syntax

2787 `GrB_Info GrB_free(<GrB_Object> *obj);`

2788 Parameters

2789 **obj** (INOUT) An existing GraphBLAS object to be destroyed. The object must have
2790 been created by an explicit call to a GraphBLAS constructor. It can be any of the
2791 opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,
2792 binary op, unary op, or type. On successful completion of **GrB_free**, **obj** behaves
2793 as an uninitialized object.

2794 Return Values

2795 **GrB_SUCCESS** operation completed successfully

2796 **GrB_PANIC** unknown internal error. If this return value is encountered when
2797 in nonblocking mode, the error responsible for the panic condition
2798 could be from any method involved in the computation of the input
2799 object. The **GrB_error()** method should be called for additional
2800 information.

2801 Description

2802 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime
2803 system. A call to **GrB_free** frees those resources so they are available for use by other GraphBLAS
2804 objects.

2805 The parameter passed into **GrB_free** is a handle referencing a GraphBLAS opaque object of a data
2806 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-
2807 structor. The behavior of a program that calls **GrB_free** on a pre-defined object is implementation
2808 defined.

2809 After the **GrB_free** method returns, the object referenced by the input handle is destroyed and the
2810 handle has the value **GrB_INVALID_HANDLE**. The handle can be used in subsequent GraphBLAS
2811 methods but only after the handle has been reinitialized with a call the the appropriate **_new** or
2812 **_dup** method.

2813 Note that unlike other GraphBLAS methods, calling **GrB_free** with an object with an invalid handle
2814 is legal. The system may attempt to free resources that might be associated with that object, if
2815 possible, and return normally.

2816 When using **GrB_free** it is possible to create a dangling reference to an object. This would occur
2817 when a handle is assigned to a second variable of the same opaque type. This creates two handles
2818 that reference the same object. If **GrB_free** is called with one of the variables, the object is destroyed
2819 and the handle associated with the other variable no longer references a valid object. This is not an
2820 error condition that the implementation of the GraphBLAS API can be expected to catch, hence
2821 programmers must take care to prevent this situation from occurring.

2822 **4.2.8 wait: Return once an object is either *complete* or *materialized***

2823 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2824 **C Syntax**

2825 `GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);`

2826 **Parameters**

2827 **obj** (INOUT) An existing GraphBLAS object. The object must have been created by an
2828 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2829 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2830 or type. On successful return of `GrB_wait`, the **obj** can be safely read from another
2831 thread (completion) or all computing to produce **obj** by all GraphBLAS operations
2832 in its sequence have finished (materialization).

2833 **mode** (IN) Set's the mode for `GrB_wait` for whether it is waiting for **obj** to be in the
2834 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or
2835 `GrB_MATERIALIZE`.

2836 **Return values**

2837 `GrB_SUCCESS` operation completed successfully.

2838 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-
2839 pletion of pending operations.

2840 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion
2841 of pending operations.

2842 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,
2843 or other constructor, method.

2844 `GrB_PANIC` unknown internal error.

2845 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`
2846 `GrB_MATERIALIZE`.

2847 **Description**

2848 On successful return from `GrB_wait()`, the input object, **obj** is in one of two states depending on
2849 the mode of `GrB_wait`:

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

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

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

4.2.9 error: Retrieve an error string

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

C Syntax

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

Parameters

`error` (OUT) A pointer to a null-terminated string. The contents of the string are implementation defined.

`obj` (IN) An existing GraphBLAS object. The object must have been created by an explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op, or type.

Return value

`GrB_SUCCESS` operation completed successfully.

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

`GrB_PANIC` unknown internal error.

Description

This method retrieves a message related to any errors that were encountered during the last GraphBLAS method that had the opaque GraphBLAS object, `obj`, as an OUT or INOUT parameter. The function returns a pointer to a null-terminated string and the contents of that string are implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error string. The string that is returned is owned by `obj` and will be valid until the next time `obj` is used as an OUT or INOUT parameter or the object is freed by a call to `GrB_free(obj)`. This is a thread-safe function. It can be safely called by multiple threads for the same object in a race-free program.

4.3 GraphBLAS operations

The GraphBLAS operations are defined in the GraphBLAS math specification and summarized in Table 4.1. In addition to methods that implement these fundamental GraphBLAS operations, we support a number of variants that have been found to be especially useful in algorithm development. A flowchart of the overall behavior of a GraphBLAS operation is shown in Figure 4.1.

Domains and Casting

A GraphBLAS operation is only valid when the domains of the GraphBLAS objects are mathematically consistent. The C programming language defines implicit casts between built-in data types. For example, floats, doubles, and ints can be freely mixed according to the rules defined for implicit casts. It is the responsibility of the user to assure that these casts are appropriate for the algorithm in question. For example, a cast to int implies truncation of a floating point type. Depending on the operation, this truncation error could lead to erroneous results. Furthermore, casting a wider type onto a narrower type can lead to overflow errors. The GraphBLAS operations do not attempt to protect a user from these sorts of errors.

When user-defined types are involved, however, GraphBLAS requires strict equivalence between types and no casting is supported. If GraphBLAS detects these mismatches, it will return a domain mismatch error.

Dimensions and Transposes

GraphBLAS operations also make assumptions about the numbers of dimensions and the sizes of vectors and matrices in an operation. An operation will test these sizes and report an error if they are not *shape compatible*. For example, when multiplying two matrices, $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, the number of rows of \mathbf{C} must equal the number of rows of \mathbf{A} , the number of columns of \mathbf{A} must match the number of rows of \mathbf{B} , and the number of columns of \mathbf{C} must match the number of columns of \mathbf{B} . This is the behavior expected given the mathematical definition of the operations.

For most of the GraphBLAS operations involving matrices, an optional descriptor can modify the matrix associated with an input GraphBLAS matrix object. For example, if an input matrix is an

Table 4.1: A mathematical notation for the fundamental GraphBLAS operations supported in this specification. Input matrices \mathbf{A} and \mathbf{B} may be optionally transposed (not shown). Use of an optional accumulate with existing values in the output object is indicated with \odot . Use of optional write masks and replace flags are indicated as $\mathbf{C}\langle\mathbf{M}, r\rangle$ when applied to the output matrix, \mathbf{C} . The mask controls which values resulting from the operation on the right-hand side are written into the output object (complement and structure flags are not shown). The “replace” option, indicated by specifying the r flag, means that all values in the output object are removed prior to assignment. If “replace” is not specified, only the values/locations computed on the right-hand side and allowed by the mask will be written to the output (“merge” mode).

Operation Name	Mathematical Notation		
mxm	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot \mathbf{A} \oplus . \otimes \mathbf{u}$
vxm	$\mathbf{w}^T\langle\mathbf{m}^T, r\rangle$	=	$\mathbf{w}^T \odot \mathbf{u}^T \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot \mathbf{u} \otimes \mathbf{v}$
eWiseAdd	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot \mathbf{u} \oplus \mathbf{v}$
extract	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A}(i, j)$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot \mathbf{u}(i)$
assign	$\mathbf{C}\langle\mathbf{M}, r\rangle(i, j)$	=	$\mathbf{C}(i, j) \odot \mathbf{A}$
	$\mathbf{w}\langle\mathbf{m}, r\rangle(i)$	=	$\mathbf{w}(i) \odot \mathbf{u}$
reduce (row)	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$
reduce (scalar)	s	=	$s \odot [\oplus_{i,j} \mathbf{A}(i, j)]$
	s	=	$s \odot [\oplus_i \mathbf{u}(i)]$
apply	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot f_u(\mathbf{A})$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot f_u(\mathbf{u})$
apply(indexop)	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot f_i(\mathbf{A}, \text{ind}(\mathbf{A}), s)$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot f_i(\mathbf{u}, \text{ind}(\mathbf{u}), s)$
select	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A}\langle f_i(\mathbf{A}, \text{ind}(\mathbf{A}), s) \rangle$
	$\mathbf{w}\langle\mathbf{m}, r\rangle$	=	$\mathbf{w} \odot \mathbf{u}\langle f_i(\mathbf{u}, \text{ind}(\mathbf{u}), s) \rangle$
transpose	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A}^T$
kronecker	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

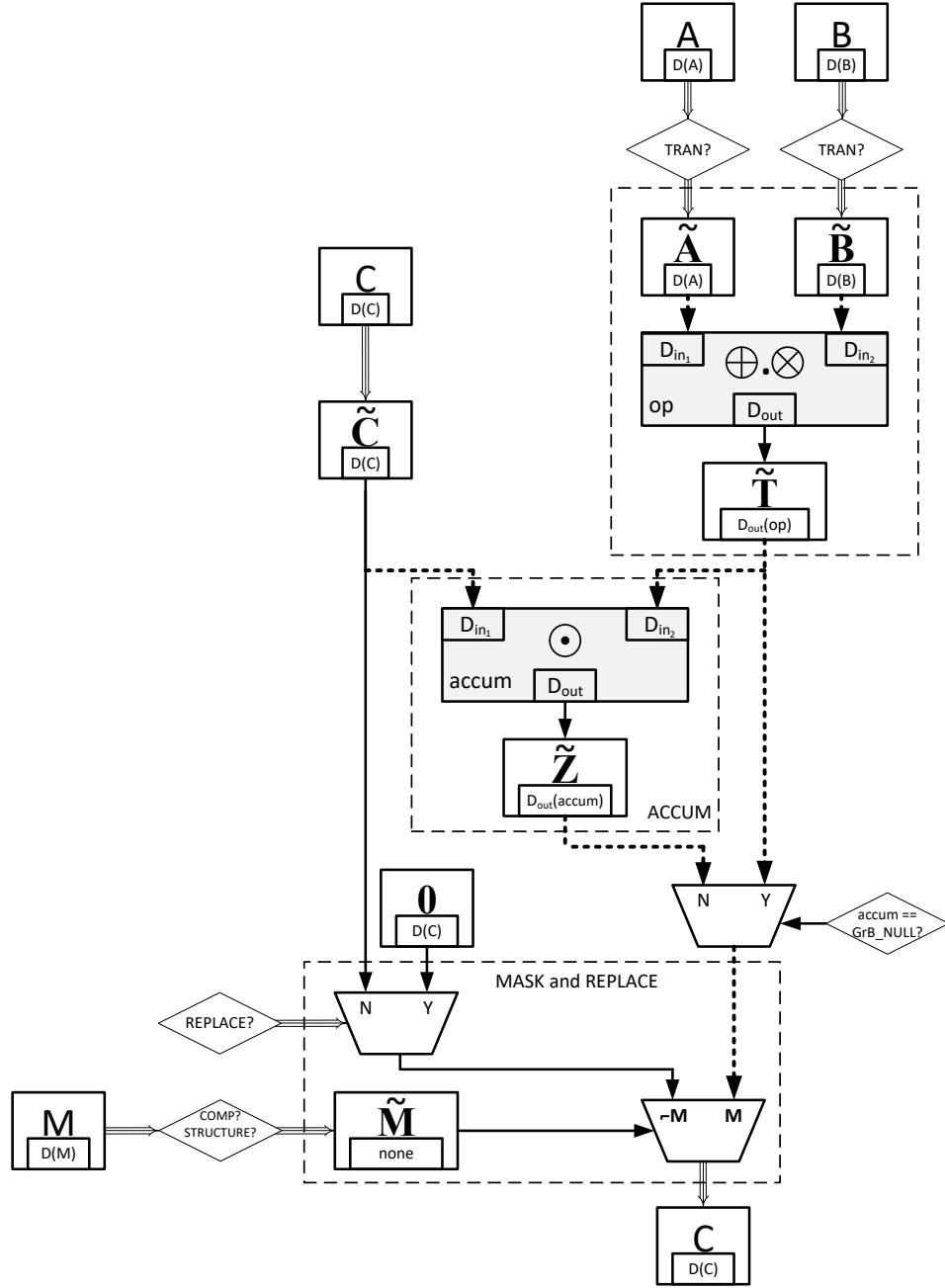


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows (\Rightarrow) denote where “as if copy” takes place (including both collections and descriptor settings). The bold, dotted arrows indicate where casting may occur between different domains.

argument to a GraphBLAS operation and the associated descriptor indicates the transpose option, then the operation occurs as if on the transposed matrix. In this case, the relationships between the sizes in each dimension shift in the mathematically expected way.

Masks: Structure-only, Complement, and Replace

When a GraphBLAS operation supports the use of an optional mask, that mask is specified through a GraphBLAS vector (for one-dimensional masks) or a GraphBLAS matrix (for two-dimensional masks). When a mask is used and the `GrB_STRUCTURE` descriptor value is not set, it is applied to the result from the operation wherever the stored values in the mask evaluate to true. If the `GrB_STRUCTURE` descriptor is set, the mask is applied to the result from the operation wherever the mask as a stored value (regardless of that value). Wherever the mask is applied, the result from the operation is either assigned to the provided output matrix/vector or, if a binary accumulation operation is provided, the result is accumulated into the corresponding elements of the provided output matrix/vector.

Given a GraphBLAS vector $\mathbf{v} = \langle D, N, \{(i, v_i)\} \rangle$, a one-dimensional mask is derived for use in the operation as follows:

$$\mathbf{m} = \begin{cases} \langle N, \{\mathbf{ind}(\mathbf{v})\} \rangle, & \text{if } \text{GrB_STRUCTURE} \text{ is specified,} \\ \langle N, \{i : (\text{bool})v_i = \text{true}\} \rangle, & \text{otherwise} \end{cases}$$

where $(\text{bool})v_i$ denotes casting the value v_i to a Boolean value (true or false). Likewise, given a GraphBLAS matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$, a two-dimensional mask is derived for use in the operation as follows:

$$\mathbf{M} = \begin{cases} \langle M, N, \{\mathbf{ind}(\mathbf{A})\} \rangle, & \text{if } \text{GrB_STRUCTURE} \text{ is specified,} \\ \langle M, N, \{(i, j) : (\text{bool})A_{ij} = \text{true}\} \rangle, & \text{otherwise} \end{cases}$$

where $(\text{bool})A_{ij}$ denotes casting the value A_{ij} to a Boolean value. (true or false)

In both the one- and two-dimensional cases, the mask may also have a subsequent complement operation applied (*Section 3.5.4*) as specified in the descriptor, before a final mask is generated for use in the operation.

When the descriptor of an operation with a mask has specified that the `GrB_REPLACE` value is to be applied to the output (`GrB_OUTP`), then anywhere the mask is not true, the corresponding location in the output is cleared.

Invalid and uninitialized objects

Upon entering a GraphBLAS operation, the first step is a check that all objects are valid and initialized. (Optional parameters can be set to `GrB_NULL`, which always counts as a valid object.) An invalid object is one that could not be computed due to a previous execution error. An uninitialized object is one that has not yet been created by a corresponding `new` or `dup` method. Appropriate error codes are returned if an object is not initialized (`GrB_UNINITIALIZED_OBJECT`) or invalid (`GrB_INVALID_OBJECT`).

2950 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-
 2951 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at
 2952 the point of their declaration, as shown in the following examples:

```
2953         GrB_Type          type = GrB_INVALID_HANDLE;
2954         GrB_Semiring      semiring = GrB_INVALID_HANDLE;
2955         GrB_Matrix        matrix = GrB_INVALID_HANDLE;
```

2956 Compliance

2957 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.
 2958 That is, for each operation we give a recipe for producing its outcome. Any implementation that
 2959 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error
 2960 model (Section 2.6) is a conforming implementation.

2961 4.3.1 mxm: Matrix-matrix multiply

2962 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

2963 C Syntax

```
2964         GrB_Info GrB_mxm(GrB_Matrix          C,
2965                         const GrB_Matrix      Mask,
2966                         const GrB_BinaryOp    accum,
2967                         const GrB_Semiring     op,
2968                         const GrB_Matrix      A,
2969                         const GrB_Matrix      B,
2970                         const GrB_Descriptor   desc);
```

2971 Parameters

2972 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 2973 that may be accumulated with the result of the matrix product. On output, the
 2974 matrix holds the results of the operation.

2975 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 2976 stored into the output matrix C. The mask dimensions must match those of the
 2977 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
 2978 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
 2979 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
 2980 dimensions of C), `GrB_NULL` should be specified.

2981 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
 2982 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 2983 specified.

2984 **op** (IN) The semiring used in the matrix-matrix multiply.

2985 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 2986 multiplication.

2987 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 2988 multiplication.

2989 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 2990 should be specified. Non-default field/value pairs are listed as follows:
 2991

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.

2993 Return Values

2994 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 2995 blocking mode, this indicates that the compatibility tests on di-
 2996 mensions and domains for the input arguments passed successfully.
 2997 Either way, output matrix **C** is ready to be used in the next method
 2998 of the sequence.

2999 **GrB_PANIC** Unknown internal error.

3000 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 3001 GraphBLAS objects (input or output) is in an invalid state caused
 3002 by a previous execution error. Call **GrB_error()** to access any error
 3003 messages generated by the implementation.

3004 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

3005 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 3006 a call to **new** (or **Matrix_dup** for matrix parameters).

3007 **GrB_DIMENSION_MISMATCH** Mask and/or matrix dimensions are incompatible.

3008 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3009 corresponding domains of the semiring or accumulation operator,
 3010 or the mask's domain is not compatible with `bool` (in the case where
 3011 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3012 Description

3013 GrB_mxm computes the matrix product $C = A \oplus . \otimes B$ or, if an optional binary accumulation operator
 3014 (\odot) is provided, $C = C \odot (A \oplus . \otimes B)$ (where matrices A and B can be optionally transposed).
 3015 Logically, this operation occurs in three steps:

3016 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3017 and dimensions are tested for compatibility.

3018 **Compute** The indicated computations are carried out.

3019 **Output** The result is written into the output matrix, possibly under control of a mask.

3020 Up to four argument matrices are used in the GrB_mxm operation:

- 3021 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3022 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)
- 3023 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3024 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3025 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for
 3026 domain compatibility as follows:

- 3027 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3028 must be from one of the pre-defined types of Table 3.2.
- 3029 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 3030 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 3031 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 3032 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3033 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 3034 of the accumulation operator.

3035 Two domains are compatible with each other if values from one domain can be cast to values in
 3036 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3037 all compatible with each other. A domain from a user-defined type is only compatible with itself.

3038 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch
 3039 error listed above is returned.

3040 From the argument matrices, the internal matrices and mask used in the computation are formed
 3041 (\leftarrow denotes copy):

- 3042 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3043 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3044 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3045 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3046 (b) If `Mask \neq GrB_NULL`,
 - 3047 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3048 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3049 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3050 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 3051 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3052 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3053 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3054 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3055 must hold:

- 3056 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 3057 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 3058 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 3059 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.
- 3060 5. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.

3061 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch
 3062 error listed above is returned.

3063 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3064 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3065 We are now ready to carry out the matrix multiplication and any additional associated operations.
 3066 We describe this in terms of two intermediate matrices:

- 3067 • $\tilde{\mathbf{T}}$: The matrix holding the product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3068 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3069 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

$$3071 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3072 where \oplus and \otimes are the additive and multiplicative operators of semiring \mathbf{op} , respectively.

3073 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3074 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3075 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3076 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3077 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
3078 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 3079 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3080 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3081 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3082 \end{aligned}$$

3083 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3085 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
3086 using what is called a *standard matrix mask and replace*. This is carried out under control of the
3087 mask which acts as a “write mask”.

- 3088 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
3089 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3090 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3091 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
3092 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
3093 mask are unchanged:

$$3094 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3095 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
3096 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
3097 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
3098 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3099 sequence.

3100 4.3.2 vxm: Vector-matrix multiply

3101 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

3102 C Syntax

```
3103         GrB_Info GrB_vxm(GrB_Vector          w,  
3104                           const GrB_Vector    mask,  
3105                           const GrB_BinaryOp   accum,  
3106                           const GrB_Semiring   op,  
3107                           const GrB_Vector    u,  
3108                           const GrB_Matrix    A,  
3109                           const GrB_Descriptor desc);
```

3110 Parameters

3111 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3112 that may be accumulated with the result of the vector-matrix product. On output,
3113 this vector holds the results of the operation.

3114 **mask** (IN) An optional “write” mask that controls which results from this operation are
3115 stored into the output vector **w**. The mask dimensions must match those of the
3116 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3117 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3118 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3119 dimensions of **w**), **GrB_NULL** should be specified.

3120 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3121 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
3122 specified.

3123 **op** (IN) Semiring used in the vector-matrix multiply.

3124 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3125 multiplication.

3126 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
3127 multiplication.

3128 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
3129 should be specified. Non-default field/value pairs are listed as follows:

3130

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.

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 the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to `new` (or `dup` for matrix or vector parameters).

GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with the corresponding domains of the semiring 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).

Description

GrB_vxm computes the vector-matrix product $w^T = u^T \oplus . \otimes A$, or, if an optional binary accumulation operator (\odot) is provided, $w^T = w^T \odot (u^T \oplus . \otimes A)$ (where matrix A can be optionally transposed). Logically, this operation occurs in three steps:

Setup The internal vectors, matrices and mask used in the computation are formed and their domains/dimensions are tested for compatibility.

Compute The indicated computations are carried out.

3158 **Output** The result is written into the output vector, possibly under control of a mask.

3159 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3160 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3161 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3162 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3163 4. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3164 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
 3165 tested for domain compatibility as follows:

- 3166 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 3167 must be from one of the pre-defined types of Table 3.2.
- 3168 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 3169 3. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 3170 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.
- 3171 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3172 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$
 3173 of the accumulation operator.

3174 Two domains are compatible with each other if values from one domain can be cast to values in
 3175 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3176 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 3177 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch
 3178 error listed above is returned.

3179 From the argument vectors and matrices, the internal matrices and mask used in the computation
 3180 are formed (\leftarrow denotes copy):

- 3181 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3182 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3183 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3184 (b) If `mask` \neq `GrB_NULL`,
 - 3185 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3186 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$.
 - 3187 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3188 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3189 4. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

3190 The internal matrices and masks are checked for shape compatibility. The following conditions
3191 must hold:

3192 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.

3193 2. $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$.

3194 3. $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$.

3195 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch
3196 error listed above is returned.

3197 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3198 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3199 We are now ready to carry out the vector-matrix multiplication and any additional associated
3200 operations. We describe this in terms of two intermediate vectors:

- 3201 • $\tilde{\mathbf{t}}$: The vector holding the product of vector $\tilde{\mathbf{u}}^T$ and matrix $\tilde{\mathbf{A}}$.
- 3202 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3203 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{ncols}(\tilde{\mathbf{A}}), \{(j, t_j) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j)) \neq \emptyset\} \rangle$ is created.
3204 The value of each of its elements is computed by

$$3205 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3206 where \oplus and \otimes are the additive and multiplicative operators of semiring `op`, respectively.

3207 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3208 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3209 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3210 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3211 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3212 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3213 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3214 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3215 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3216 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3217 \end{aligned}$$

3218 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3219 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3220 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3221 mask which acts as a “write mask”.

- 3222 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 3223 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3224 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3225 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3226 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3227 mask are unchanged:

$$3228 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3229 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 3230 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 3231 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 3232 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3233 sequence.

3234 4.3.3 mxv: Matrix-vector multiply

3235 Multiplies a matrix by a vector on a semiring. The result is a vector.

3236 C Syntax

```
3237 GrB_Info GrB_mxv(GrB_Vector      w,
3238                  const GrB_Vector mask,
3239                  const GrB_BinaryOp accum,
3240                  const GrB_Semiring op,
3241                  const GrB_Matrix A,
3242                  const GrB_Vector u,
3243                  const GrB_Descriptor desc);
```

3244 Parameters

3245 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 3246 that may be accumulated with the result of the matrix-vector product. On output,
 3247 this vector holds the results of the operation.

3248 **mask** (IN) An optional “write” mask that controls which results from this operation are
 3249 stored into the output vector \mathbf{w} . The mask dimensions must match those of the
 3250 vector \mathbf{w} . If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain

3251 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types
 3252 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
 3253 dimensions of `w`), `GrB_NULL` should be specified.

3254 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`
 3255 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 3256 specified.

3257 `op` (IN) Semiring used in the vector-matrix multiply.

3258 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3259 multiplication.

3260 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3261 multiplication.

3262 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 3263 should be specified. Non-default field/value pairs are listed as follows:
 3264

Param	Field	Value	Description
<code>w</code>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <code>w</code> is cleared (all elements removed) before the result is stored in it.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <code>mask</code> .
<code>A</code>	<code>GrB_INP0</code>	<code>GrB_TRAN</code>	Use transpose of <code>A</code> for the operation.

3266 Return Values

3267 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 3268 blocking mode, this indicates that the compatibility tests on di-
 3269 mensions and domains for the input arguments passed successfully.
 3270 Either way, output vector `w` is ready to be used in the next method
 3271 of the sequence.

3272 `GrB_PANIC` Unknown internal error.

3273 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 3274 GraphBLAS objects (input or output) is in an invalid state caused
 3275 by a previous execution error. Call `GrB_error()` to access any error
 3276 messages generated by the implementation.

3277 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3278 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by
 3279 a call to `new` (or `dup` for matrix or vector parameters).

3280 GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3281 GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with
3282 the corresponding domains of the semiring or accumulation opera-
3283 tor, or the mask's domain is not compatible with **bool** (in the case
3284 where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3285 Description

3286 GrB_mvx computes the matrix-vector product $w = A \oplus . \otimes u$, or, if an optional binary accumulation
3287 operator (\odot) is provided, $w = w \odot (A \oplus . \otimes u)$ (where matrix A can be optionally transposed).
3288 Logically, this operation occurs in three steps:

3289 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
3290 domains/dimensions are tested for compatibility.

3291 **Compute** The indicated computations are carried out.

3292 **Output** The result is written into the output vector, possibly under control of a mask.

3293 Up to four argument vectors or matrices are used in the GrB_mvx operation:

- 3294 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
3295 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
3296 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
3297 4. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3298 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
3299 tested for domain compatibility as follows:

- 3300 1. If **mask** is not GrB_NULL, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3301 must be from one of the pre-defined types of Table 3.2.
3302 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
3303 3. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
3304 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
3305 5. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3306 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
3307 of the accumulation operator.

3308 Two domains are compatible with each other if values from one domain can be cast to values in
 3309 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3310 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 3311 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the domain mismatch
 3312 error listed above is returned.

3313 From the argument vectors and matrices, the internal matrices and mask used in the computation
 3314 are formed (\leftarrow denotes copy):

- 3315 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3316 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3317 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3318 (b) If `mask \neq GrB_NULL`,
 - 3319 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3320 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3321 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3322 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3323 4. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3324 The internal matrices and masks are checked for shape compatibility. The following conditions
 3325 must hold:

- 3326 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.
- 3327 2. $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}(\tilde{\mathbf{A}})$.
- 3328 3. $\text{size}(\tilde{\mathbf{u}}) = \text{ncols}(\tilde{\mathbf{A}})$.

3329 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the dimension mismatch
 3330 error listed above is returned.

3331 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3332 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3333 We are now ready to carry out the matrix-vector multiplication and any additional associated
 3334 operations. We describe this in terms of two intermediate vectors:

- 3335 • $\tilde{\mathbf{t}}$: The vector holding the product of matrix $\tilde{\mathbf{A}}$ and vector $\tilde{\mathbf{u}}$.
- 3336 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3337 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{nrows}(\tilde{\mathbf{A}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}}) \neq \emptyset\} \rangle$ is created.
 3338 The value of each of its elements is computed by

$$3339 \quad t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3340 where \oplus and \otimes are the additive and multiplicative operators of semiring **op**, respectively.

3341 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3342 • If **accum** = **GrB_NULL**, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3343 • If **accum** is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3344 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3345 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3346 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3347 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3348 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3349 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3350 \end{aligned}$$

3351 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3352 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector **w**,
 3353 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3354 mask which acts as a “write mask”.

- 3355 • If **desc[GrB_OUTP].GrB_REPLACE** is set, then any values in **w** on input to this operation are
 3356 deleted and the content of the new output vector, **w**, is defined as,

$$3357 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3358 • If **desc[GrB_OUTP].GrB_REPLACE** is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3359 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the
 3360 mask are unchanged:

$$3361 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3362 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new content
 3363 of vector **w** is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method
 3364 exits with return value **GrB_SUCCESS** and the new content of vector **w** is as defined above but
 3365 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3366 sequence.

3367 4.3.4 eWiseMult: Element-wise multiplication

3368 **Note:** The difference between **eWiseAdd** and **eWiseMult** is not about the element-wise operation
 3369 but how the index sets are treated. **eWiseAdd** returns an object whose indices are the “union” of
 3370 the indices of the inputs whereas **eWiseMult** returns an object whose indices are the “intersection”
 3371 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
 3372 the set of values from the resulting index set.

3374 4.3.4.1 eWiseMult: Vector variant

3375 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-
3376 ducing a third vector as result.

3377 C Syntax

```
3378     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3379                           const GrB_Vector mask,  
3380                           const GrB_BinaryOp accum,  
3381                           const GrB_Semiring op,  
3382                           const GrB_Vector u,  
3383                           const GrB_Vector v,  
3384                           const GrB_Descriptor desc);  
3385  
3386     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3387                           const GrB_Vector mask,  
3388                           const GrB_BinaryOp accum,  
3389                           const GrB_Monoid op,  
3390                           const GrB_Vector u,  
3391                           const GrB_Vector v,  
3392                           const GrB_Descriptor desc);  
3393  
3394     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3395                           const GrB_Vector mask,  
3396                           const GrB_BinaryOp accum,  
3397                           const GrB_BinaryOp op,  
3398                           const GrB_Vector u,  
3399                           const GrB_Vector v,  
3400                           const GrB_Descriptor desc);
```

3401 Parameters

3402 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3403 that may be accumulated with the result of the element-wise operation. On output,
3404 this vector holds the results of the operation.

3405 **mask** (IN) An optional “write” mask that controls which results from this operation are
3406 stored into the output vector **w**. The mask dimensions must match those of the
3407 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3408 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3409 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3410 dimensions of **w**), **GrB_NULL** should be specified.

3411 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3412 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3413 specified.

3414 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3415 operation. Depending on which type is passed, the following defines the binary
3416 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3417 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3418 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-
3419 nored.

3420 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
3421 is ignored.

3422 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3423 operation.

3424 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
3425 operation.

3426 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
3427 should be specified. Non-default field/value pairs are listed as follows:
3428

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.

3430 Return Values

3431 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3432 blocking mode, this indicates that the compatibility tests on di-
3433 mensions and domains for the input arguments passed successfully.
3434 Either way, output vector w is ready to be used in the next method
3435 of the sequence.

3436 GrB_PANIC Unknown internal error.

3437 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3438 GraphBLAS objects (input or output) is in an invalid state caused
3439 by a previous execution error. Call GrB_error() to access any error
3440 messages generated by the implementation.

3441 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3442 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3443 a call to `new` (or `dup` for vector parameters).

3444 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3445 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 3446 responding domains of the binary operator (`op`) or accumulation
 3447 operator, or the mask's domain is not compatible with `bool` (in the
 3448 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3449 Description

3450 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:
 3451 $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$.
 3452 Logically, this operation occurs in three steps:

3453 **Setup** The internal vectors and mask used in the computation are formed and their domains
 3454 and dimensions are tested for compatibility.

3455 **Compute** The indicated computations are carried out.

3456 **Output** The result is written into the output vector, possibly under control of a mask.

3457 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3458 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3459 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3460 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3461 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3462 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)
 3463 are tested for domain compatibility as follows:

- 3464 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 3465 must be from one of the pre-defined types of Table 3.2.
- 3466 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 3467 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 3468 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3469 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3470 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
 3471 the accumulation operator.

3472 Two domains are compatible with each other if values from one domain can be cast to values in
 3473 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3474 compatible with each other. A domain from a user-defined type is only compatible with itself. If any
 3475 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3476 error listed above is returned.

3477 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3478 denotes copy):

- 3479 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3480 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3481 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3482 (b) If `mask \neq GrB_NULL`,
 - 3483 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3484 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3485 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3486 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3487 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3488 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3489 must hold:

- 3490 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$.

3491 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3492 mismatch error listed above is returned.

3493 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3494 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3495 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3496 We describe this in terms of two intermediate vectors:

- 3497 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “product” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3498 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3499 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$ is created. The
 3500 value of each of its elements is computed by:

$$3501 \quad t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))$$

3502 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

3503 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3504 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

3505
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3506 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3507 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

3508
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3509

3510
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3511

3512
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3513 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3514 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3515 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3516 mask which acts as a “write mask”.

3517 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in \mathbf{w} on input to this operation are
 3518 deleted and the content of the new output vector, \mathbf{w} , is defined as,

3519
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3520 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3521 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3522 mask are unchanged:

3523
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3524 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new content
 3525 of vector \mathbf{w} is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method
 3526 exits with return value **GrB_SUCCESS** and the new content of vector \mathbf{w} is as defined above but
 3527 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3528 sequence.

3529 4.3.4.2 eWiseMult: Matrix variant

3530 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-
 3531 ducing a third matrix as result.

3532 C Syntax

```

3533     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3534                           const GrB_Matrix Mask,
3535                           const GrB_BinaryOp accum,
3536                           const GrB_Semiring op,
3537                           const GrB_Matrix A,
3538                           const GrB_Matrix B,
3539                           const GrB_Descriptor desc);
3540
3541     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3542                           const GrB_Matrix Mask,
3543                           const GrB_BinaryOp accum,
3544                           const GrB_Monoid op,
3545                           const GrB_Matrix A,
3546                           const GrB_Matrix B,
3547                           const GrB_Descriptor desc);
3548
3549     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3550                           const GrB_Matrix Mask,
3551                           const GrB_BinaryOp accum,
3552                           const GrB_BinaryOp op,
3553                           const GrB_Matrix A,
3554                           const GrB_Matrix B,
3555                           const GrB_Descriptor desc);

```

3556 Parameters

3557 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3558 that may be accumulated with the result of the element-wise operation. On output,
3559 the matrix holds the results of the operation.

3560 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3561 stored into the output matrix C. The mask dimensions must match those of the
3562 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3563 of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types
3564 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
3565 dimensions of C), `GrB_NULL` should be specified.

3566 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3567 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3568 specified.

3569 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3570 operation. Depending on which type is passed, the following defines the binary
3571 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3572

BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3573

Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ignored.

3574

3575

Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$; the additive monoid is ignored.

3576

3577

A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the operation.

3578

3579

B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the operation.

3580

3581

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:

3582

3583

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.

3584

3585 Return Values

3586

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.

3587

3588

3589

3590

3591

GrB_PANIC Unknown internal error.

3592

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.

3593

3594

3595

3596

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3597

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).

3598

3599

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3600 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3601 corresponding domains of the binary operator (op) or accumulation
 3602 operator, or the mask's domain is not compatible with `bool` (in the
 3603 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3604 Description

3605 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:
 3606 $C = A \otimes B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \otimes B)$.
 3607 Logically, this operation occurs in three steps:

3608 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3609 and dimensions are tested for compatibility.

3610 **Compute** The indicated computations are carried out.

3611 **Output** The result is written into the output matrix, possibly under control of a mask.

3612 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3613 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3614 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)
- 3615 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3616 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3617 The argument matrices, the “product” operator (op), and the accumulation operator (if provided)
 3618 are tested for domain compatibility as follows:

- 3619 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3620 must be from one of the pre-defined types of Table 3.2.
- 3621 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3622 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3623 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3624 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3625 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3626 the accumulation operator.

3627 Two domains are compatible with each other if values from one domain can be cast to values in
 3628 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3629 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3630 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3631 error listed above is returned.

3632 From the argument matrices, the internal matrices and mask used in the computation are formed
 3633 (\leftarrow denotes copy):

- 3634 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3635 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3636 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3637 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3638 (b) If `Mask \neq GrB_NULL`,
 - 3639 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3640 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3641 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3642 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$.
 - 3643 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3644 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3645 4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3646 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3647 must hold:

- 3648 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3649 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3650 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3651 mismatch error listed above is returned.

3652 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3653 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3654 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3655 We describe this in terms of two intermediate matrices:

- 3656 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise product of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3657 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3658 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$
 3659 is created. The value of each of its elements is computed by

$$3660 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3661 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

3662 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

3663 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3664 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3665 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
3666 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3667 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3668 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3670 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3672 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3673 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
3674 using what is called a *standard matrix mask and replace*. This is carried out under control of the
3675 mask which acts as a “write mask”.

3676 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
3677 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3678 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3679 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
3680 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
3681 mask are unchanged:

$$3682 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3683 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
3684 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
3685 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
3686 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3687 sequence.

3688 4.3.5 eWiseAdd: Element-wise addition

3689 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation
3690 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of
3691 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”
3692 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
3693 the set of values from the resulting index set.

3694 4.3.5.1 eWiseAdd: Vector variant

3695 Perform element-wise (general) addition on the elements of two vectors, producing a third vector
3696 as result.

3697 C Syntax

```
3698     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3699                          const GrB_Vector mask,  
3700                          const GrB_BinaryOp accum,  
3701                          const GrB_Semiring op,  
3702                          const GrB_Vector u,  
3703                          const GrB_Vector v,  
3704                          const GrB_Descriptor desc);  
3705  
3706     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3707                          const GrB_Vector mask,  
3708                          const GrB_BinaryOp accum,  
3709                          const GrB_Monoid op,  
3710                          const GrB_Vector u,  
3711                          const GrB_Vector v,  
3712                          const GrB_Descriptor desc);  
3713  
3714     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3715                          const GrB_Vector mask,  
3716                          const GrB_BinaryOp accum,  
3717                          const GrB_BinaryOp op,  
3718                          const GrB_Vector u,  
3719                          const GrB_Vector v,  
3720                          const GrB_Descriptor desc);
```

3721 Parameters

3722 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3723 that may be accumulated with the result of the element-wise operation. On output,
3724 this vector holds the results of the operation.

3725 **mask** (IN) An optional “write” mask that controls which results from this operation are
3726 stored into the output vector **w**. The mask dimensions must match those of the
3727 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3728 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3729 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3730 dimensions of **w**), **GrB_NULL** should be specified.

3731 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3732 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 3733 specified.

3734 op (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
 3735 operation. Depending on which type is passed, the following defines the binary
 3736 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3737 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3738 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-
 3739 nored.

3740 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-
 3741 nary op and additive identity are ignored.

3742 u (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 3743 operation.

3744 v (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3745 operation.

3746 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3747 should be specified. Non-default field/value pairs are listed as follows:

3748

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.

3749

3750 Return Values

3751 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3752 blocking mode, this indicates that the compatibility tests on di-
 3753 mensions and domains for the input arguments passed successfully.
 3754 Either way, output vector w is ready to be used in the next method
 3755 of the sequence.

3756 GrB_PANIC Unknown internal error.

3757 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3758 GraphBLAS objects (input or output) is in an invalid state caused
 3759 by a previous execution error. Call GrB_error() to access any error
 3760 messages generated by the implementation.

3761 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3762 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3763 a call to `new` (or `dup` for vector parameters).

3764 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3765 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
3766 responding domains of the binary operator (`op`) or accumulation
3767 operator, or the mask's domain is not compatible with `bool` (in the
3768 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3769 Description

3770 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors: $\mathbf{w} =$
3771 $\mathbf{u} \oplus \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$. Logically,
3772 this operation occurs in three steps:

3773 **Setup** The internal vectors and mask used in the computation are formed and their domains
3774 and dimensions are tested for compatibility.

3775 **Compute** The indicated computations are carried out.

3776 **Output** The result is written into the output vector, possibly under control of a mask.

3777 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3778 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3779 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3780 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3781 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3782 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are
3783 tested for domain compatibility as follows:

- 3784 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
3785 must be from one of the pre-defined types of Table 3.2.
- 3786 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 3787 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 3788 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3789 5. $\mathbf{D}(\mathbf{u})$ and $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3790 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
3791 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
3792 the accumulation operator.

3793 Two domains are compatible with each other if values from one domain can be cast to values in
 3794 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3795 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3796 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch
 3797 error listed above is returned.

3798 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3799 denotes copy):

- 3800 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3801 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3802 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3803 (b) If `mask \neq GrB_NULL`,
 - 3804 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3805 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3806 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3807 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3808 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3809 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3810 must hold:

- 3811 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$.

3812 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 3813 mismatch error listed above is returned.

3814 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3815 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3816 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3817 We describe this in terms of two intermediate vectors:

- 3818 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “sum” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3819 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3820 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
 3821 value of each of its elements is computed by:

$$\begin{aligned}
 3822 \quad t_i &= (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})) \\
 3823 \quad & \\
 3824 \quad t_i &= \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})))
 \end{aligned}$$

3825
3826

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3827

where the difference operator in the previous expressions refers to set difference.

3828

The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

3829

- If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3830

- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

3831

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3832

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 $\tilde{\mathbf{t}}$.

3833

3834

$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

3835

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3836

3837

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3838

3839

where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3840

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”.

3841

3842

3843

- 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,

3844

3845

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3846

- 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:

3847

3848

3849

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3850

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.

3851

3852

3853

3854

3855

4.3.5.2 eWiseAdd: Matrix variant

3856

Perform element-wise (general) addition on the elements of two matrices, producing a third matrix as result.

3857

3858 C Syntax

```

3859         GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3860                               const GrB_Matrix Mask,
3861                               const GrB_BinaryOp accum,
3862                               const GrB_Semiring op,
3863                               const GrB_Matrix A,
3864                               const GrB_Matrix B,
3865                               const GrB_Descriptor desc);
3866
3867         GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3868                               const GrB_Matrix Mask,
3869                               const GrB_BinaryOp accum,
3870                               const GrB_Monoid op,
3871                               const GrB_Matrix A,
3872                               const GrB_Matrix B,
3873                               const GrB_Descriptor desc);
3874
3875         GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3876                               const GrB_Matrix Mask,
3877                               const GrB_BinaryOp accum,
3878                               const GrB_BinaryOp op,
3879                               const GrB_Matrix A,
3880                               const GrB_Matrix B,
3881                               const GrB_Descriptor desc);

```

3882 Parameters

3883 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3884 that may be accumulated with the result of the element-wise operation. On output,
3885 the matrix holds the results of the operation.

3886 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3887 stored into the output matrix C. The mask dimensions must match those of the
3888 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3889 of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types
3890 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
3891 dimensions of C), `GrB_NULL` should be specified.

3892 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3893 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3894 specified.

3895 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
3896 operation. Depending on which type is passed, the following defines the binary
3897 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3898 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 3899 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-
 3900 nored.
 3901 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-
 3902 nary op and additive identity are ignored.

3903 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3904 operation.

3905 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3906 operation.

3907 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3908 should be specified. Non-default field/value pairs are listed as follows:
 3909

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.

3911 Return Values

3912 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3913 blocking mode, this indicates that the compatibility tests on di-
 3914 mensions and domains for the input arguments passed successfully.
 3915 Either way, output matrix C is ready to be used in the next method
 3916 of the sequence.

3917 GrB_PANIC Unknown internal error.

3918 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3919 GraphBLAS objects (input or output) is in an invalid state caused
 3920 by a previous execution error. Call GrB_error() to access any error
 3921 messages generated by the implementation.

3922 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3923 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3924 a call to new (or Matrix_dup for matrix parameters).

3925 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3926 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3927 corresponding domains of the binary operator (op) or accumulation
 3928 operator, or the mask's domain is not compatible with `bool` (in the
 3929 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3930 Description

3931 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:
 3932 $C = A \oplus B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \oplus B)$.
 3933 Logically, this operation occurs in three steps:

3934 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3935 and dimensions are tested for compatibility.

3936 **Compute** The indicated computations are carried out.

3937 **Output** The result is written into the output matrix, possibly under control of a mask.

3938 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3939 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3940 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)
- 3941 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3942 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3943 The argument matrices, the “sum” operator (op), and the accumulation operator (if provided) are
 3944 tested for domain compatibility as follows:

- 3945 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3946 must be from one of the pre-defined types of Table 3.2.
- 3947 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3948 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3949 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3950 5. $\mathbf{D}(A)$ and $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3951 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3952 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3953 the accumulation operator.

3954 Two domains are compatible with each other if values from one domain can be cast to values in
 3955 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3956 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3957 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch
 3958 error listed above is returned.

3959 From the argument matrices, the internal matrices and mask used in the computation are formed
 3960 (\leftarrow denotes copy):

- 3961 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3962 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3963 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3964 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3965 (b) If `Mask \neq GrB_NULL`,
 - 3966 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3967 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3968 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3969 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 3970 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3971 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3972 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3973 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3974 must hold:

- 3975 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3976 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3977 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 3978 mismatch error listed above is returned.

3979 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3980 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3981 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3982 We describe this in terms of two intermediate matrices:

- 3983 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise sum of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3984 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3985 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$
 3986 is created. The value of each of its elements is computed by

$$\begin{aligned} 3987 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\ 3988 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\ 3989 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \end{aligned}$$

3992 where the difference operator in the previous expressions refers to set difference.

3993 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3994 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3995 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3996 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3997 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3998 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 3999 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4000 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4001 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \end{aligned}$$

4004 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4005 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4006 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4007 mask which acts as a “write mask”.

- 4008 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 4009 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4010 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4011 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 4012 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 4013 mask are unchanged:

$$4014 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4015 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4016 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4017 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 4018 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4019 sequence.

4020 4.3.6 extract: Selecting sub-graphs

4021 Extract a subset of a matrix or vector.

4022 4.3.6.1 extract: Standard vector variant

4023 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector
4024 whose size is equal to the number of indices.

4025 C Syntax

```
4026      GrB_Info GrB_extract(GrB_Vector      w,  
4027                          const GrB_Vector mask,  
4028                          const GrB_BinaryOp accum,  
4029                          const GrB_Vector u,  
4030                          const GrB_Index *indices,  
4031                          GrB_Index nindices,  
4032                          const GrB_Descriptor desc);
```

4033 Parameters

4034 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4035 that may be accumulated with the result of the extract operation. On output, this
4036 vector holds the results of the operation.

4037 **mask** (IN) An optional “write” mask that controls which results from this operation are
4038 stored into the output vector **w**. The mask dimensions must match those of the
4039 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4040 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4041 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4042 dimensions of **w**), **GrB_NULL** should be specified.

4043 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4044 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4045 specified.

4046 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4047 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of
4048 elements from **u** that are extracted. If all elements of **u** are to be extracted in order
4049 from 0 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution
4050 mode and return value, this array may be manipulated by the caller after this
4051 operation returns without affecting any deferred computations for this operation.

4052 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4053 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4054 should be specified. Non-default field/value pairs are listed as follows:

4055

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 .

4056

4057 Return Values

4058 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 4059 blocking mode, this indicates that the compatibility tests on
 4060 dimensions and domains for the input arguments passed suc-
 4061 cessfully. Either way, output vector **w** is ready to be used in the
 4062 next method of the sequence.

4063 GrB_PANIC Unknown internal error.

4064 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 4065 opaque GraphBLAS objects (input or output) is in an invalid
 4066 state caused by a previous execution error. Call GrB_error() to
 4067 access any error messages generated by the implementation.

4068 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4069 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
 4070 by a call to **new** (or **dup** for vector parameters).

4071 GrB_INDEX_OUT_OF_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-
 4072 blocking mode, this error can be deferred.

4073 GrB_DIMENSION_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices** \neq **size(w)**.

4074 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each
 4075 other or the corresponding domains of the accumulation oper-
 4076 ator, or the mask's domain is not compatible with **bool** (in the
 4077 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4078 GrB_NULL_POINTER Argument **row_indices** is a NULL pointer.

4079 Description

4080 This variant of GrB_extract computes the result of extracting a subset of locations from a Graph-
 4081 BLAS vector in a specific order: **w** = **u(indices)**; or, if an optional binary accumulation operator

4082 (\odot) is provided, $w = w \odot u(\text{indices})$. More explicitly:

$$4083 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4084 Logically, this operation occurs in three steps:

4085 **Setup** The internal vectors and mask used in the computation are formed and their domains
4086 and dimensions are tested for compatibility.

4087 **Compute** The indicated computations are carried out.

4088 **Output** The result is written into the output vector, possibly under control of a mask.

4089 Up to three argument vectors are used in this GrB_extract operation:

- 4090 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4091 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4092 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4093 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4094 bility as follows:

- 4095 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4096 must be from one of the pre-defined types of Table 3.2.
- 4097 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(u)$.
- 4098 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4099 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4100 mulation operator.

4101 Two domains are compatible with each other if values from one domain can be cast to values in
4102 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4103 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4104 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
4105 error listed above is returned.

4106 From the arguments, the internal vectors, mask, and index array used in the computation are
4107 formed (\leftarrow denotes copy):

- 4108 1. Vector $\tilde{w} \leftarrow w$.
- 4109 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
4110 (a) If `mask = GrB_NULL`, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.

- 4111 (b) If $\text{mask} \neq \text{GrB_NULL}$,
 4112 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 4113 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 4114 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
 4115 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
 4116 4. The internal index array, $\widetilde{\mathbf{I}}$, is computed from argument indices as follows:
 4117 (a) If $\text{indices} = \text{GrB_ALL}$, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.
 4118 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

4119 The internal vectors and mask are checked for dimension compatibility. The following conditions
 4120 must hold:

- 4121 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
 4122 2. $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$.

4123 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
 4124 match error listed above is returned.

4125 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4126 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4127 We are now ready to carry out the extract and any additional associated operations. We describe
 4128 this in terms of two intermediate vectors:

- 4129 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{w}}$.
- 4130 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4131 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4132 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4133 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
 4134 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`
 4135 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
 4136 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4137 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 4138 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 4139 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$4140 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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.

4.3.6.2 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.

4207 Return Values

4208	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
4209		blocking mode, this indicates that the compatibility tests on
4210		dimensions and domains for the input arguments passed suc-
4211		cessfully. Either way, output matrix C is ready to be used in the
4212		next method of the sequence.
4213	GrB_PANIC	Unknown internal error.
4214	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
4215		opaque GraphBLAS objects (input or output) is in an invalid
4216		state caused by a previous execution error. Call <code>GrB_error()</code> to
4217		access any error messages generated by the implementation.
4218	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
4219	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
4220		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4221	GrB_INDEX_OUT_OF_BOUNDS	A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or
4222		a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In
4223		non-blocking mode, this error can be deferred.
4224	GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, <code>nrows</code> \neq <code>nrows(C)</code> , or
4225		<code>ncols</code> \neq <code>ncols(C)</code> .
4226	GrB_DOMAIN_MISMATCH	The domains of the various matrices are incompatible with each
4227		other or the corresponding domains of the accumulation oper-
4228		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4229		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4230	GrB_NULL_POINTER	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4231		is a NULL pointer, or both.

4232 Description

4233 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified
 4234 rows and columns of a GraphBLAS matrix in a specific order: $C = A(\text{row_indices}, \text{col_indices})$; or,
 4235 if an optional binary accumulation operator (\odot) is provided, $C = C \odot A(\text{row_indices}, \text{col_indices})$.
 4236 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4238 Logically, this operation occurs in three steps:

4239 **Setup** The internal matrices and mask used in the computation are formed and their domains
 4240 and dimensions are tested for compatibility.

4241 **Compute** The indicated computations are carried out.

4242 **Output** The result is written into the output matrix, possibly under control of a mask.

4243 Up to three argument matrices are used in the `GrB_extract` operation:

- 4244 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
4245 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)
4246 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4247 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
4248 ibility as follows:

- 4249 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
4250 must be from one of the pre-defined types of Table 3.2.
4251 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
4252 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4253 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4254 mulation operator.

4255 Two domains are compatible with each other if values from one domain can be cast to values in
4256 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4257 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4258 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
4259 error listed above is returned.

4260 From the arguments, the internal matrices, mask, and index arrays used in the computation are
4261 formed (\leftarrow denotes copy):

- 4262 1. Matrix $\tilde{C} \leftarrow C$.
4263 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:
4264 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
4265 $j < \mathbf{ncols}(C)\} \rangle$.
4266 (b) If `Mask` \neq `GrB_NULL`,
4267 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
4268 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
4269 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
4270 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
4271 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
4272 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

- 4273 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4274 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 4275 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \text{nrows}$.
- 4276 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4277 (a) If `col_indices` = `GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$.
- 4278 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \text{ncols}$.

4279 The internal matrices and mask are checked for dimension compatibility. The following conditions
4280 must hold:

- 4281 1. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$.
- 4282 2. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$.
- 4283 3. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$.
- 4284 4. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$.

4285 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
4286 match error listed above is returned.

4287 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4288 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4289 We are now ready to carry out the extract and any additional associated operations. We describe
4290 this in terms of two intermediate matrices:

- 4291 • $\tilde{\mathbf{T}}$: The matrix holding the extraction from $\tilde{\mathbf{A}}$.
- 4292 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4293 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

4294
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{ (i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}}) \} \rangle.$$

4295 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 $\tilde{\mathbf{J}}$
4296 array is not in the range $[0, \text{ncols}(\tilde{\mathbf{A}}))$, the execution of `GrB_extract` ends and the index out-of-
4297 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
4298 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
4299 this point forward in the sequence.

4300 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4301 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

4302 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4303 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4304 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4305 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4306 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4307 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4309 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4311 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4312 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
4313 using what is called a *standard matrix mask and replace*. This is carried out under control of the
4314 mask which acts as a “write mask”.

4315 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
4316 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4317 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4318 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
4319 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
4320 mask are unchanged:

$$4321 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4322 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4323 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4324 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
4325 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4326 sequence.

4327 4.3.6.3 extract: Column (and row) variant

4328 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the
4329 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as
4330 well.

4331 C Syntax

```

4332         GrB_Info GrB_extract(GrB_Vector      w,
4333                             const GrB_Vector  mask,
4334                             const GrB_BinaryOp accum,
4335                             const GrB_Matrix  A,
4336                             const GrB_Index   *row_indices,
4337                             GrB_Index         nrows,
4338                             GrB_Index         col_index,
4339                             const GrB_Descriptor desc);

```

4340 Parameters

4341 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4342 that may be accumulated with the result of the extract operation. On output, this
4343 vector holds the results of the operation.

4344 **mask** (IN) An optional “write” mask that controls which results from this operation are
4345 stored into the output vector **w**. The mask dimensions must match those of the
4346 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4347 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4348 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4349 dimensions of **w**), **GrB_NULL** should be specified.

4350 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4351 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4352 specified.

4353 **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4354 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations
4355 within the specified column of **A** from which elements are extracted. If elements in
4356 all rows of **A** are to be extracted in order, **GrB_ALL** should be specified. Regardless
4357 of execution mode and return value, this array may be manipulated by the caller
4358 after this operation returns without affecting any deferred computations for this
4359 operation.

4360 **nrows** (IN) The number of indices in the **row_indices** array. Must be equal to **size(w)**.

4361 **col_index** (IN) The index of the column of **A** from which to extract values. It must be in the
4362 range $[0, \mathbf{ncols}(A))$.

4363 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4364 should be specified. Non-default field/value pairs are listed as follows:

4365

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

4393 an optional binary accumulation operator (\odot) is provided, $w = w \odot A(:, \text{col_index})(\text{row_indices})$.
 4394 More explicitly:

$$4395 \quad \begin{aligned} w(i) &= A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4396 Logically, this operation occurs in three steps:

4397 **Setup** The internal matrices, vectors, and mask used in the computation are formed and their
 4398 domains and dimensions are tested for compatibility.

4399 **Compute** The indicated computations are carried out.

4400 **Output** The result is written into the output vector, possibly under control of a mask.

4401 Up to three argument vectors and matrices are used in this GrB_extract operation:

- 4402 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4403 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4404 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4405 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain
 4406 compatibility as follows:

- 4407 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
 4408 must be from one of the pre-defined types of Table 3.2.
- 4409 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(A)$.
- 4410 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4411 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4412 mulation operator.

4413 Two domains are compatible with each other if values from one domain can be cast to values in
 4414 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4415 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4416 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
 4417 error listed above is returned.

4418 From the arguments, the internal vector, matrix, mask, and index array used in the computation
 4419 are formed (\leftarrow denotes copy):

- 4420 1. Vector $\tilde{w} \leftarrow w$.
- 4421 2. One-dimensional mask, \tilde{m} , is computed from argument **mask** as follows:
 4422 (a) If **mask** = GrB_NULL, then $\tilde{m} = \langle \mathbf{size}(w), \{i, \forall i : 0 \leq i < \mathbf{size}(w)\} \rangle$.

4423 (b) If $\text{mask} \neq \text{GrB_NULL}$,
 4424 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 4425 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 4426 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
 4427 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
 4428 4. The internal row index array, $\widetilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
 4429 (a) If `indices = GrB_ALL`, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
 4430 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$.

4431 The internal vector, `mask`, and index array are checked for dimension compatibility. The following
 4432 conditions must hold:

- 4433 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4434 2. $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$.

4435 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
 4436 match error listed above is returned.

4437 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4438 1. $0 \leq \text{col_index} < \text{ncols}(\mathbf{A})$

4439 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above
 4440 is returned.

4441 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4442 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4443 We are now ready to carry out the extract and any additional associated operations. We describe
 4444 this in terms of two intermediate vectors:

- 4445 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from a column of $\widetilde{\mathbf{A}}$.
- 4446 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4447 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4448 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4449 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`
 4450 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,
 4451 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result
 4452 vector, \mathbf{w} , is invalid from this point forward in the sequence.

4453 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

4454 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

4455 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4456 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4457 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4458 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4459 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

$$4460 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4461 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4462 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4463 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 4464 using what is called a *standard vector mask and replace*. This is carried out under control of the
 4465 mask which acts as a “write mask”.

4466 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 4467 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4470 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4471 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 4472 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 4473 mask are unchanged:

$$4474 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4475 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4476 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4477 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
 4478 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4479 sequence.

4480 4.3.7 assign: Modifying sub-graphs

4481 Assign the contents of a subset of a matrix or vector.

4482 4.3.7.1 assign: Standard vector variant

4483 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.
 4484 The size of the input vector is the same size as the index array provided.

4485 C Syntax

```
4486         GrB_Info GrB_assign(GrB_Vector      w,  
4487                             const GrB_Vector mask,  
4488                             const GrB_BinaryOp accum,  
4489                             const GrB_Vector u,  
4490                             const GrB_Index *indices,  
4491                             GrB_Index      nindices,  
4492                             const GrB_Descriptor desc);
```

4493 Parameters

4494 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4495 that may be accumulated with the result of the assign operation. On output, this
4496 vector holds the results of the operation.

4497 **mask** (IN) An optional “write” mask that controls which results from this operation are
4498 stored into the output vector **w**. The mask dimensions must match those of the
4499 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4500 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4501 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4502 dimensions of **w**), **GrB_NULL** should be specified.

4503 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4504 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4505 specified.

4506 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4507 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4508 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0
4509 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4510 and return value, this array may be manipulated by the caller after this operation
4511 returns without affecting any deferred computations for this operation. If this
4512 array contains duplicate values, it implies in assignment of more than one value to
4513 the same location which leads to undefined results.

4514 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4515 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4516 should be specified. Non-default field/value pairs are listed as follows:
4517

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:

$$\begin{aligned}
 w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
 \end{aligned}$$

4546 Logically, this operation occurs in three steps:

4547 **Setup** The internal vectors and mask used in the computation are formed and their domains
4548 and dimensions are tested for compatibility.

4549 **Compute** The indicated computations are carried out.

4550 **Output** The result is written into the output vector, possibly under control of a mask.

4551 Up to three argument vectors are used in the `GrB_assign` operation:

- 4552 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4553 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4554 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4555 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4556 bility as follows:

- 4557 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
4558 must be from one of the pre-defined types of Table 3.2.
- 4559 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 4560 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
4561 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
4562 mulation operator.

4563 Two domains are compatible with each other if values from one domain can be cast to values in
4564 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4565 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4566 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4567 error listed above is returned.

4568 From the arguments, the internal vectors, mask and index array used in the computation are formed
4569 (\leftarrow denotes copy):

- 4570 1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 4571 2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 4572 (a) If `mask` = `GrB_NULL`, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 4573 (b) If `mask` \neq `GrB_NULL`,
 - 4574 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 4575 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 4576 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.

4577 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4578 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:

4579 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i$, $\forall i : 0 \leq i < \text{nindices}$.

4580 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i]$, $\forall i : 0 \leq i < \text{nindices}$.

4581 The internal vector and mask are checked for dimension compatibility. The following conditions
4582 must hold:

4583 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4584 2. $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$.

4585 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4586 match error listed above is returned.

4587 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4588 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4589 We are now ready to carry out the assign and any additional associated operations. We describe
4590 this in terms of two intermediate vectors:

4591 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.

4592 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4593 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

4594
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4595 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{w}}$, computation
4596 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`
4597 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4598 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4599 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

4600 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

4601
$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4602 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4603 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
4604 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4605 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4606 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

4607
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4608
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4609

4610 where the difference operator refers to set difference.

4611 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4612 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4613 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4614 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 4615 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 4616 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4617 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4618 \end{aligned}$$

4619 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4621 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
4622 using what is called a *standard vector mask and replace*. This is carried out under control of the
4623 mask which acts as a “write mask”.

4624 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
4625 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4626 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4627 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4628 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
4629 mask are unchanged:

$$4630 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4631 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4632 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4633 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
4634 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4635 sequence.

4636 4.3.7.2 assign: Standard matrix variant

4637 Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices.
4638 The dimensions of the input matrix are the same size as the row and column index arrays provided.

4639 C Syntax

```
4640      GrB_Info GrB_assign(GrB_Matrix      C,
4641                        const GrB_Matrix  Mask,
4642                        const GrB_BinaryOp accum,
4643                        const GrB_Matrix  A,
```

```

4644         const GrB_Index      *row_indices,
4645         GrB_Index             nrows,
4646         const GrB_Index      *col_indices,
4647         GrB_Index             ncols,
4648         const GrB_Descriptor desc);

```

4649 Parameters

4650 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
4651 that may be accumulated with the result of the assign operation. On output, the
4652 matrix holds the results of the operation.

4653 **Mask** (IN) An optional “write” mask that controls which results from this operation are
4654 stored into the output matrix **C**. The mask dimensions must match those of the
4655 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4656 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
4657 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4658 dimensions of **C**), **GrB_NULL** should be specified.

4659 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4660 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4661 specified.

4662 **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4663 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
4664 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
4665 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
4666 this array may be manipulated by the caller after this operation returns without
4667 affecting any deferred computations for this operation. If this array contains du-
4668 plicate values, it implies assignment of more than one value to the same location
4669 which leads to undefined results.

4670 **nrows** (IN) The number of values in the **row_indices** array. Must be equal to **nrows(A)**
4671 if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4672 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns
4673 of **C** that are assigned. If all columns of **C** are to be assigned in order from 0
4674 to **ncols** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4675 and return value, this array may be manipulated by the caller after this operation
4676 returns without affecting any deferred computations for this operation. If this
4677 array contains duplicate values, it implies assignment of more than one value to
4678 the same location which leads to undefined results.

4679 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **ncols(A)** if **A** is
4680 not transposed, or equal to **nrows(A)** if **A** is transposed.

4681
4682
4683

4684

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.

4685 Return Values

4686
4687
4688
4689
4690

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.

4691

GrB_PANIC Unknown internal error.

4692
4693
4694
4695

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.

4696

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

4697
4698

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).

4699
4700
4701

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.

4702
4703

GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrow \neq nrow(A), or ncols \neq ncols(A).

4704
4705
4706
4707

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).

4708
4709

GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices is a NULL pointer, or both.

4710 Description

4711 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows
 4712 and columns in `C` in a specified order: $C(\text{row_indices}, \text{col_indices}) = A$; or, if an optional binary
 4713 accumulation operator (\odot) is provided, $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot$
 4714 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4715 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot A(i, j), \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4716 Logically, this operation occurs in three steps:

4717 Setup The internal matrices and mask used in the computation are formed and their domains
 4718 and dimensions are tested for compatibility.

4719 Compute The indicated computations are carried out.

4720 Output The result is written into the output matrix, possibly under control of a mask.

4721 Up to three argument matrices are used in the `GrB_assign` operation:

- 4722 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4723 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)
- 4724 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4725 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
 4726 ibility as follows:

- 4727 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 4728 must be from one of the pre-defined types of Table 3.2.
- 4729 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 4730 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4731 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4732 mulation operator.

4733 Two domains are compatible with each other if values from one domain can be cast to values in
 4734 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4735 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4736 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4737 error listed above is returned.

4738 From the arguments, the internal matrices, mask, and index arrays used in the computation are
 4739 formed (\leftarrow denotes copy):

- 4740 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4741 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
- 4742 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
4743 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 4744 (b) If `Mask \neq GrB_NULL`,
- 4745 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
4746 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
- 4747 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
4748 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
- 4749 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4750 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 4751 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4752 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
- 4753 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 4754 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4755 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
- 4756 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4757 The internal matrices and mask are checked for dimension compatibility. The following conditions
4758 must hold:

- 4759 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 4760 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 4761 3. $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$.
- 4762 4. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$.

4763 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4764 match error listed above is returned.

4765 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4766 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4767 We are now ready to carry out the assign and any additional associated operations. We describe
4768 this in terms of two intermediate vectors:

- 4769 • $\tilde{\mathbf{T}}$: The matrix holding the contents from $\tilde{\mathbf{A}}$ in their destination locations relative to $\tilde{\mathbf{C}}$.
- 4770 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4771 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4772 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \} \}.$$

4773 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
 4774 $\tilde{\mathbf{J}}$ array is not in the range $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-
 4775 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
 4776 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
 4777 this point forward in the sequence.

4778 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 4779 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$4780 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4781 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})) \} \}.$$

4782 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 4783 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
 4784 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

4785 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4786 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4787 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4788 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4790 where the difference operator refers to set difference.

- 4791 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4792 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}}) \} \}.$$

4793 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4794 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4795 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4796 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4797 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4800 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4801 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4802 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4803 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

4837 bool or any of the predefined “built-in” types in Table 3.2. If the default mask
 4838 is desired (i.e., a mask that is all true with the dimensions of a column of C),
 4839 GrB_NULL should be specified.

4840 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 4841 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 4842 specified.

4843 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column
 4844 of C.

4845 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
 4846 the specified column of C that are to be assigned. If all elements of the column
 4847 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB_ALL should
 4848 be specified. Regardless of execution mode and return value, this array may be
 4849 manipulated by the caller after this operation returns without affecting any de-
 4850 ferred computations for this operation. If this array contains duplicate values, it
 4851 implies in assignment of more than one value to the same location which leads to
 4852 undefined results.

4853 **nrows** (IN) The number of values in row_indices array. Must be equal to **size(u)**.

4854 **col_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4855 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4856 should be specified. Non-default field/value pairs are listed as follows:

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.

4859 Return Values

4860 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 4861 blocking mode, this indicates that the compatibility tests on
 4862 dimensions and domains for the input arguments passed suc-
 4863 cessfully. Either way, output matrix C is ready to be used in the
 4864 next method of the sequence.

4865 **GrB_PANIC** Unknown internal error.

4897 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4898 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
4899 compatibility as follows:

- 4900 1. If **mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\mathbf{mask})$
4901 must be from one of the pre-defined types of Table 3.2.
- 4902 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 4903 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
4904 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
4905 mulation operator.

4906 Two domains are compatible with each other if values from one domain can be cast to values in
4907 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4908 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4909 any compatibility rule above is violated, execution of **GrB_assign** ends and the domain mismatch
4910 error listed above is returned.

4911 The **col_index** parameter is checked for a valid value. The following condition must hold:

- 4912 1. $0 \leq \mathbf{col_index} < \mathbf{ncols}(\mathbf{C})$

4913 If the rule above is violated, execution of **GrB_assign** ends and the invalid index error listed above
4914 is returned.

4915 From the arguments, the internal vectors, **mask**, and index array used in the computation are
4916 formed (\leftarrow denotes copy):

- 4917 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a column of **C** as follows:

$$4918 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \mathbf{col_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

- 4919 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument **mask** as follows:

- 4920 (a) If **mask** = **GrB_NULL**, then $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$.
- 4921 (b) If **mask** \neq **GrB_NULL**,
 - 4922 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 4923 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 4924 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

- 4925 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

- 4926 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument **row_indices** as follows:

- 4927 (a) If **row_indices** = **GrB_ALL**, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.

4928 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i]$, $\forall i : 0 \leq i < \text{nrows}$.

4929 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
4930 conditions must hold:

- 4931 1. $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4932 2. $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$.

4933 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4934 match error listed above is returned.

4935 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4936 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4937 We are now ready to carry out the assign and any additional associated operations. We describe
4938 this in terms of two intermediate vectors:

- 4939 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 4940 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4941 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4942 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4943 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
4944 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
4945 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4946 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4947 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4948 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4949 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4950 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4951 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
4952 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4953 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4954 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$4955 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4956 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4958 where the difference operator refers to set difference.

- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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 $\tilde{\mathbf{t}}$.

$$z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final result matrix, $\mathbf{C}(:, \text{col_index})$. 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 $\mathbf{C}(:, \text{col_index})$ on input to this operation are deleted and the new contents of the column is given by:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are copied into the column of the final result matrix, $\mathbf{C}(:, \text{col_index})$, and elements of this column that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} \mathbf{L}(\mathbf{C}) = & \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \\ & \{(i, \text{col_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ & \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

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.

4.3.7.4 assign: Row variant

Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the output cannot be transposed, a different variant of `assign` is provided to assign to a column of a matrix.

4989 C Syntax

```
4990         GrB_Info GrB_assign(GrB_Matrix      C,  
4991                             const GrB_Vector mask,  
4992                             const GrB_BinaryOp accum,  
4993                             const GrB_Vector u,  
4994                             GrB_Index      row_index,  
4995                             const GrB_Index *col_indices,  
4996                             GrB_Index      ncols,  
4997                             const GrB_Descriptor desc);
```

4998 Parameters

4999 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values
5000 that may be accumulated with the result of the assign operation. On output, this
5001 matrix holds the results of the operation.

5002 **mask** (IN) An optional “write” mask that controls which results from this operation are
5003 stored into the specified row of the output matrix **C**. The mask dimensions must
5004 match those of a single row of the matrix **C**. If the **GrB_STRUCTURE** descriptor
5005 is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or
5006 any of the predefined “built-in” types in Table 3.2. If the default mask is desired
5007 (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB_NULL** should
5008 be specified.

5009 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5010 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5011 specified.

5012 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of
5013 **C**.

5014 **row_index** (IN) The index of the row in **C** to assign. Must be in the range $[0, \mathbf{nrows}(\mathbf{C})]$.

5015 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
5016 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to
5017 be assigned in order from index 0 to $\mathbf{ncols} - 1$, then **GrB_ALL** should be specified.
5018 Regardless of execution mode and return value, this array may be manipulated by
5019 the caller after this operation returns without affecting any deferred computations
5020 for this operation. If this array contains duplicate values, it implies in assignment
5021 of more than one value to the same location which leads to undefined results.

5022 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **size(u)**.

5023 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5024 should be specified. Non-default field/value pairs are listed as follows:
5025

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:

5054 $C(\text{row_index}, :) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(\text{row_index}, :$
5055 $) = C(\text{row_index}, :) \odot u$. Taking order of `col_indices` into account it is more explicitly written as:

5056 $C(\text{row_index}, \text{col_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$
 $C(\text{row_index}, \text{col_indices}[j]) = C(\text{row_index}, \text{col_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$

5057 Logically, this operation occurs in three steps:

5058 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
5059 domains and dimensions are tested for compatibility.

5060 **Compute** The indicated computations are carried out.

5061 **Output** The result is written into the output matrix, possibly under control of a mask.

5062 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5063 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
5064 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
5065 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5066 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
5067 compatibility as follows:

- 5068 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5069 must be from one of the pre-defined types of Table 3.2.
5070 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(u)$.
5071 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5072 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
5073 mulation operator.

5074 Two domains are compatible with each other if values from one domain can be cast to values in
5075 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5076 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5077 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
5078 error listed above is returned.

5079 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5080 1. $0 \leq \text{row_index} < \mathbf{nrows}(C)$

5081 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
5082 is returned.

5083 From the arguments, the internal vectors, mask, and index array used in the computation are
5084 formed (\leftarrow denotes copy):

5085 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a row of \mathbf{C} as follows:

$$5086 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5087 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

5088 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$.

5089 (b) If `mask \neq GrB_NULL`,

5090 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

5091 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

5092 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

5093 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5094 4. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:

5095 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.

5096 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

5097 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
5098 conditions must hold:

5099 1. $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5100 2. $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$.

5101 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
5102 match error listed above is returned.

5103 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5104 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5105 We are now ready to carry out the assign and any additional associated operations. We describe
5106 this in terms of two intermediate vectors:

- 5107 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 5108 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5109 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$5110 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5111 At this point, if any value of $\tilde{\mathbf{J}}[j]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
5112 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
5113 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
5114 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

5115 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5116 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$5117 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5118 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
5119 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
5120 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5121 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5122 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$5123 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5124 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5126 where the difference operator refers to set difference.

5127 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5128 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5129 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5130 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5131 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5132 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5133 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5136 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5137 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
5138 result matrix, $\mathbf{C}(\text{row_index}, :)$. This is carried out under control of the mask which acts as a “write
5139 mask”.

5140 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(\text{row_index}, :)$ on input to this
5141 operation are deleted and the new contents of the column is given by:

$$5142 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5143 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
5144 copied into the column of the final result matrix, $\mathbf{C}(\text{row_index}, :)$, and elements of this column
5145 that fall outside the set indicated by the mask are unchanged:

$$5146 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup$$

$$5147 \quad \{(\text{row_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5148 \quad \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5149 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5150 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5151 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may
5152 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5153 **4.3.7.5 assign: Constant vector variant**[Scott: NEW CONTENT]

5154 Assign the same value to a specified subset of vector elements. With the use of GrB_ALL, the entire
5155 destination vector can be filled with the constant.

5156 **C Syntax**

```
5157         GrB_Info GrB_assign(GrB_Vector          w,  
5158                             const GrB_Vector    mask,  
5159                             const GrB_BinaryOp    accum,  
5160                             <type>              val,  
5161                             const GrB_Index      *indices,  
5162                             GrB_Index            nindices,  
5163                             const GrB_Descriptor desc);
```

```
5164         GrB_Info GrB_assign(GrB_Vector          w,  
5165                             const GrB_Vector    mask,  
5166                             const GrB_BinaryOp    accum,  
5167                             const GrB_Scalar      s,  
5168                             const GrB_Index      *indices,  
5169                             GrB_Index            nindices,  
5170                             const GrB_Descriptor desc);
```

5171 **Parameters**

5172 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5173 that may be accumulated with the result of the assign operation. On output, this
5174 vector holds the results of the operation.

5175 **mask** (IN) An optional “write” mask that controls which results from this operation are
5176 stored into the output vector w. The mask dimensions must match those of the
5177 vector w. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5178 of the mask vector must be of type bool or any of the predefined “built-in” types
5179 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5180 dimensions of w), GrB_NULL should be specified.

5181 **accum** (IN) An optional binary operator used for accumulating entries into existing w
5182 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5183 specified.

5184 **val** (IN) Scalar value to assign to (a subset of) w.

5185 **s** (IN) Scalar value to assign to (a subset of) w.

5186 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
5187 w that are to be assigned. If all elements of w are to be assigned in order from 0

5188 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode
5189 and return value, this array may be manipulated by the caller after this operation
5190 returns without affecting any deferred computations for this operation. In this
5191 variant, the specific order of the values in the array has no effect on the result.
5192 Unlike other variants, if there are duplicated values in this array the result is still
5193 defined.

5194 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If
5195 `nindices` is zero, the operation becomes a NO-OP.

5196 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
5197 should be specified. Non-default field/value pairs are listed as follows:

5198

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.
5199 <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> .

5200 Return Values

5201 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5202 blocking mode, this indicates that the compatibility tests on
5203 dimensions and domains for the input arguments passed suc-
5204 cessfully. Either way, output vector `w` is ready to be used in the
5205 next method of the sequence.

5206 **GrB_PANIC** Unknown internal error.

5207 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
5208 opaque GraphBLAS objects (input or output) is in an invalid
5209 state caused by a previous execution error. Call `GrB_error()` to
5210 access any error messages generated by the implementation.

5211 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5212 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized
5213 by a call to `new` (or `dup` for vector parameters).

5214 **GrB_INDEX_OUT_OF_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-
5215 blocking mode, this can be reported as an execution error.

5216 **GrB_DIMENSION_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less
5217 than `size(w)`.

5248 4. If **accum** is not **GrB_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the
 5249 method, must be compatible with **D_{in2}(accum)** of the accumulation operator.

5250 Two domains are compatible with each other if values from one domain can be cast to values in
 5251 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 5252 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5253 any compatibility rule above is violated, execution of **GrB_assign** ends and the domain mismatch
 5254 error listed above is returned.

5255 From the arguments, the internal vectors, mask and index array used in the computation are formed
 5256 (\leftarrow denotes copy):

- 5257 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5258 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument **mask** as follows:
 - 5259 (a) If **mask** = **GrB_NULL**, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 5260 (b) If **mask** \neq **GrB_NULL**,
 - 5261 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 5262 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 5263 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5264 3. Scalar $\tilde{s} \leftarrow s$ (**GrB_Scalar** version only).
- 5265 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument **indices** as follows:
 - 5266 (a) If **indices** = **GrB_ALL**, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$.
 - 5267 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$.

5268 The internal vector and mask are checked for dimension compatibility. The following conditions
 5269 must hold:

- 5270 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5271 2. $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$.

5272 If any compatibility rule above is violated, execution of **GrB_assign** ends and the dimension mis-
 5273 match error listed above is returned.

5274 From this point forward, in **GrB_NONBLOCKING** mode, the method can optionally exit with
 5275 **GrB_SUCCESS** return code and defer any computation and/or execution error codes.

5276 We are now ready to carry out the assign and any additional associated operations. We describe
 5277 this in terms of two intermediate vectors:

- 5278 • $\tilde{\mathbf{t}}$: The vector holding the copies of the scalar, either **val** or \tilde{s} , in their destination locations
 5279 relative to $\tilde{\mathbf{w}}$.

5280 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5281 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows. If a non-opaque scalar \mathbf{val} is provided:

$$5282 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5283 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 1$):

$$5284 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5285 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 0$):

$$5286 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5287 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
 5288 is not in the range $[0, \mathbf{size}(\tilde{\mathbf{w}}))$, the execution of `GrB_assign` ends and the index out-of-bounds
 5289 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a
 5290 sequence-terminating `GrB_wait()` is called. Regardless, the result vector, \mathbf{w} , is invalid from this
 5291 point forward in the sequence.

5292 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5293 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$5294 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5295 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 5296 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
 5297 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5298 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5299 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5300 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5301 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5303 where the difference operator refers to set difference. We note that in this case of assigning
 5304 a constant, $\{\tilde{\mathbf{I}}[k], \forall k\}$ and $\mathbf{ind}(\tilde{\mathbf{t}})$ are identical.

5305 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5306 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5307 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5308 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5309 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$5310 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5311 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5312 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5315 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 5316 using what is called a *standard vector mask and replace*. This is carried out under control of the
 5317 mask which acts as a “write mask”.

- 5318 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 5319 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$5320 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5321 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 5322 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 5323 mask are unchanged:

$$5324 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5325 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 5326 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 5327 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 5328 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 5329 sequence.

5330 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5331 Assign the same value to a specified subset of matrix elements. With the use of GrB_ALL, the
 5332 entire destination matrix can be filled with the constant.

5333 C Syntax

```
5334      GrB_Info GrB_assign(GrB_Matrix      C,
5335                          const GrB_Matrix Mask,
5336                          const GrB_BinaryOp accum,
5337                          <type>         val,
5338                          const GrB_Index *row_indices,
5339                          GrB_Index      nrows,
5340                          const GrB_Index *col_indices,
5341                          GrB_Index      ncols,
5342                          const GrB_Descriptor desc);
```

```
5343      GrB_Info GrB_assign(GrB_Matrix      C,
5344                          const GrB_Matrix Mask,
5345                          const GrB_BinaryOp accum,
5346                          const GrB_Scalar s,
5347                          const GrB_Index *row_indices,
5348                          GrB_Index      nrows,
```

```

5349             const GrB_Index      *col_indices,
5350             GrB_Index             ncols,
5351             const GrB_Descriptor  desc);

```

5352 Parameters

5353 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5354 that may be accumulated with the result of the assign operation. On output, the
5355 matrix holds the results of the operation.

5356 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5357 stored into the output matrix **C**. The mask dimensions must match those of the
5358 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5359 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
5360 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5361 dimensions of **C**), **GrB_NULL** should be specified.

5362 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5363 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5364 specified.

5365 **val** (IN) Scalar value to assign to (a subset of) **C**.

5366 **s** (IN) Scalar value to assign to (a subset of) **C**.

5367 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
5368 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
5369 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
5370 this array may be manipulated by the caller after this operation returns without
5371 affecting any deferred computations for this operation. Unlike other variants, if
5372 there are duplicated values in this array the result is still defined.

5373 **nrows** (IN) The number of values in **row_indices** array. Must be in the range: [0, **nrows**(**C**)].
5374 If **nrows** is zero, the operation becomes a NO-OP.

5375 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**
5376 that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** – 1,
5377 then **GrB_ALL** should be specified. Regardless of execution mode and return value,
5378 this array may be manipulated by the caller after this operation returns without
5379 affecting any deferred computations for this operation. Unlike other variants, if
5380 there are duplicated values in this array the result is still defined.

5381 **ncols** (IN) The number of values in **col_indices** array. Must be in the range: [0, **ncols**(**C**)].
5382 If **ncols** is zero, the operation becomes a NO-OP.

5383 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5384 should be specified. Non-default field/value pairs are listed as follows:

5385

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`

5415 or $C(\text{row_indices}, \text{col_indices}) = s$ is performed. If an optional binary accumulation operator
 5416 (\odot) is provided, then either $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot \text{val}$ or
 5417 $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot s$ is performed. More explicitly, if a
 5418 non-opaque value val is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = \text{val}, \text{ or} \\ 5419 \quad & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5420 Correspondingly, if a `GrB_Scalar` s is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = s, \text{ or} \\ 5421 \quad & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5422 Logically, this operation occurs in three steps:

5423 Setup The internal vectors and mask used in the computation are formed and their domains
 5424 and dimensions are tested for compatibility.

5425 Compute The indicated computations are carried out.

5426 Output The result is written into the output matrix, possibly under control of a mask.

5427 Up to two argument matrices are used in the `GrB_assign` operation:

- 5428 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5429 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)

5430 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain
 5431 compatibility as follows:

- 5432 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 5433 must be from one of the pre-defined types of Table 3.2.
- 5434 2. $\mathbf{D}(C)$ must be compatible with either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5435 method.
- 5436 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5437 of the accumulation operator.
- 5438 4. If `accum` is not `GrB_NULL`, then either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5439 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, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
3. 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, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
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, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

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.

5472 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5473 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5474 We are now ready to carry out the assign and any additional associated operations. We describe
 5475 this in terms of two intermediate matrices:

- 5476 • $\tilde{\mathbf{T}}$: The matrix holding the copies of the scalar, either `val` or \tilde{s} , in their destination locations
 5477 relative to $\tilde{\mathbf{C}}$.
- 5478 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5479 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows. If a non-opaque scalar `val` is provided:

$$5480 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5481 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 1):

$$5482 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5483 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 0):

$$5484 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5485 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
 5486 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
 5487 $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-bounds error listed above is
 5488 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating
 5489 `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from this point forward in the
 5490 sequence.

5491 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 5492 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$5493 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5494 \quad \{(i, j, Z_{ij}) \mid (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}}))\} \rangle.$$

5495 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 5496 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
 5497 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

5498 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 5499 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5500 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5501 \\ 5502 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

5503 where the difference operator refers to set difference. We note that, in this particular case of
 5504 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.

- If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} , using what is called a *standard matrix 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{C} on input to this operation are deleted and the content of the new output matrix, \mathbf{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, \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.

4.3.8 apply: Apply a function to the elements of an object

Computes the transformation of the values of the elements of a vector or a matrix using a unary function, or a binary function where one argument is bound to a scalar.

4.3.8.1 apply: Vector variant

Computes the transformation of the values of the elements of a vector using a unary function.

5535 C Syntax

```

5536      GrB_Info GrB_apply(GrB_Vector      w,
5537                        const GrB_Vector  mask,
5538                        const GrB_BinaryOp accum,
5539                        const GrB_UnaryOp  op,
5540                        const GrB_Vector  u,
5541                        const GrB_Descriptor desc);

```

5542 Parameters

5543 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5544 that may be accumulated with the result of the apply operation. On output, this
5545 vector holds the results of the operation.

5546 **mask** (IN) An optional “write” mask that controls which results from this operation are
5547 stored into the output vector **w**. The mask dimensions must match those of the
5548 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5549 of the mask vector must be of type **bool** or any of the predefined “built-in” types
5550 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5551 dimensions of **w**), **GrB_NULL** should be specified.

5552 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5553 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5554 specified.

5555 **op** (IN) A unary operator applied to each element of input vector **u**.

5556 **u** (IN) The GraphBLAS vector to which the unary function is applied.

5557 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5558 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 .

5561 Return Values

5562 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5563 blocking mode, this indicates that the compatibility tests on di-
5564 mensions and domains for the input arguments passed successfully.

5565 Either way, output vector w is ready to be used in the next method
5566 of the sequence.

5567 **GrB_PANIC** Unknown internal error.

5568 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
5569 GraphBLAS objects (input or output) is in an invalid state caused
5570 by a previous execution error. Call **GrB_error()** to access any error
5571 messages generated by the implementation.

5572 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5573 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
5574 a call to **new** (or **dup** for vector parameters).

5575 **GrB_DIMENSION_MISMATCH** $mask$, w and/or u dimensions are incompatible.

5576 **GrB_DOMAIN_MISMATCH** The domains of the various vectors are incompatible with the corre-
5577 sponding domains of the accumulation operator or unary function,
5578 or the mask's domain is not compatible with **bool** (in the case where
5579 $desc[GrB_MASK].GrB_STRUCTURE$ is not set).

5580 **Description**

5581 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
5582 GraphBLAS vector: $w = f(u)$; or, if an optional binary accumulation operator (\odot) is provided,
5583 $w = w \odot f(u)$.

5584 Logically, this operation occurs in three steps:

5585 **Setup** The internal vectors and mask used in the computation are formed and their domains
5586 and dimensions are tested for compatibility.

5587 **Compute** The indicated computations are carried out.

5588 **Output** The result is written into the output vector, possibly under control of a mask.

5589 Up to three argument vectors are used in this **GrB_apply** operation:

- 5590 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5591 2. $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$ (optional)
- 5592 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5593 The argument vectors, unary operator and the accumulation operator (if provided) are tested for
5594 domain compatibility as follows:

- 5595 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5596 must be from one of the pre-defined types of Table 3.2.
- 5597 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5598 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5599 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5600 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5601 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in}(\text{op})$.

5602 Two domains are compatible with each other if values from one domain can be cast to values in
5603 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5604 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5605 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
5606 error listed above is returned.

5607 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
5608 denotes copy):

- 5609 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5610 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 5611 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 5612 (b) If `mask` \neq `GrB_NULL`,
 - 5613 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 5614 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 5615 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5616 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5617 The internal vectors and masks are checked for dimension compatibility. The following conditions
5618 must hold:

- 5619 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5620 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

5621 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5622 error listed above is returned.

5623 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5624 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5625 We are now ready to carry out the apply and any additional associated operations. We describe
5626 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.

5659 4.3.8.2 apply: Matrix variant

5660 Computes the transformation of the values of the elements of a matrix using a unary function.

5661 C Syntax

```
5662      GrB_Info GrB_apply(GrB_Matrix      C,
5663                        const GrB_Matrix  Mask,
5664                        const GrB_BinaryOp accum,
5665                        const GrB_UnaryOp  op,
5666                        const GrB_Matrix  A,
5667                        const GrB_Descriptor desc);
```

5668 Parameters

5669 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5670 that may be accumulated with the result of the apply operation. On output, the
5671 matrix holds the results of the operation.

5672 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5673 stored into the output matrix C. The mask dimensions must match those of the
5674 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5675 of the Mask matrix must be of type bool or any of the predefined “built-in” types
5676 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5677 dimensions of C), GrB_NULL should be specified.

5678 **accum** (IN) An optional binary operator used for accumulating entries into existing C
5679 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5680 specified.

5681 **op** (IN) A unary operator applied to each element of input matrix A.

5682 **A** (IN) The GraphBLAS matrix to which the unary function is applied.

5683 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
5684 should be specified. Non-default field/value pairs are listed as follows:

5685

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.

5686

5687 Return Values

5688 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5689 blocking mode, this indicates that the compatibility tests on
5690 dimensions and domains for the input arguments passed suc-
5691 cessfully. Either way, output matrix **C** is ready to be used in the
5692 next method of the sequence.

5693 **GrB_PANIC** Unknown internal error.

5694 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
5695 opaque GraphBLAS objects (input or output) is in an invalid
5696 state caused by a previous execution error. Call **GrB_error()** to
5697 access any error messages generated by the implementation.

5698 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

5699 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized
5700 by a call to **new** (or **Matrix_dup** for matrix parameters).

5701 **GrB_DIMENSION_MISMATCH** Mask and **C** dimensions are incompatible, **nrows** \neq **nrows(C)**, or
5702 **ncols** \neq **ncols(C)**.

5703 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
5704 corresponding domains of the accumulation operator or unary
5705 function, or the mask's domain is not compatible with **bool** (in
5706 the case where **desc[GrB_MASK].GrB_STRUCTURE** is not set).

5707 Description

5708 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
5709 GraphBLAS matrix: $C = f(A)$; or, if an optional binary accumulation operator (\odot) is provided,
5710 $C = C \odot f(A)$.

5711 Logically, this operation occurs in three steps:

5712 **Setup** The internal matrices and mask used in the computation are formed and their domains
5713 and dimensions are tested for compatibility.

5714 **Compute** The indicated computations are carried out.

5715 **Output** The result is written into the output matrix, possibly under control of a mask.

5716 Up to three argument matrices are used in the **GrB_apply** operation:

- 5717 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
5718 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)

5719 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5720 The argument matrices, unary operator and the accumulation operator (if provided) are tested for
5721 domain compatibility as follows:

- 5722 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\text{Mask})$
5723 must be from one of the pre-defined types of Table 3.2.
- 5724 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5725 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5726 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5727 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5728 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in}(\text{op})$ of the unary operator.

5729 Two domains are compatible with each other if values from one domain can be cast to values in
5730 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5731 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5732 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
5733 error listed above is returned.

5734 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
5735 are formed (\leftarrow denotes copy):

- 5736 1. Matrix $\tilde{C} \leftarrow C$.
- 5737 2. Two-dimensional mask, \tilde{M} , is computed from argument **Mask** as follows:
 - 5738 (a) If **Mask** = **GrB_NULL**, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
5739 $j < \mathbf{ncols}(C)\} \rangle$.
 - 5740 (b) If **Mask** \neq **GrB_NULL**,
 - 5741 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
5742 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - 5743 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
5744 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - 5745 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
- 5746 3. Matrix $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$.

5747 The internal matrices and mask are checked for dimension compatibility. The following conditions
5748 must hold:

- 5749 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.
- 5750 2. $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$.
- 5751 3. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$.

5752 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

5753 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5754 error listed above is returned.

5755 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5756 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5757 We are now ready to carry out the apply and any additional associated operations. We describe
5758 this in terms of two intermediate matrices:

- 5759 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the unary operator to the input matrix $\tilde{\mathbf{A}}$.
- 5760 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5761 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$5762 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5763 where $f = \mathbf{f}(\mathbf{op})$.

5764 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 5765 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 5766 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5767 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5768 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5769 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5770 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5771 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5772 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5773
5774
5775 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5776 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5777 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5778 mask which acts as a “write mask”.

- 5779 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5780 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5781 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 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

```
// bind-first + scalar value
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
                   const GrB_BinaryOp   accum,
                   const GrB_BinaryOp   op,
                   <type>               val,
                   const GrB_Vector     u,
                   const GrB_Descriptor desc);
```

```
// bind-first + GraphBLAS scalar
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
                   const GrB_BinaryOp   accum,
                   const GrB_BinaryOp   op,
                   const GrB_Scalar     s,
                   const GrB_Vector     u,
                   const GrB_Descriptor desc);
```

```
// bind-second + scalar value
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
```

```

5818             const GrB_BinaryOp      accum,
5819             const GrB_BinaryOp      op,
5820             const GrB_Vector        u,
5821             <type>                  val,
5822             const GrB_Descriptor    desc);

5823 // bind-second + GraphBLAS scalar
5824 GrB_Info GrB_apply(GrB_Vector        w,
5825                   const GrB_Vector    mask,
5826                   const GrB_BinaryOp  accum,
5827                   const GrB_BinaryOp  op,
5828                   const GrB_Vector    u,
5829                   const GrB_Scalar     s,
5830                   const GrB_Descriptor desc);

```

5831 Parameters

5832 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5833 that may be accumulated with the result of the apply operation. On output, this
5834 vector holds the results of the operation.

5835 **mask** (IN) An optional “write” mask that controls which results from this operation are
5836 stored into the output vector **w**. The mask dimensions must match those of the
5837 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5838 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
5839 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5840 dimensions of **w**), **GrB_NULL** should be specified.

5841 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5842 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5843 specified.

5844 **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar
5845 value, **val**.

5846 **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as
5847 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
5848 argument in the *bind-second* variant.

5849 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
5850 argument in the *bind-first* variant, or the right-hand (second) argument in the
5851 *bind-second* variant.

5852 **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand
5853 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
5854 the *bind-second* variant. It must not be empty.

5855 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
5856 should be specified. Non-default field/value pairs are listed as follows:

5857

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 .

5858

5859 Return Values

5860 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
5861 blocking mode, this indicates that the compatibility tests on di-
5862 mensions and domains for the input arguments passed successfully.
5863 Either way, output vector **w** is ready to be used in the next method
5864 of the sequence.

5865 GrB_PANIC Unknown internal error.

5866 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
5867 GraphBLAS objects (input or output) is in an invalid state caused
5868 by a previous execution error. Call GrB_error() to access any error
5869 messages generated by the implementation.

5870 GrB_OUT_OF_MEMORY Not enough memory available for operation.

5871 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
5872 a call to new (or dup for vector parameters).

5873 GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

5874 GrB_DOMAIN_MISMATCH The domains of the various vectors and scalar are incompatible with
5875 the corresponding domains of the binary operator or accumulation
5876 operator, or the mask's domain is not compatible with bool (in the
5877 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

5878 GrB_EMPTY_OBJECT The GrB_Scalar **s** used in the call is empty (**nvals(s) = 0**) and
5879 therefore a value cannot be passed to the binary operator.

5880 Description

5881 This variant of GrB_apply computes the result of applying a binary operator to the elements of a
5882 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5883 bind-first: $w = f(\text{val}, u)$ or $w = f(s, u)$

5884 bind-second: $w = f(u, \text{val})$ or $w = f(u, s)$,

5885 or if an optional binary accumulation operator (\odot) is provided:

5886 bind-first: $w = w \odot f(\text{val}, u)$ or $w = w \odot f(s, u)$

5887 bind-second: $w = w \odot f(u, \text{val})$ or $w = w \odot f(u, s)$.

5888 Logically, this operation occurs in three steps:

5889 **Setup** The internal vectors and mask used in the computation are formed and their domains
5890 and dimensions are tested for compatibility.

5891 **Compute** The indicated computations are carried out.

5892 **Output** The result is written into the output vector, possibly under control of a mask.

5893 Up to three argument vectors are used in this GrB_apply operation:

- 5894 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
5895 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
5896 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5897 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are
5898 tested for domain compatibility as follows:

- 5899 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
5900 must be from one of the pre-defined types of Table 3.2.
- 5901 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.
- 5902 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5903 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
5904 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5905 4. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.
- 5906 5. If bind-first:
- 5907 (a) $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.
- 5908 (b) If the non-opaque scalar **val** is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
5909 of the binary operator.
- 5910 (c) If the GrB_Scalar **s** is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
5911 binary operator.

- 5912 6. If bind-second:
- 5913 (a) $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the binary operator.
- 5914 (b) If the non-opaque scalar \mathbf{val} is provided, then $\mathbf{D}(\mathbf{val})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
- 5915 of the binary operator.
- 5916 (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
- 5917 binary operator.

5918 Two domains are compatible with each other if values from one domain can be cast to values in

5919 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5920 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5921 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5922 error listed above is returned.

5923 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow

5924 denotes copy):

- 5925 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5926 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 5927 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 5928 (b) If `mask \neq GrB_NULL`,
- 5929 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 5930 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 5931 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5932 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 5933 4. Scalar $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$ (GraphBLAS scalar case).

5934 The internal vectors and masks are checked for dimension compatibility. The following conditions

5935 must hold:

- 5936 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5937 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

5938 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5939 error listed above is returned.

5940 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5941 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5942 If an empty `GrB_Scalar` $\tilde{\mathbf{s}}$ is provided ($\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

5943 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

5944 `val` with the same domain as $\tilde{\mathbf{s}}$ and set `val = val($\tilde{\mathbf{s}}$)`.

5945 We are now ready to carry out the apply and any additional associated operations. We describe

5946 this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$: The vector holding the result from applying the binary 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 one of the following:

$$\begin{aligned} \text{bind-first: } \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\text{val}, \tilde{\mathbf{u}}(i))) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \\ \text{bind-second: } \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i), \text{val})) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \end{aligned}$$

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

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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 $\tilde{\mathbf{t}}$.

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

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.

5980 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

5981 Computes the transformation of the values of the stored elements of a matrix using a binary
5982 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the
5983 first argument to the binary operator and stored elements of the matrix are passed as the second
5984 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument
5985 and the specified scalar value is passed as the second argument. The scalar can be passed either as
5986 a non-opaque variable or as a GrB_Scalar object.

5987 C Syntax

```
5988 // bind-first + scalar value
5989 GrB_Info GrB_apply(GrB_Matrix      C,
5990                   const GrB_Matrix  Mask,
5991                   const GrB_BinaryOp accum,
5992                   const GrB_BinaryOp op,
5993                   <type>            val,
5994                   const GrB_Matrix  A,
5995                   const GrB_Descriptor desc);
```

```
5996 // bind-first + GraphBLAS scalar
5997 GrB_Info GrB_apply(GrB_Matrix      C,
5998                   const GrB_Matrix  Mask,
5999                   const GrB_BinaryOp accum,
6000                   const GrB_BinaryOp op,
6001                   const GrB_Scalar  s,
6002                   const GrB_Matrix  A,
6003                   const GrB_Descriptor desc);
```

```
6004 // bind-second + scalar value
6005 GrB_Info GrB_apply(GrB_Matrix      C,
6006                   const GrB_Matrix  Mask,
6007                   const GrB_BinaryOp accum,
6008                   const GrB_BinaryOp op,
6009                   const GrB_Matrix  A,
6010                   <type>            val,
6011                   const GrB_Descriptor desc);
```

```
6012 // bind-second + GraphBLAS scalar
6013 GrB_Info GrB_apply(GrB_Matrix      C,
6014                   const GrB_Matrix  Mask,
6015                   const GrB_BinaryOp accum,
6016                   const GrB_BinaryOp op,
6017                   const GrB_Matrix  A,
```

```

6018         const GrB_Scalar      s,
6019         const GrB_Descriptor desc);

```

6020 Parameters

6021 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6022 that may be accumulated with the result of the apply operation. On output, the
6023 matrix holds the results of the operation.

6024 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6025 stored into the output matrix C. The mask dimensions must match those of the
6026 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
6027 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
6028 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
6029 dimensions of C), `GrB_NULL` should be specified.

6030 **accum** (IN) An optional binary operator used for accumulating entries into existing C
6031 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
6032 specified.

6033 **op** (IN) A binary operator applied to each element of input matrix, A, with the element
6034 of the input matrix used as the left-hand argument, and the scalar value, `val`, used
6035 as the right-hand argument.

6036 **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as
6037 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
6038 argument in the *bind-second* variant.

6039 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
6040 argument in the *bind-first* variant, or the right-hand (second) argument in the
6041 *bind-second* variant.

6042 **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand
6043 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
6044 the *bind-second* variant. It must not be empty.

6045 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
6046 should be specified. Non-default field/value pairs are listed as follows:

6047

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).

6048

6049 Return Values

6050 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
6051 blocking mode, this indicates that the compatibility tests on
6052 dimensions and domains for the input arguments passed suc-
6053 cessfully. Either way, output matrix C is ready to be used in the
6054 next method of the sequence.

6055 GrB_PANIC Unknown internal error.

6056 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6057 opaque GraphBLAS objects (input or output) is in an invalid
6058 state caused by a previous execution error. Call GrB_error() to
6059 access any error messages generated by the implementation.

6060 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

6061 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6062 by a call to new (or Matrix_dup for matrix parameters).

6063 GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to nrows(A), or
6064 a value in col_indices is greater than or equal to ncols(A). In
6065 non-blocking mode, this can be reported as an execution error.

6066 GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrows \neq nrows(C), or
6067 ncols \neq ncols(C).

6068 GrB_DOMAIN_MISMATCH The domains of the various matrices and scalar are incompatible
6069 with the corresponding domains of the binary operator or accu-
6070 mulation operator, or the mask's domain is not compatible with
6071 bool (in the case where desc[GrB_MASK].GrB_STRUCTURE is
6072 not set).

6073 GrB_EMPTY_OBJECT The GrB_Scalar s used in the call is empty (nvals(s) = 0) and
6074 therefore a value cannot be passed to the binary operator.

6075 Description

6076 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a
6077 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6078 bind-first: $C = f(\text{val}, A)$ or $C = f(s, A)$

6079 bind-second: $C = f(A, \text{val})$ or $C = f(A, s)$,

6080 or if an optional binary accumulation operator (\odot) is provided:

6081 bind-first: $C = C \odot f(\text{val}, A)$ or $C = C \odot f(s, A)$

6082 bind-second: $C = C \odot f(A, \text{val})$ or $C = C \odot f(A, s)$.

6083 Logically, this operation occurs in three steps:

6084 **Setup** The internal matrices and mask used in the computation are formed and their domains
6085 and dimensions are tested for compatibility.

6086 **Compute** The indicated computations are carried out.

6087 **Output** The result is written into the output matrix, possibly under control of a mask.

6088 Up to three argument matrices are used in the `GrB_apply` operation:

- 6089 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6090 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)
- 6091 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6092 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are
6093 tested for domain compatibility as follows:

- 6094 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
6095 must be from one of the pre-defined types of Table 3.2.
- 6096 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.
- 6097 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6098 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
6099 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6100 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.
- 6101 5. If bind-first:
6102 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

6103 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
 6104 of the binary operator.

6105 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
 6106 binary operator.

6107 6. If `bind-second`:

6108 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

6109 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$
 6110 of the binary operator.

6111 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the
 6112 binary operator.

6113 Two domains are compatible with each other if values from one domain can be cast to values in
 6114 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6115 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6116 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
 6117 error listed above is returned.

6118 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
 6119 are formed (\leftarrow denotes copy):

6120 1. Matrix $\tilde{C} \leftarrow C$.

6121 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:

6122 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
 6123 $j < \mathbf{ncols}(C)\} \rangle$.

6124 (b) If `Mask` \neq `GrB_NULL`,

6125 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
 6126 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,

6127 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
 6128 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.

6129 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.

6130 3. Matrix \tilde{A} is computed from argument `A` as follows:

6131 `bind-first`: $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? A^T : A$

6132 `bind-second`: $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$

6133 4. Scalar $\tilde{s} \leftarrow s$ (`GraphBLAS` scalar case).

6134 The internal matrices and mask are checked for dimension compatibility. The following conditions
 6135 must hold:

6136 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.

6137 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.

6138 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6139 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6140 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6141 error listed above is returned.

6142 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6143 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6144 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6145 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6146 \mathbf{val} with the same domain as \tilde{s} and set $\mathbf{val} = \mathbf{val}(\tilde{s})$.

6147 We are now ready to carry out the apply and any additional associated operations. We describe
6148 this in terms of two intermediate matrices:

- 6149 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the binary operator to the input matrix $\tilde{\mathbf{A}}$.
- 6150 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6151 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as one of the following:

6152 bind-first: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$,

6153 bind-second: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$,

6154 where $f = \mathbf{f}(\mathbf{op})$.

6155 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6156 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6157 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6158 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6159 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6160 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6161 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6162 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6163 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6166 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6167 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6168 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6169 mask which acts as a “write mask”.

- 6170 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
6171 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6172 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6173 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6174 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6175 mask are unchanged:

$$6176 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6177 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
6178 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
6179 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
6180 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6181 sequence.

6182 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6183 Computes the transformation of the values of the stored elements of a vector using an index unary
6184 operator that is a function of the stored value, its location indices, and an user provided scalar
6185 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6186 C Syntax

```
6187     GrB_Info GrB_apply(GrB_Vector          w,
6188                       const GrB_Vector    mask,
6189                       const GrB_BinaryOp   accum,
6190                       const GrB_IndexUnaryOp op,
6191                       const GrB_Vector    u,
6192                       <type>              val,
6193                       const GrB_Descriptor desc);
```

```
6194     GrB_Info GrB_apply(GrB_Vector          w,
6195                       const GrB_Vector    mask,
6196                       const GrB_BinaryOp   accum,
6197                       const GrB_IndexUnaryOp op,
6198                       const GrB_Vector    u,
6199                       const GrB_Scalar    s,
6200                       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.

6232 GrB_PANIC Unknown internal error.

6233 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6234 opaque GraphBLAS objects (input or output) is in an invalid
6235 state caused by a previous execution error. Call `GrB_error()` to
6236 access any error messages generated by the implementation.

6237 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6238 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6239 by a call to `new` (or another constructor).

6240 GrB_DIMENSION_MISMATCH `mask`, `w` and/or `u` dimensions are incompatible.

6241 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
6242 responding domains of the accumulation operator or index unary
6243 operator, or the mask's domain is not compatible with `bool` (in
6244 the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6245 GrB_EMPTY_OBJECT The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and
6246 therefore a value cannot be passed to the index unary operator.

6247 Description

6248 This variant of `GrB_apply` computes the result of applying an index unary operator to the elements
6249 of a GraphBLAS vector each composed with the element's index and a scalar constant, `val` or `s`:

$$6250 \quad w = f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = f_i(u, \mathbf{ind}(u), 0, s),$$

6251 or if an optional binary accumulation operator (\odot) is provided:

$$6252 \quad w = w \odot f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = w \odot f_i(u, \mathbf{ind}(u), 0, s).$$

6253 Logically, this operation occurs in three steps:

6254 **Setup** The internal vectors and mask used in the computation are formed and their domains
6255 and dimensions are tested for compatibility.

6256 **Compute** The indicated computations are carried out.

6257 **Output** The result is written into the output vector, possibly under control of a mask.

6258 Up to three argument vectors are used in this `GrB_apply` operation:

- 6259 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6260 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

6261 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6262 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6263 are tested for domain compatibility as follows:

- 6264 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
6265 must be from one of the pre-defined types of Table 3.2.
- 6266 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the index unary operator.
- 6267 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6268 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be compatible
6269 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6270 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6271 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
6272 the index unary operator.
- 6273 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
6274 unary operator.

6275 Two domains are compatible with each other if values from one domain can be cast to values in
6276 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6277 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6278 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
6279 error listed above is returned.

6280 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6281 denotes copy):

- 6282 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6283 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 6284 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 6285 (b) If `mask \neq GrB_NULL`,
 - 6286 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,
 - 6287 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 6288 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6289 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 6290 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6291 The internal vectors and masks are checked for dimension compatibility. The following conditions
6292 must hold:

6293 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6294 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

6295 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6296 error listed above is returned.

6297 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6298 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6299 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6300 If a non-empty `GrB_Scalar`, \tilde{s} , is provided ($\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable `val`
6301 with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6302 We are now ready to carry out the apply and any additional associated operations. We describe
6303 this in terms of two intermediate vectors:

- 6304 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6305 $\tilde{\mathbf{u}}$.
- 6306 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6307 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6308 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6309 where $f_i = \mathbf{f}(\text{op})$.

6310 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6311 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 6312 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6313 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6314 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6315 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6316 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6317 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6318 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6319 \quad & \\ 6320 \end{aligned}$$

6321 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6322 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
6323 using what is called a *standard vector mask and replace*. This is carried out under control of the
6324 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.

6337 4.3.8.6 apply: Matrix index unary operator variant[Scott: NEW CONTENT]

6338 Computes the transformation of the values of the stored elements of a matrix using an index unary
6339 operator that is a function of the stored value, its location indices, and an user provided scalar
6340 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6341 C Syntax

```
6342     GrB_Info GrB_apply(GrB_Matrix      C,
6343                       const GrB_Matrix Mask,
6344                       const GrB_BinaryOp accum,
6345                       const GrB_IndexUnaryOp op,
6346                       const GrB_Matrix A,
6347                       <type>          val,
6348                       const GrB_Descriptor desc);
```

```
6349     GrB_Info GrB_apply(GrB_Matrix      C,
6350                       const GrB_Matrix Mask,
6351                       const GrB_BinaryOp accum,
6352                       const GrB_IndexUnaryOp op,
6353                       const GrB_Matrix A,
6354                       const GrB_Scalar s,
6355                       const GrB_Descriptor desc);
```

6356 Parameters

6357 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6358 that may be accumulated with the result of the apply operation. On output, the
6359 matrix holds the results of the operation.

6360 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6361 stored into the output matrix **C**. The mask dimensions must match those of the
6362 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6363 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
6364 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6365 dimensions of **C**), **GrB_NULL** should be specified.

6366 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
6367 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6368 specified.

6369 **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
6370 to each element stored in the input matrix, **A**. It is a function of the stored element’s
6371 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6372 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6373 ator.

6374 **val** (IN) An additional scalar value that is passed to the index unary operator.

6375 **s** (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6376 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6377 should be specified. Non-default field/value pairs are listed as follows:
6378

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.

6380 Return Values

6381 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
6382 blocking mode, this indicates that the compatibility tests on di-
6383 mensions and domains for the input arguments passed successfully.
6384 Either way, output matrix **C** is ready to be used in the next method
6385 of the sequence.

6386 **GrB_PANIC** Unknown internal error.

6387 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
6388 GraphBLAS objects (input or output) is in an invalid state caused

6389 by a previous execution error. Call `GrB_error()` to access any error
6390 messages generated by the implementation.

6391 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

6392 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
6393 a call to `new` (or another constructor).

6394 **GrB_DIMENSION_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6395 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
6396 corresponding domains of the accumulation operator or index unary
6397 operator, or the mask's domain is not compatible with `bool` (in the
6398 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6399 **GrB_EMPTY_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and
6400 therefore a value cannot be passed to the index unary operator.

6401 Description

6402 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements
6403 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar
6404 constant, `val` or `s`:

$$6405 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6406 or if an optional binary accumulation operator (\odot) is provided:

$$6407 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6408 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
6409 indices, respectively.

6410 Logically, this operation occurs in three steps:

6411 **Setup** The internal matrices and mask used in the computation are formed and their domains
6412 and dimensions are tested for compatibility.

6413 **Compute** The indicated computations are carried out.

6414 **Output** The result is written into the output matrix, possibly under control of a mask.

6415 Up to three argument matrices are used in the `GrB_apply` operation:

- 6416 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6417 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)

6418 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6419 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6420 are tested for domain compatibility as follows:

- 6421 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\mathbf{Mask})$
6422 must be from one of the pre-defined types of Table 3.2.
- 6423 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator.
- 6424 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6425 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be compatible
6426 with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accumulation operator.
- 6427 4. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
- 6428 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
6429 the index unary operator.
- 6430 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
6431 unary operator.

6432 Two domains are compatible with each other if values from one domain can be cast to values in
6433 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6434 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6435 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
6436 error listed above is returned.

6437 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation
6438 are formed (\leftarrow denotes copy):

- 6439 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 6440 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument **Mask** as follows:
 - 6441 (a) If **Mask** = **GrB_NULL**, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
6442 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 6443 (b) If **Mask** \neq **GrB_NULL**,
 - 6444 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
6445 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 6446 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
6447 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 6448 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 6449 3. Matrix $\tilde{\mathbf{A}}$ is computed from argument **A** as follows:
 - 6450 $\tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6451 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6452 The internal matrices and mask are checked for dimension compatibility. The following conditions
6453 must hold:

- 6454 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 6455 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 6456 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 6457 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6458 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6459 error listed above is returned.

6460 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6461 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6462 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6463 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6464 \mathbf{val} with the same domain as \tilde{s} and set $\mathbf{val} = \mathbf{val}(\tilde{s})$.

6465 We are now ready to carry out the apply and any additional associated operations. We describe
6466 this in terms of two intermediate matrices:

- 6467 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6468 $\tilde{\mathbf{A}}$.
- 6469 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6470 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6471 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6472 where $f_i = \mathbf{f}(\mathbf{op})$.

6473 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6474 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6475 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6476 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6477 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6478 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 6479 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6480 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6481 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6482 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6483 \end{aligned}$$

6484 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6485 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6486 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6487 mask which acts as a “write mask”.

- 6488 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
6489 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6490 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6491 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6492 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6493 mask are unchanged:

$$6494 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6495 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
6496 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
6497 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
6498 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6499 sequence.

6500 4.3.9 select:

6501 Apply a select operator to the stored elements of an object to determine whether or not to keep
6502 them.

6503 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6504 Apply a select operator (an index unary operator) to the elements of a vector.

6505 C Syntax

```
6506 // scalar value variant
6507 GrB_Info GrB_select(GrB_Vector          w,
6508                    const GrB_Vector      mask,
6509                    const GrB_BinaryOp    accum,
6510                    const GrB_IndexUnaryOp op,
6511                    const GrB_Vector      u,
6512                    <type>                val,
6513                    const GrB_Descriptor   desc);
6514
6515 // GraphBLAS scalar variant
6516 GrB_Info GrB_select(GrB_Vector          w,
6517                    const GrB_Vector      mask,
```

```

6518         const GrB_BinaryOp      accum,
6519         const GrB_IndexUnaryOp  op,
6520         const GrB_Vector        u,
6521         const GrB_Scalar        s,
6522         const GrB_Descriptor    desc);
6523

```

6524 Parameters

6525 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6526 that may be accumulated with the result of the select operation. On output, this
6527 vector holds the results of the operation.

6528 **mask** (IN) An optional “write” mask that controls which results from this operation are
6529 stored into the output vector **w**. The mask dimensions must match those of the
6530 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6531 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6532 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6533 dimensions of **w**), **GrB_NULL** should be specified.

6534 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6535 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6536 specified.

6537 **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
6538 to each element stored in the input vector, **u**. It is a function of the stored element’s
6539 value, its location index, and a user supplied scalar value (either **s** or **val**).

6540 **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-
6541 ator.

6542 **val** (IN) An additional scalar value that is passed to the index unary operator.

6543 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6544 not be empty.

6545 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6546 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 .

6548

6549 Return Values

6550 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 6551 blocking mode, this indicates that the compatibility tests on di-
 6552 mensions and domains for the input arguments passed success-
 6553 fully. Either way, output vector **w** is ready to be used in the next
 6554 method of the sequence.

6555 GrB_PANIC Unknown internal error.

6556 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 6557 opaque GraphBLAS objects (input or output) is in an invalid
 6558 state caused by a previous execution error. Call **GrB_error()** to
 6559 access any error messages generated by the implementation.

6560 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6561 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
 6562 by a call to one of its constructors.

6563 GrB_DIMENSION_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

6564 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 6565 responding domains of the accumulation operator or index unary
 6566 operator, or the **mask**'s domain is not compatible with **bool** (in
 6567 the case where **desc[GrB_MASK].GrB_STRUCTURE** is not set).

6568 GrB_EMPTY_OBJECT The **GrB_Scalar s** used in the call is empty (**nvals(s) = 0**) and
 6569 therefore a value cannot be passed to the index unary operator.

6570 Description

6571 This variant of **GrB_select** computes the result of applying a index unary operator to select the
 6572 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored
 6573 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding
 6574 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.
 6575 This acts like a functional mask on the input vector as follows:

$$6576 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6577 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6578 Correspondingly, if a **GrB_Scalar s**, is provided:

$$6579 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6580 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6581 Logically, this operation occurs in three steps:

6582 **Setup** The internal vectors and mask used in the computation are formed and their domains
6583 and dimensions are tested for compatibility.

6584 **Compute** The indicated computations are carried out.

6585 **Output** The result is written into the output vector, possibly under control of a mask.

6586 Up to three argument vectors are used in this GrB_select operation:

- 6587 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
6588 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
6589 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6590 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6591 are tested for domain compatibility as follows:

- 6592 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\mathbf{mask})$
6593 must be from one of the pre-defined types of Table 3.2.
6594 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
6595 3. If **accum** is not GrB_NULL, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6596 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
6597 mulation operator.
6598 4. $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
6599 i.e., castable to **bool**.
6600 5. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
6601 6. $\mathbf{D}(\mathbf{val})$ or $\mathbf{D}(\mathbf{s})$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
6602 of the index unary operator.

6603 Two domains are compatible with each other if values from one domain can be cast to values in
6604 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6605 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6606 any compatibility rule above is violated, execution of GrB_select ends and the domain mismatch
6607 error listed above is returned.

6608 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6609 denotes copy):

- 6610 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
6611 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument **mask** as follows:

6612 (a) If $\text{mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
6613 (b) If $\text{mask} \neq \text{GrB_NULL}$,
6614 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
6615 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
6616 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
6617 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
6618 4. Scalar $\widetilde{s} \leftarrow s$ (GrB_Scalar version only).

6619 The internal vectors and masks are checked for dimension compatibility. The following conditions
6620 must hold:

- 6621 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6622 2. $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$.

6623 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch
6624 error listed above is returned.

6625 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6626 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6627 If an empty `GrB_Scalar` \widetilde{s} is provided (i.e., $\text{nvals}(\widetilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6628 If a non-empty `GrB_Scalar`, \widetilde{s} , is provided (i.e., $\text{nvals}(\widetilde{s}) = 1$), we then create an internal variable
6629 `val` with the same domain as \widetilde{s} and set $\text{val} = \text{val}(\widetilde{s})$.

6630 We are now ready to carry out the `select` and any additional associated operations. We describe
6631 this in terms of two intermediate vectors:

- 6632 • $\widetilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6633 $\widetilde{\mathbf{u}}$.
- 6634 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6635 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$6636 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6637 where $f_i = \mathbf{f}(\text{op})$.

6638 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6639 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 6640 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$6641 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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);
```

```

6677 // GraphBLAS scalar variant
6678 GrB_Info GrB_select(GrB_Matrix          C,
6679                    const GrB_Matrix     Mask,
6680                    const GrB_BinaryOp    accum,
6681                    const GrB_IndexUnaryOp op,
6682                    const GrB_Matrix      A,
6683                    const GrB_Scalar      s,
6684                    const GrB_Descriptor   desc);

```

6685 Parameters

6686 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6687 that may be accumulated with the result of the select operation. On output, the
6688 matrix holds the results of the operation.

6689 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6690 stored into the output matrix **C**. The mask dimensions must match those of the
6691 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6692 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
6693 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6694 dimensions of **C**), **GrB_NULL** should be specified.

6695 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
6696 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6697 specified.

6698 **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
6699 to each element stored in the input matrix, **A**. It is a function of the stored element’s
6700 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6701 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6702 ator.

6703 **val** (IN) An additional scalar value that is passed to the index unary operator.

6704 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6705 not be empty.

6706 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6707 should be specified. Non-default field/value pairs are listed as follows:
6708

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:

6738 $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$, or
6739 $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$.

6740 Correspondingly, if a GrB_Scalar, s, is provided:

6741 $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$, or
6742 $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$.

6743 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
6744 indices, respectively.

6745 Logically, this operation occurs in three steps:

6746 **Setup** The internal matrices and mask used in the computation are formed and their domains
6747 and dimensions are tested for compatibility.

6748 **Compute** The indicated computations are carried out.

6749 **Output** The result is written into the output matrix, possibly under control of a mask.

6750 Up to three argument matrices are used in the GrB_select operation:

- 6751 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6752 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)
- 6753 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6754 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6755 are tested for domain compatibility as follows:

- 6756 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
6757 must be from one of the pre-defined types of Table 3.2.
- 6758 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 6759 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6760 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
6761 mulation operator.
- 6762 4. $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
6763 i.e., castable to bool.
- 6764 5. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6765 6. $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\text{op})$
6766 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:

- 6801 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6802 $\tilde{\mathbf{A}}$.
- 6803 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6804 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6805 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6806 where $f_i = \mathbf{f}(\text{op})$.

6807 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6808 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6809 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6810 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

6811 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6812 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6813 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6814 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6815 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6816 \quad 6817$$

6818 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6819 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6820 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6821 mask which acts as a “write mask”.

- 6822 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
6823 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6824 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6825 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6826 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6827 mask are unchanged:

$$6828 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6829 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6830 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6831 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
6832 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6833 sequence.

6834 4.3.10 reduce: Perform a reduction across the elements of an object

6835 Computes the reduction of the values of the elements of a vector or matrix.

6836 4.3.10.1 reduce: Standard matrix to vector variant

6837 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns
6838 is desired, the input matrix should be transposed using the descriptor.

6839 C Syntax

```
6840     GrB_Info GrB_reduce(GrB_Vector      w,  
6841                        const GrB_Vector mask,  
6842                        const GrB_BinaryOp accum,  
6843                        const GrB_Monoid op,  
6844                        const GrB_Matrix A,  
6845                        const GrB_Descriptor desc);  
6846  
6847     GrB_Info GrB_reduce(GrB_Vector      w,  
6848                        const GrB_Vector mask,  
6849                        const GrB_BinaryOp accum,  
6850                        const GrB_BinaryOp op,  
6851                        const GrB_Matrix A,  
6852                        const GrB_Descriptor desc);
```

6853 Parameters

6854 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6855 that may be accumulated with the result of the reduction operation. On output,
6856 this vector holds the results of the operation.

6857 **mask** (IN) An optional “write” mask that controls which results from this operation are
6858 stored into the output vector **w**. The mask dimensions must match those of the
6859 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6860 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6861 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6862 dimensions of **w**), **GrB_NULL** should be specified.

6863 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6864 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6865 specified.

6866 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.
6867 Depending on which type is passed, the following defines the binary operator with
6868 one domain, $F_b = \langle D, D, D, \oplus \rangle$, that is used:

6869 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
6870 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
6871 monoid is ignored.

6872 If op is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in
6873 both cases $\odot(\text{op})$ must be commutative and associative. Otherwise, the outcome
6874 of the operation is undefined.

6875 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

6876 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
6877 should be specified. Non-default field/value pairs are listed as follows:
6878

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.

6880 Return Values

6881 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
6882 blocking mode, this indicates that the compatibility tests on di-
6883 mensions and domains for the input arguments passed successfully.
6884 Either way, output vector w is ready to be used in the next method
6885 of the sequence.

6886 **GrB_PANIC** Unknown internal error.

6887 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
6888 GraphBLAS objects (input or output) is in an invalid state caused
6889 by a previous execution error. Call `GrB_error()` to access any error
6890 messages generated by the implementation.

6891 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

6892 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
6893 a call to `new` (or `dup` for vector parameters).

6894 **GrB_DIMENSION_MISMATCH** mask, w and/or u dimensions are incompatible.

6895 **GrB_DOMAIN_MISMATCH** Either the domains of the various vectors and matrices are incom-
6896 patible with the corresponding domains of the accumulation oper-
6897 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).

6901 Description

6902 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows
 6903 of an input matrix: $w(i) = \bigoplus A(i, :) \forall i$; or, if an optional binary accumulation operator is provided,
 6904 $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$, where $\bigoplus = \odot(F_b)$ and $\odot = \odot(\text{accum})$.

6905 Logically, this operation occurs in three steps:

6906 **Setup** The internal vector, matrix and mask used in the computation are formed and their
 6907 domains and dimensions are tested for compatibility.

6908 **Compute** The indicated computations are carried out.

6909 **Output** The result is written into the output vector, possibly under control of a mask.

6910 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6911 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6912 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 6913 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6914 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested
 6915 for domain compatibility as follows:

- 6916 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 6917 must be from one of the pre-defined types of Table 3.2.
- 6918 2. $\mathbf{D}(w)$ must be compatible with the domain of the reduction binary operator, $\mathbf{D}(F_b)$.
- 6919 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6920 of the accumulation operator and $\mathbf{D}(F_b)$, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 6921 mulation operator.
- 6922 4. $\mathbf{D}(A)$ must be compatible with the domain of the binary reduction operator, $\mathbf{D}(F_b)$.

6923 Two domains are compatible with each other if values from one domain can be cast to values in
 6924 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6925 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6926 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 6927 error listed above is returned.

6928 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 6929 denotes copy):

- 6930 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6931 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 6932 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 6933 (b) If `mask \neq GrB_NULL`,
- 6934 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 6935 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 6936 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6937 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

6938 The internal vectors and masks are checked for dimension compatibility. The following conditions
6939 must hold:

- 6940 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6941 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6942 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-
6943 match error listed above is returned.

6944 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6945 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6946 We carry out the reduce and any additional associated operations. We describe this in terms of
6947 two intermediate vectors:

- 6948 • $\tilde{\mathbf{t}}$: The vector holding the result from reducing along the rows of input matrix $\tilde{\mathbf{A}}$.
- 6949 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6950 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6951 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6952 The value of each of its elements is computed by

$$6953 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6954 where $\bigoplus = \odot(F_b)$.

6955 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6956 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

6957 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6958 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6959 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6960 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6961 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 6962 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6963 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6964 \quad & \\ 6965 \end{aligned}$$

6966 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6967 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
6968 using what is called a *standard vector mask and replace*. This is carried out under control of the
6969 mask which acts as a “write mask”.

6970 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
6971 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$6972 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6973 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
6974 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
6975 mask are unchanged:

$$6976 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6977 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6978 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6979 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
6980 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6981 sequence.

6982 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

6983 Reduce all stored values into a single scalar.

6984 C Syntax

```
6985 // scalar value + monoid (only)
6986 GrB_Info GrB_reduce(<type>          *val,
6987                      const GrB_BinaryOp accum,
6988                      const GrB_Monoid  op,
6989                      const GrB_Vector  u,
```

```

6990             const GrB_Descriptor desc);
6991
6992 // GraphBLAS Scalar + monoid
6993 GrB_Info GrB_reduce(GrB_Scalar      s,
6994                   const GrB_BinaryOp accum,
6995                   const GrB_Monoid   op,
6996                   const GrB_Vector   u,
6997                   const GrB_Descriptor desc);
6998
6999 // GraphBLAS Scalar + binary operator
7000 GrB_Info GrB_reduce(GrB_Scalar      s,
7001                   const GrB_BinaryOp accum,
7002                   const GrB_BinaryOp op,
7003                   const GrB_Vector   u,
7004                   const GrB_Descriptor desc);

```

7005 Parameters

7006 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
7007 a value that may be accumulated (optionally) with the result of the reduction
7008 operation. On output, this scalar holds the results of the operation.

7009 **accum** (IN) An optional binary operator used for accumulating entries into an exist-
7010 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,
7011 **GrB_NULL** should be specified.

7012 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
7013 the reduction operation. The \oplus operator must be commutative and associative;
7014 otherwise, the outcome of the operation is undefined.

7015 **u** (IN) The GraphBLAS vector on which reduction will be performed.

7016 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
7017 should be specified. Non-default field/value pairs are listed as follows:

7019 Param	Field	Value	Description
------------	-------	-------	-------------

7020 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
7021 tions. There are currently no non-default field/value pairs that can be set for this
7022 operation.

7023 Return Values

7024 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
7025 cessfully, and the output scalar (**s** or **val**) is ready to be used in the
7026 next method of the sequence.

7027	GrB_PANIC	Unknown internal error.
7028	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque
7029		GraphBLAS objects (input or output) is in an invalid state caused
7030		by a previous execution error. Call GrB_error() to access any error
7031		messages generated by the implementation.
7032	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
7033	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized by
7034		a call to a respective constructor.
7035	GrB_NULL_POINTER	val pointer is NULL.
7036	GrB_DOMAIN_MISMATCH	The domains of input and output arguments are incompatible with
7037		the corresponding domains of the accumulation operator, or reduce
7038		operator.

7039 Description

This variant of **GrB_reduce** computes the result of performing a reduction across all of the stored elements of an input vector storing the result into either **s** or **val**. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[\bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7040 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7041 Logically, this operation occurs in three steps:

7042 **Setup** The internal vector used in the computation is formed and its domain is tested for
7043 compatibility.

7044 **Compute** The indicated computations are carried out.

7045 **Output** The result is written into the output scalar.

7046 One vector argument is used in this **GrB_reduce** operation:

- 7047 1. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7048 The output scalar, argument vector, reduction operator and accumulation operator (if provided)
7049 are tested for domain compatibility as follows:

- 7050 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
7051 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

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 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

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).

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_reduce` ends and the domain mismatch error listed above is returned.

The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one of the following occurs depending on the output variant:

$$\mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

or

$$\text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

For all other cases, the internal vector and scalar used in the computation is formed (\leftarrow denotes copy):

1. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

2. Scalar $\tilde{s} \leftarrow \text{s}$ (GraphBLAS scalar case).

We are now ready to carry out the reduction and any additional associated operations. An intermediate scalar result t is computed as follows:

$$t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

where $\oplus = \odot(\text{op})$.

The final reduction value is computed as follows:

$$\mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

or

$$\text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7079 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
7080 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7081 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7082 Reduce all stored values into a single scalar.

7083 C Syntax

```
7084 // scalar value + monoid (only)
7085 GrB_Info GrB_reduce(<type>          *val,
7086                    const GrB_BinaryOp accum,
7087                    const GrB_Monoid  op,
7088                    const GrB_Matrix  A,
7089                    const GrB_Descriptor desc);
7090
7091 // GraphBLAS Scalar + monoid
7092 GrB_Info GrB_reduce(GrB_Scalar      s,
7093                    const GrB_BinaryOp accum,
7094                    const GrB_Monoid  op,
7095                    const GrB_Matrix  A,
7096                    const GrB_Descriptor desc);
7097
7098 // GraphBLAS Scalar + binary operator
7099 GrB_Info GrB_reduce(GrB_Scalar      s,
7100                    const GrB_BinaryOp accum,
7101                    const GrB_BinaryOp op,
7102                    const GrB_Matrix  A,
7103                    const GrB_Descriptor desc);
```

7104 Parameters

7105 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
7106 a value that may be accumulated (optionally) with the result of the reduction
7107 operation. On output, this scalar holds the results of the operation.

7108 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or
7109 **val**) value. If assignment rather than accumulation is desired, GrB_NULL should
7110 be specified.

7111 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
7112 the reduction operation. The \oplus operator must be commutative and associative;
7113 otherwise, the outcome of the operation is undefined.

7114 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

7115 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 7116 should be specified. Non-default field/value pairs are listed as follows:
 7117

7118	Param	Field	Value	Description
------	-------	-------	-------	-------------

7119 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
 7120 tions. There are currently no non-default field/value pairs that can be set for this
 7121 operation.

7122 Return Values

7123 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
 7124 cessfully, and the output scalar (s or val) is ready to be used in the
 7125 next method of the sequence.

7126 GrB_PANIC Unknown internal error.

7127 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 7128 GraphBLAS objects (input or output) is in an invalid state caused
 7129 by a previous execution error. Call GrB_error() to access any error
 7130 messages generated by the implementation.

7131 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7132 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 7133 a call to a respective constructor.

7134 GrB_NULL_POINTER val pointer is NULL.

7135 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
 7136 the corresponding domains of the accumulation operator, or reduce
 7137 operator.

7138 Description

This variant of GrB_reduce computes the result of performing a reduction across all of the stored elements of an input matrix storing the result into either s or val. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[\bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7139 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7140 Logically, this operation occurs in three steps:

7141 **Setup** The internal matrix used in the computation is formed and its domain is tested for
 7142 compatibility.

7143 **Compute** The indicated computations are carried out.

7144 **Output** The result is written into the output scalar.

7145 One matrix argument is used in this GrB_reduce operation:

7146 1. $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7147 The output scalar, argument matrix, reduction operator and accumulation operator (if provided)
 7148 are tested for domain compatibility as follows:

7149 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
 7150 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

7151 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
 7152 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 7153 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

7154 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).

7155 Two domains are compatible with each other if values from one domain can be cast to values in
 7156 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7157 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7158 any compatibility rule above is violated, execution of GrB_reduce ends and the domain mismatch
 7159 error listed above is returned.

7160 The number of values stored in the input, A , is checked. If there are no stored values in A , then
 7161 one of the following occurs depending on the output variant:

$$7162 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7163 or

$$7164 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7165 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with GrB_SUCCESS.

7166 For all other cases, the internal matrix and scalar used in the computation is formed (\leftarrow denotes
 7167 copy):

7168 1. Matrix $\tilde{A} \leftarrow A$.

7169 2. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

7170 We are now ready to carry out the reduce and any additional associated operations. An intermediate
 7171 scalar result t is computed as follows:

$$7172 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7173 where $\oplus = \odot(\text{op})$.

7174 The final reduction value is computed as follows:

$$7175 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7176 or

$$7177 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7178 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
 7179 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7180 4.3.11 transpose: Transpose rows and columns of a matrix

7181 This version computes a new matrix that is the transpose of the source matrix.

7182 C Syntax

```
7183      GrB_Info GrB_transpose(GrB_Matrix      C,
7184                           const GrB_Matrix Mask,
7185                           const GrB_BinaryOp accum,
7186                           const GrB_Matrix A,
7187                           const GrB_Descriptor desc);
```

7188 Parameters

7189 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 7190 that may be accumulated with the result of the transpose operation. On output,
 7191 the matrix holds the results of the operation.

7192 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 7193 stored into the output matrix C. The mask dimensions must match those of the
 7194 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 7195 of the Mask matrix must be of type bool or any of the predefined “built-in” types
 7196 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
 7197 dimensions of C), GrB_NULL should be specified.

7198 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
7199 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
7200 specified.

7201 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7202 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
7203 should be specified. Non-default field/value pairs are listed as follows:
7204

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.

7206 **Return Values**

7207 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
7208 blocking mode, this indicates that the compatibility tests on di-
7209 mensions and domains for the input arguments passed successfully.
7210 Either way, output matrix **C** is ready to be used in the next method
7211 of the sequence.

7212 **GrB_PANIC** Unknown internal error.

7213 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
7214 GraphBLAS objects (input or output) is in an invalid state caused
7215 by a previous execution error. Call **GrB_error()** to access any error
7216 messages generated by the implementation.

7217 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

7218 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
7219 a call to **new** (or **Matrix_dup** for matrix parameters).

7220 **GrB_DIMENSION_MISMATCH** **mask**, **C** and/or **A** dimensions are incompatible.

7221 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the cor-
7222 responding domains of the accumulation operator, or the mask's do-
7223 main is not compatible with **bool** (in the case where **desc[GrB_MASK].GrB_STRUCT**
7224 is not set).

7225 Description

7226 GrB_transpose computes the result of performing a transpose of the input matrix: $C = A^T$; or, if an
 7227 optional binary accumulation operator (\odot) is provided, $C = C \odot A^T$. We note that the input matrix
 7228 A can itself be optionally transposed before the operation, which would cause either an assignment
 7229 from A to C or an accumulation of A into C.

7230 Logically, this operation occurs in three steps:

7231 **Setup** The internal matrix and mask used in the computation are formed and their domains
 7232 and dimensions are tested for compatibility.

7233 **Compute** The indicated computations are carried out.

7234 **Output** The result is written into the output matrix, possibly under control of a mask.

7235 Up to three matrix arguments are used in this GrB_transpose operation:

- 7236 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7237 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)
- 7238 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7239 The argument matrices and accumulation operator (if provided) are tested for domain compatibility
 7240 as follows:

- 7241 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
 7242 must be from one of the pre-defined types of Table 3.2.
- 7243 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$ of the input matrix.
- 7244 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 7245 of the accumulation operator and $\mathbf{D}(A)$ of the input matrix must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 7246 of the accumulation operator.

7247 Two domains are compatible with each other if values from one domain can be cast to values in
 7248 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7249 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7250 any compatibility rule above is violated, execution of GrB_transpose ends and the domain mismatch
 7251 error listed above is returned.

7252 From the argument matrices, the internal matrices and mask used in the computation are formed
 7253 (\leftarrow denotes copy):

- 7254 1. Matrix $\tilde{C} \leftarrow C$.
- 7255 2. Two-dimensional mask, \tilde{M} , is computed from argument Mask as follows:

- 7256 (a) If $\text{Mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
7257 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 7258 (b) If $\text{Mask} \neq \text{GrB_NULL}$,
- 7259 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
7260 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
- 7261 ii. Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
7262 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
- 7263 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$.
- 7264 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

7265 The internal matrices and masks are checked for dimension compatibility. The following conditions
7266 must hold:

- 7267 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
- 7268 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
- 7269 3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.
- 7270 4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

7271 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension
7272 mismatch error listed above is returned.

7273 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
7274 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7275 We are now ready to carry out the matrix transposition and any additional associated operations.
7276 We describe this in terms of two intermediate matrices:

- 7277 • $\widetilde{\mathbf{T}}$: The matrix holding the transpose of $\widetilde{\mathbf{A}}$.
- 7278 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

7279 The intermediate matrix

$$7280 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7281 is created.

7282 The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7283 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- 7284 • If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$7285 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7286 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 7287 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned}
 7288 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 7289 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7290 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7291 \quad & \\
 7292 \quad &
 \end{aligned}$$

7293 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7294 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 7295 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 7296 mask which acts as a “write mask”.

- 7297 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 7298 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7299 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7300 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 7301 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 7302 mask are unchanged:

$$7303 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7304 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 7305 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 7306 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 7307 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 7308 sequence.

7309 4.3.12 kronecker: Kronecker product of two matrices

7310 Computes the Kronecker product of two matrices. The result is a matrix.

7311 C Syntax

```

7312      GrB_Info GrB_kronecker(GrB_Matrix      C,
7313                             const GrB_Matrix  Mask,
7314                             const GrB_BinaryOp accum,
7315                             const GrB_Semiring op,
7316                             const GrB_Matrix  A,
7317                             const GrB_Matrix  B,
7318                             const GrB_Descriptor desc);
7319

```

```

7320     GrB_Info GrB_kronecker(GrB_Matrix      C,
7321                           const GrB_Matrix Mask,
7322                           const GrB_BinaryOp accum,
7323                           const GrB_Monoid  op,
7324                           const GrB_Matrix A,
7325                           const GrB_Matrix B,
7326                           const GrB_Descriptor desc);
7327
7328     GrB_Info GrB_kronecker(GrB_Matrix      C,
7329                           const GrB_Matrix Mask,
7330                           const GrB_BinaryOp accum,
7331                           const GrB_BinaryOp op,
7332                           const GrB_Matrix A,
7333                           const GrB_Matrix B,
7334                           const GrB_Descriptor desc);

```

7335 Parameters

7336 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
7337 that may be accumulated with the result of the Kronecker product. On output,
7338 the matrix holds the results of the operation.

7339 **Mask** (IN) An optional “write” mask that controls which results from this operation are
7340 stored into the output matrix C. The mask dimensions must match those of the
7341 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
7342 of the Mask matrix must be of type bool or any of the predefined “built-in” types
7343 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
7344 dimensions of C), GrB_NULL should be specified.

7345 **accum** (IN) An optional binary operator used for accumulating entries into existing C
7346 entries. If assignment rather than accumulation is desired, GrB_NULL should be
7347 specified.

7348 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
7349 operation. Depending on which type is passed, the following defines the binary
7350 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

7351 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

7352 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-
7353 nored.

7354 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
7355 is ignored.

7356 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
7357 product.

7358 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
7359 product.

7360 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
7361 should be specified. Non-default field/value pairs are listed as follows:
7362

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.

7364 Return Values

7365 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
7366 blocking mode, this indicates that the compatibility tests on di-
7367 mensions and domains for the input arguments passed successfully.
7368 Either way, output matrix C is ready to be used in the next method
7369 of the sequence.

7370 GrB_PANIC Unknown internal error.

7371 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
7372 GraphBLAS objects (input or output) is in an invalid state caused
7373 by a previous execution error. Call GrB_error() to access any error
7374 messages generated by the implementation.

7375 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7376 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
7377 a call to new (or Matrix_dup for matrix parameters).

7378 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

7379 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
7380 corresponding domains of the binary operator (op) or accumulation
7381 operator, or the mask's domain is not compatible with bool (in the
7382 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

7383 Description

7384 GrB_kronecker computes the Kronecker product $C = A \otimes B$ or, if an optional binary accumulation
7385 operator (\odot) is provided, $C = C \odot (A \otimes B)$ (where matrices A and B can be optionally transposed).

7386 The Kronecker product is defined as follows:

7387

$$7388 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

7389 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
7390 Kronecker product are defined as

$$7391 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7392 where \otimes is the multiplicative operator specified by the **op** parameter.

7393 Logically, this operation occurs in three steps:

7394 **Setup** The internal matrices and mask used in the computation are formed and their domains
7395 and dimensions are tested for compatibility.

7396 **Compute** The indicated computations are carried out.

7397 **Output** The result is written into the output matrix, possibly under control of a mask.

7398 Up to four argument matrices are used in the **GrB_kronecker** operation:

- 7399 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7400 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)
- 7401 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 7402 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

7403 The argument matrices, the "product" operator (**op**), and the accumulation operator (if provided)
7404 are tested for domain compatibility as follows:

- 7405 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\text{Mask})$
7406 must be from one of the pre-defined types of Table 3.2.
- 7407 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 7408 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 7409 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 7410 5. If **accum** is not **GrB_NULL**, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
7411 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of **op** must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
7412 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.

7446 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$
7447 $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$ is created. The value of
7448 each of its elements is computed by

$$7449 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7450 where \otimes is the multiplicative operator specified by the `op` parameter.

7451 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7452 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 7453 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$7454 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7455 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
7456 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$7457 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7458 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7460 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7462 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7463 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
7464 using what is called a *standard matrix mask and replace*. This is carried out under control of the
7465 mask which acts as a “write mask”.

- 7466 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
7467 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7468 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7469 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
7470 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
7471 mask are unchanged:

$$7472 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7473 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
7474 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
7475 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
7476 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7477 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
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)	GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)	GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)	GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)	GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)	GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)	GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)	GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)	GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)	GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)	GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)	GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)	GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)

Table 5.11: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_reduce(GrB_Vector,...,GrB_Monoid,...)	GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Vector,...)	GrB_Vector_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Vector,...)	GrB_Vector_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(bool*,...,GrB_Vector,...)	GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)
GrB_reduce(int8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)
GrB_reduce(uint8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)
GrB_reduce(int16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)
GrB_reduce(uint16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)
GrB_reduce(int32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)
GrB_reduce(uint32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)
GrB_reduce(int64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)
GrB_reduce(uint64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)
GrB_reduce(float*,...,GrB_Vector,...)	GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)
GrB_reduce(double*,...,GrB_Vector,...)	GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)
GrB_reduce(<i>other</i> *,...,GrB_Vector,...)	GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)	GrB_Matrix_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)	GrB_Matrix_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)
GrB_reduce(bool*,...,GrB_Matrix,...)	GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)
GrB_reduce(int8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)
GrB_reduce(uint8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)
GrB_reduce(int16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)
GrB_reduce(uint16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)
GrB_reduce(int32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)
GrB_reduce(uint32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)
GrB_reduce(int64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)
GrB_reduce(uint64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)
GrB_reduce(float*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)
GrB_reduce(double*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)
GrB_reduce(<i>other</i> *,...,GrB_Matrix,...)	GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)
GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)

Table 5.12: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_get(GrB_Scalar,...,GrB_Scalar)	GrB_Scalar_get_Scalar(GrB_Scalar,...,GrB_Scalar)
GrB_Scalar_get(GrB_Scalar,...,char*)	GrB_Scalar_get_String(GrB_Scalar,...,char*)
GrB_Scalar_get(GrB_Scalar,...,GrB_Type*)	GrB_Scalar_get_Type(GrB_Scalar,...,GrB_Type*)
GrB_Scalar_get(GrB_Scalar,...,void*)	GrB_Scalar_get_VOID(GrB_Scalar,...,void*)
GrB_Vector_get(GrB_Vector,...,GrB_Scalar)	GrB_Vector_get_Scalar(GrB_Vector,...,GrB_Scalar)
GrB_Vector_get(GrB_Vector,...,char*)	GrB_Vector_get_String(GrB_Vector,...,char*)
GrB_Vector_get(GrB_Vector,...,int*)	GrB_Matrix_get_INT32(GrB_Vector,...,int*)
GrB_Vector_get(GrB_Vector,...,GrB_Type*)	GrB_Vector_get_Type(GrB_Vector,...,GrB_Type*)
GrB_Vector_get(GrB_Vector,...,void*)	GrB_Vector_get_VOID(GrB_Vector,...,void*)
GrB_Matrix_get(GrB_Matrix,...,GrB_Scalar)	GrB_Matrix_get_Scalar(GrB_Matrix,...,GrB_Scalar)
GrB_Matrix_get(GrB_Matrix,...,char*)	GrB_Matrix_get_String(GrB_Matrix,...,char*)
GrB_Matrix_get(GrB_Matrix,...,GrB_Type*)	GrB_Matrix_get_Type(GrB_Matrix,...,GrB_Type*)
GrB_Matrix_get(GrB_Matrix,...,void*)	GrB_Matrix_get_VOID(GrB_Matrix,...,void*)
GrB_UnaryOp_get(GrB_UnaryOp,...,GrB_Scalar)	GrB_UnaryOp_get_Scalar(GrB_UnaryOp,...,GrB_Scalar)
GrB_UnaryOp_get(GrB_UnaryOp,...,char*)	GrB_UnaryOp_get_String(GrB_UnaryOp,...,char*)
GrB_UnaryOp_get(GrB_UnaryOp,...,GrB_Type*)	GrB_UnaryOp_get_Type(GrB_UnaryOp,...,GrB_Type*)
GrB_UnaryOp_get(GrB_UnaryOp,...,void*)	GrB_UnaryOp_get_VOID(GrB_UnaryOp,...,void*)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,GrB_Scalar)	GrB_IndexUnaryOp_get_Scalar(GrB_IndexUnaryOp,...,GrB_Scalar)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,char*)	GrB_IndexUnaryOp_get_String(GrB_IndexUnaryOp,...,char*)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,GrB_Type*)	GrB_IndexUnaryOp_get_Type(GrB_IndexUnaryOp,...,GrB_Type*)
GrB_IndexUnaryOp_get(GrB_IndexUnaryOp,...,void*)	GrB_IndexUnaryOp_get_VOID(GrB_IndexUnaryOp,...,void*)
GrB_BinaryOp_get(GrB_BinaryOp,...,GrB_Scalar)	GrB_BinaryOp_get_Scalar(GrB_BinaryOp,...,GrB_Scalar)
GrB_BinaryOp_get(GrB_BinaryOp,...,char*)	GrB_BinaryOp_get_String(GrB_BinaryOp,...,char*)
GrB_BinaryOp_get(GrB_BinaryOp,...,GrB_Type*)	GrB_BinaryOp_get_Type(GrB_BinaryOp,...,GrB_Type*)
GrB_BinaryOp_get(GrB_BinaryOp,...,void*)	GrB_BinaryOp_get_VOID(GrB_BinaryOp,...,void*)
GrB_Monoid_get(GrB_Monoid,...,GrB_Scalar)	GrB_Monoid_get_Scalar(GrB_Monoid,...,GrB_Scalar)
GrB_Monoid_get(GrB_Monoid,...,char*)	GrB_Monoid_get_String(GrB_Monoid,...,char*)
GrB_Monoid_get(GrB_Monoid,...,GrB_Type*)	GrB_Monoid_get_Type(GrB_Monoid,...,GrB_Type*)
GrB_Monoid_get(GrB_Monoid,...,void*)	GrB_Monoid_get_VOID(GrB_Monoid,...,void*)
GrB_Semiring_get(GrB_Semiring,...,GrB_Scalar)	GrB_Semiring_get_Scalar(GrB_Semiring,...,GrB_Scalar)
GrB_Semiring_get(GrB_Semiring,...,char*)	GrB_Semiring_get_String(GrB_Semiring,...,char*)
GrB_Semiring_get(GrB_Semiring,...,GrB_Type*)	GrB_Semiring_get_Type(GrB_Semiring,...,GrB_Type*)
GrB_Semiring_get(GrB_Semiring,...,void*)	GrB_Semiring_get_VOID(GrB_Semiring,...,void*)
GrB_Descriptor_get(GrB_Descriptor,...,GrB_Scalar)	GrB_Descriptor_get_Scalar(GrB_Descriptor,...,GrB_Scalar)
GrB_Descriptor_get(GrB_Descriptor,...,char*)	GrB_Descriptor_get_String(GrB_Descriptor,...,char*)
GrB_Descriptor_get(GrB_Descriptor,...,GrB_Type*)	GrB_Descriptor_get_Type(GrB_Descriptor,...,GrB_Type*)
GrB_Descriptor_get(GrB_Descriptor,...,void*)	GrB_Descriptor_get_VOID(GrB_Descriptor,...,void*)
GrB_Type_get(GrB_Type,...,GrB_Scalar)	GrB_Type_get_Scalar(GrB_Type,...,GrB_Scalar)
GrB_Type_get(GrB_Type,...,char*)	GrB_Type_get_String(GrB_Type,...,char*)
GrB_Type_get(GrB_Type,...,GrB_Type*)	GrB_Type_get_Type(GrB_Type,...,GrB_Type*)
GrB_Type_get(GrB_Type,...,void*)	GrB_Type_get_VOID(GrB_Type,...,void*)
GrB_Global_get(...,GrB_Scalar)	GrB_Global_get_Scalar(...,GrB_Scalar)
GrB_Global_get(...,char*)	GrB_Global_get_String(...,char*)
GrB_Global_get(...,GrB_Global*)	GrB_Global_get_Global(...,GrB_Global*)
GrB_Global_get(...,void*)	GrB_Global_get_VOID(...,void*)

Table 5.13: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_set(GrB_Scalar,...,GrB_Scalar)	GrB_Scalar_set_Scalar(GrB_Scalar,...,GrB_Scalar)
GrB_Scalar_set(GrB_Scalar,...,char*)	GrB_Scalar_set_String(GrB_Scalar,...,char*)
GrB_Scalar_set(GrB_Scalar,...,void*)	GrB_Scalar_set_VOID(GrB_Scalar,...,void*)
GrB_Vector_set(GrB_Vector,...,GrB_Scalar)	GrB_Vector_set_Scalar(GrB_Vector,...,GrB_Scalar)
GrB_Vector_set(GrB_Vector,...,char*)	GrB_Vector_set_String(GrB_Vector,...,char*)
GrB_Vector_set(GrB_Vector,...,void*)	GrB_Vector_set_VOID(GrB_Vector,...,void*)
GrB_Matrix_set(GrB_Matrix,...,GrB_Scalar)	GrB_Matrix_set_Scalar(GrB_Matrix,...,GrB_Scalar)
GrB_Matrix_set(GrB_Matrix,...,char*)	GrB_Matrix_set_String(GrB_Matrix,...,char*)
GrB_Matrix_set(GrB_Matrix,...,void*)	GrB_Matrix_set_VOID(GrB_Matrix,...,void*)
GrB_UnaryOp_set(GrB_UnaryOp,...,GrB_Scalar)	GrB_UnaryOp_set_Scalar(GrB_UnaryOp,...,GrB_Scalar)
GrB_UnaryOp_set(GrB_UnaryOp,...,char*)	GrB_UnaryOp_set_String(GrB_UnaryOp,...,char*)
GrB_UnaryOp_set(GrB_UnaryOp,...,void*)	GrB_UnaryOp_set_VOID(GrB_UnaryOp,...,void*)
GrB_IndexUnaryOp_set(GrB_IndexUnaryOp,...,GrB_Scalar)	GrB_IndexUnaryOp_set_Scalar(GrB_IndexUnaryOp,...,GrB_Scalar)
GrB_IndexUnaryOp_set(GrB_IndexUnaryOp,...,char*)	GrB_IndexUnaryOp_set_String(GrB_IndexUnaryOp,...,char*)
GrB_IndexUnaryOp_set(GrB_IndexUnaryOp,...,void*)	GrB_IndexUnaryOp_set_VOID(GrB_IndexUnaryOp,...,void*)
GrB_BinaryOp_set(GrB_BinaryOp,...,GrB_Scalar)	GrB_BinaryOp_set_Scalar(GrB_BinaryOp,...,GrB_Scalar)
GrB_BinaryOp_set(GrB_BinaryOp,...,char*)	GrB_BinaryOp_set_String(GrB_BinaryOp,...,char*)
GrB_BinaryOp_set(GrB_BinaryOp,...,void*)	GrB_BinaryOp_set_VOID(GrB_BinaryOp,...,void*)
GrB_Monoid_set(GrB_Monoid,...,GrB_Scalar)	GrB_Monoid_set_Scalar(GrB_Monoid,...,GrB_Scalar)
GrB_Monoid_set(GrB_Monoid,...,char*)	GrB_Monoid_set_String(GrB_Monoid,...,char*)
GrB_Monoid_set(GrB_Monoid,...,void*)	GrB_Monoid_set_VOID(GrB_Monoid,...,void*)
GrB_Semiring_set(GrB_Semiring,...,GrB_Scalar)	GrB_Semiring_set_Scalar(GrB_Semiring,...,GrB_Scalar)
GrB_Semiring_set(GrB_Semiring,...,char*)	GrB_Semiring_set_String(GrB_Semiring,...,char*)
GrB_Semiring_set(GrB_Semiring,...,void*)	GrB_Semiring_set_VOID(GrB_Semiring,...,void*)
GrB_Descriptor_set(GrB_Descriptor,...,GrB_Scalar)	GrB_Descriptor_set_Scalar(GrB_Descriptor,...,GrB_Scalar)
GrB_Descriptor_set(GrB_Descriptor,...,char*)	GrB_Descriptor_set_String(GrB_Descriptor,...,char*)
GrB_Descriptor_set(GrB_Descriptor,...,void*)	GrB_Descriptor_set_VOID(GrB_Descriptor,...,void*)
GrB_Type_set(GrB_Type,...,GrB_Scalar)	GrB_Type_set_Scalar(GrB_Type,...,GrB_Scalar)
GrB_Type_set(GrB_Type,...,char*)	GrB_Type_set_String(GrB_Type,...,char*)
GrB_Type_set(GrB_Type,...,void*)	GrB_Type_set_VOID(GrB_Type,...,void*)
GrB_Global_set(...,GrB_Scalar)	GrB_Global_set_Scalar(...,GrB_Scalar)
GrB_Global_set(...,char*)	GrB_Global_set_String(...,char*)
GrB_Global_set(...,void*)	GrB_Global_set_VOID(...,void*)

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.

- 7508 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified
7509 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats
7510 have been deferred.
- 7511 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7512 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return
7513 error if duplicate locations encountered.
- 7514 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7515 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7516 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type T (not to
7517 be confused with the proposed unary operator).
- 7518 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added
7519 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to
7520 a method that is not supported by the implementation.
- 7521 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque
7522 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7523 • (Issue BB-45) Removed language about annihilators.
- 7524 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7525 • Updated a number algorithms in the appendix to use new operations and methods.
- 7526 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the
7527 specification.
- 7528 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and
7529 `GRB_SUBVERSION`.
- 7530 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7531 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection
7532 fixed to read `eWiseAdd`/set union.
- 7533 • Typographical error (`NEQ` \rightarrow `NE`) in Description of Table 3.8.

7534 Changes in 1.3.0 (Released: 25 September 2019):

- 7535 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque
7536 GraphBLAS object as an argument.
- 7537 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7538 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a
7539 scalar.

7540
7541
7542
7543
7544
7545
7546
7547
7548
7549
7550
7551
7552
7553
7554
7555
7556
7557
7558
7559
7560
7561
7562
7563
7564
7565
7566
7567
7568
7569
7570
7571
7572

- (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.

- 7573 • Fixed miscellaneous typographical errors (such as \otimes , \oplus).
- 7574 Changes in 1.2.0:
- 7575 • Removed "provisional" clause.
- 7576 Changes in 1.1.0:
- 7577 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
 - 7578 `assign` operations.
 - 7579 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
 - 7580 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
 - 7581 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
 - 7582 from the domains of the binary operator provided.
 - 7583 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
 - 7584 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
 - 7585 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
 - 7586 which is returned when the capacities of the output arrays are insufficient to hold all of the
 - 7587 tuples.
 - 7588 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
 - 7589 face.
 - 7590 • Added replace flag (`z`) notation to Table 4.1.
 - 7591 • Updated the "Mathematical Description" of the `assign` operation in Table 4.1.
 - 7592 • Added triangle counting example.
 - 7593 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
 - 7594 sections of GraphBLAS operations when the respective text was the "standard" text (i.e.,
 - 7595 identical in a majority of the operations).
 - 7596 • Fixed typographical errors.
- 7597 Changes in 1.0.2:
- 7598 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
 - 7599 matrices and avoid casting issues in certain implementations.
 - 7600 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
 - 7601 erased outside the assigned area.
 - 7602 • Changes non-polymorphic interface:
 - 7603 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7604 – Renamed GrB_Vector_reduce_BinaryOp to GrB_Matrix_reduce_BinaryOp.
- 7605 – Renamed GrB_Vector_reduce_Monoid to GrB_Matrix_reduce_Monoid.
- 7606 • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7607 • 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.

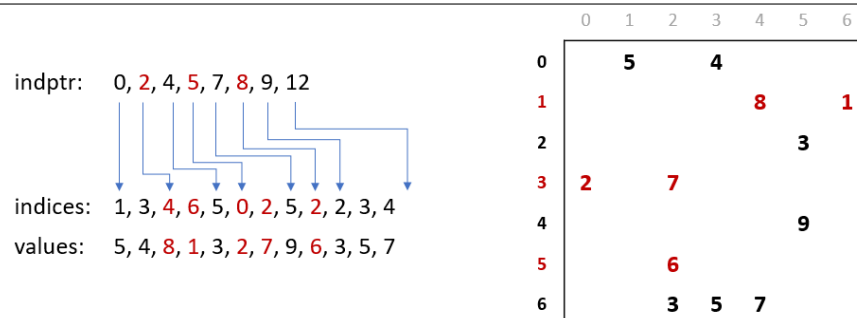


Figure B.1: Data layout for CSR format.

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.

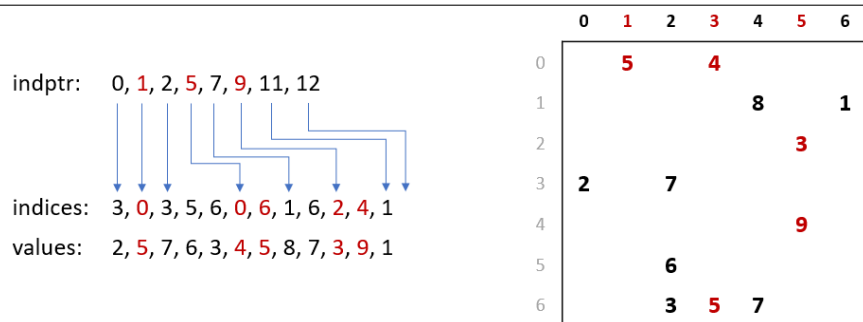


Figure B.2: Data layout for CSC format.

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.

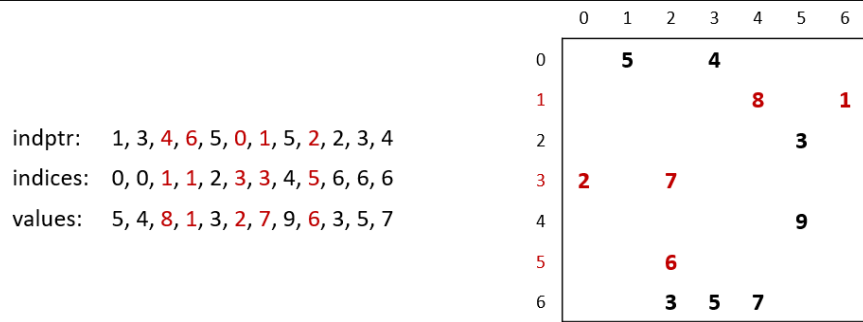


Figure B.3: Data layout for COO format.

7640 **Appendix C**

7641 **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);      //  $q[!v] = q \vee A$ ; 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); //  $\text{Matrix}<\text{int32}_t> \text{sigma}(n,n)$ 
20                                           //  $\text{sigma}[d,k] = \#$  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); //  $\text{sigma}[d,:] = q$ 
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); //  $t2 = \text{sigma}[i,:]$ 
63         GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t2 = (1 + \delta) / \text{sigma}[i,:]$ 
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 }
```