

The GraphBLAS C API Specification [†]:

Version 2.1

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

Benjamin Brock, Aydın Buluç, Raye Kimmerer, Jim Kitchen, Manoj Kumar, Timothy
Mattson, Scott McMillan, José Moreira, Erik Welch

Generated on 2023/05/15 at 16:37:22 EDT

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

7 Copyright © 2017-2023 Carnegie Mellon University, The Regents of the University of California,
8 through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from
9 the U.S. Dept. of Energy), the Regents of the University of California (U.C. Davis and U.C.
10 Berkeley), Intel Corporation, International Business Machines Corporation, and Massachusetts
11 Institute of Technology Lincoln Laboratory.

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

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

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

Contents

27	List of Tables	9
28	List of Figures	11
29	Acknowledgments	12
30	1 Introduction	15
31	2 Basic concepts	17
32	2.1 Glossary	17
33	2.1.1 GraphBLAS API basic definitions	17
34	2.1.2 GraphBLAS objects and their structure	18
35	2.1.3 Algebraic structures used in the GraphBLAS	19
36	2.1.4 The execution of an application using the GraphBLAS C API	20
37	2.1.5 GraphBLAS methods: behaviors and error conditions	21
38	2.2 Notation	23
39	2.3 Mathematical foundations	24
40	2.4 GraphBLAS opaque objects	25
41	2.5 Execution model	26
42	2.5.1 Execution modes	27
43	2.5.2 Multi-threaded execution	28
44	2.6 Error model	30
45	3 Objects	33
46	3.1 Enumerations for <code>init()</code> and <code>wait()</code>	33
47	3.2 Indices, index arrays, and scalar arrays	33
48	3.3 Types (domains)	34

49	3.4	Algebraic objects, operators and associated functions	35
50	3.4.1	Operators	36
51	3.4.2	Monoids	41
52	3.4.3	Semirings	41
53	3.5	Collections	45
54	3.5.1	Scalars	45
55	3.5.2	Vectors	45
56	3.5.3	Matrices	46
57	3.5.3.1	External matrix formats	46
58	3.5.4	Masks	46
59	3.6	Descriptors	47
60	3.7	Fields	48
61	3.7.1	Input Types	51
62	3.7.1.1	ENUM Handling	51
63	3.7.1.2	GrB_Scalar Handling	51
64	3.7.1.3	String (char*) Handling	51
65	3.7.1.4	void* Handling	51
66	3.7.2	Hints	51
67	3.7.3	GrB_NAME	52
68	3.8	GrB_Info return values	54
69	4	Methods	57
70	4.1	Context methods	57
71	4.1.1	init: Initialize a GraphBLAS context	57
72	4.1.2	finalize: Finalize a GraphBLAS context	58
73	4.1.3	getVersion: Get the version number of the standard.	59
74	4.2	Object methods	59
75	4.2.1	Get and Set methods	60
76	4.2.1.1	get: Query the value of an object	60
77	4.2.1.2	set: Set field of an object	60
78	4.2.2	Algebra methods	61

79	4.2.2.1	Type_new: Construct a new GraphBLAS (user-defined) type	61
80	4.2.2.2	UnaryOp_new: Construct a new GraphBLAS unary operator	62
81	4.2.2.3	BinaryOp_new: Construct a new GraphBLAS binary operator . . .	64
82	4.2.2.4	Monoid_new: Construct a new GraphBLAS monoid	66
83	4.2.2.5	Semiring_new: Construct a new GraphBLAS semiring	67
84	4.2.2.6	IndexUnaryOp_new: Construct a new GraphBLAS index unary op-	
85		erator [Scott: NEW CONTENT]	68
86	4.2.3	Scalar methods	70
87	4.2.3.1	Scalar_new: Construct a new scalar	70
88	4.2.3.2	Scalar_dup: Construct a copy of a GraphBLAS scalar	71
89	4.2.3.3	Scalar_clear: Clear/remove a stored value from a scalar	72
90	4.2.3.4	Scalar_nvals: Number of stored elements in a scalar	73
91	4.2.3.5	Scalar_setElement: Set the single element in a scalar	74
92	4.2.3.6	Scalar_extractElement: Extract a single element from a scalar. . . .	75
93	4.2.4	Vector methods	76
94	4.2.4.1	Vector_new: Construct new vector	76
95	4.2.4.2	Vector_dup: Construct a copy of a GraphBLAS vector	77
96	4.2.4.3	Vector_resize: Resize a vector	78
97	4.2.4.4	Vector_clear: Clear a vector	79
98	4.2.4.5	Vector_size: Size of a vector	80
99	4.2.4.6	Vector_nvals: Number of stored elements in a vector	81
100	4.2.4.7	Vector_build: Store elements from tuples into a vector	82
101	4.2.4.8	Vector_setElement: Set a single element in a vector	84
102	4.2.4.9	Vector_removeElement: Remove an element from a vector	86
103	4.2.4.10	Vector_extractElement: Extract a single element from a vector. . . .	87
104	4.2.4.11	Vector_extractTuples: Extract tuples from a vector	89
105	4.2.5	Matrix methods	90
106	4.2.5.1	Matrix_new: Construct new matrix	90
107	4.2.5.2	Matrix_dup: Construct a copy of a GraphBLAS matrix	92
108	4.2.5.3	Matrix_diag: Construct a diagonal GraphBLAS matrix	93
109	4.2.5.4	Matrix_resize: Resize a matrix	94

110	4.2.5.5	Matrix_clear: Clear a matrix	95
111	4.2.5.6	Matrix_nrows: Number of rows in a matrix	96
112	4.2.5.7	Matrix_ncols: Number of columns in a matrix	96
113	4.2.5.8	Matrix_nvals: Number of stored elements in a matrix	97
114	4.2.5.9	Matrix_build: Store elements from tuples into a matrix	98
115	4.2.5.10	Matrix_setElement: Set a single element in matrix	100
116	4.2.5.11	Matrix_removeElement: Remove an element from a matrix	102
117	4.2.5.12	Matrix_extractElement: Extract a single element from a matrix . . .	103
118	4.2.5.13	Matrix_extractTuples: Extract tuples from a matrix	105
119	4.2.5.14	Matrix_exportHint: Provide a hint as to which storage format might	
120		be most efficient for exporting a matrix	107
121	4.2.5.15	Matrix_exportSize: Return the array sizes necessary to export a	
122		GraphBLAS matrix object	108
123	4.2.5.16	Matrix_export: Export a GraphBLAS matrix to a pre-defined format	109
124	4.2.5.17	Matrix_import: Import a matrix into a GraphBLAS object	111
125	4.2.5.18	Matrix_serializeSize: Compute the serialize buffer size	113
126	4.2.5.19	Matrix_serialize: Serialize a GraphBLAS matrix.	114
127	4.2.5.20	Matrix_deserialize: Deserialize a GraphBLAS matrix.	115
128	4.2.6	Descriptor methods	116
129	4.2.6.1	Descriptor_new: Create new descriptor	116
130	4.2.6.2	Descriptor_set: Set content of descriptor	117
131	4.2.7	free: Destroy an object and release its resources	118
132	4.2.8	wait: Return once an object is either <i>complete</i> or <i>materialized</i>	120
133	4.2.9	error: Retrieve an error string	121
134	4.3	GraphBLAS operations	122
135	4.3.1	mxm: Matrix-matrix multiply	126
136	4.3.2	vxm: Vector-matrix multiply	131
137	4.3.3	mxv: Matrix-vector multiply	135
138	4.3.4	eWiseMult: Element-wise multiplication	139
139	4.3.4.1	eWiseMult: Vector variant	140
140	4.3.4.2	eWiseMult: Matrix variant	144

141	4.3.5	eWiseAdd: Element-wise addition	149
142	4.3.5.1	eWiseAdd: Vector variant	150
143	4.3.5.2	eWiseAdd: Matrix variant	154
144	4.3.6	extract: Selecting sub-graphs	160
145	4.3.6.1	extract: Standard vector variant	160
146	4.3.6.2	extract: Standard matrix variant	164
147	4.3.6.3	extract: Column (and row) variant	169
148	4.3.7	assign: Modifying sub-graphs	174
149	4.3.7.1	assign: Standard vector variant	174
150	4.3.7.2	assign: Standard matrix variant	179
151	4.3.7.3	assign: Column variant	185
152	4.3.7.4	assign: Row variant	190
153	4.3.7.5	assign: Constant vector variant[Scott: NEW CONTENT]	196
154	4.3.7.6	assign: Constant matrix variant[Scott: NEW CONTENT]	201
155	4.3.8	apply: Apply a function to the elements of an object	207
156	4.3.8.1	apply: Vector variant	207
157	4.3.8.2	apply: Matrix variant	212
158	4.3.8.3	apply: Vector-BinaryOp variants[Scott: NEW CONTENT]	216
159	4.3.8.4	apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]	222
160	4.3.8.5	apply: Vector index unary operator variant[Scott: NEW CONTENT]	228
161	4.3.8.6	apply: Matrix index unary operator variant[Scott: NEW CONTENT]	233
162	4.3.9	select:	238
163	4.3.9.1	select: Vector variant[Scott: NEW CONTENT]	238
164	4.3.9.2	select: Matrix variant[Scott: NEW CONTENT]	243
165	4.3.10	reduce: Perform a reduction across the elements of an object	249
166	4.3.10.1	reduce: Standard matrix to vector variant	249
167	4.3.10.2	reduce: Vector-scalar variant[Scott: NEW CONTENT]	253
168	4.3.10.3	reduce: Matrix-scalar variant[Scott: NEW CONTENT]	257
169	4.3.11	transpose: Transpose rows and columns of a matrix	260
170	4.3.12	kronecker: Kronecker product of two matrices	264

171	5 Nonpolymorphic interface [Scott: NEW CONTENT]	271
172	A Revision history	285
173	B Non-opaque data format definitions	291
174	B.1 GrB_Format: Specify the format for input/output of a GraphBLAS matrix.	291
175	B.1.1 GrB_CSR_FORMAT	291
176	B.1.2 GrB_CSC_FORMAT	292
177	B.1.3 GrB_COO_FORMAT	292
178	C Examples	293
179	C.1 Example: Level breadth-first search (BFS) in GraphBLAS	294
180	C.2 Example: Level BFS in GraphBLAS using apply	295
181	C.3 Example: Parent BFS in GraphBLAS	296
182	C.4 Example: Betweenness centrality (BC) in GraphBLAS	297
183	C.5 Example: Batched BC in GraphBLAS	299
184	C.6 Example: Maximal independent set (MIS) in GraphBLAS	301
185	C.7 Example: Counting triangles in GraphBLAS	303

List of Tables

186		
187	2.1	Types of GraphBLAS opaque objects. 25
188	2.2	Methods that forced completion prior to GraphBLAS v2.0. 30
189	3.1	Enumeration literals and corresponding values input to various GraphBLAS methods. 34
190	3.2	Predefined GrB_Type values. 35
191	3.3	Operator input for relevant GraphBLAS operations. 36
192	3.4	Properties and recipes for building GraphBLAS algebraic objects. 37
193	3.5	Predefined unary and binary operators for GraphBLAS in C. 39
194	3.6	Predefined index unary operators for GraphBLAS in C. 40
195	3.7	Predefined monoids for GraphBLAS in C. 42
196	3.8	Predefined “true” semirings for GraphBLAS in C. 43
197	3.9	Other useful predefined semirings for GraphBLAS in C. 44
198	3.10	GrB_Format enumeration literals and corresponding values for matrix import and
199		export methods. 46
200	3.11	Descriptor types and literals for fields and values. 49
201	3.12	Predefined GraphBLAS descriptors. 50
202	3.13	Field values of type GrB_Field enumeration, corresponding types, and the objects
203		which must implement that GrB_Field. Collection refers to GrB_Matrix, GrB_Vector,
204		and GrB_Scalar, Algebraic refers to Operators, Monoids, and Semirings, while All refers
205		to all GraphBLAS objects. Global fields are denoted by Global. All fields may be
206		read, some may be written (denoted by W), and some are hints (denoted by H) which
207		may be ignored by the implementation. For * see 3.7 53
208	3.14	Descriptions of select <i>field</i> , <i>value</i> pairs listed in 3.13 54
209	3.15	Field value enumerations. 55
210	3.16	Enumeration literals and corresponding values returned by GraphBLAS methods
211		and operations. 56

212	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in	
213		this specification.	123
214	5.1	Long-name, nonpolymorphic form of GraphBLAS methods.	271
215	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	272
216	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	273
217	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	274
218	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	275
219	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	276
220	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	277
221	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	278
222	5.9	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
223		CONTENT]	279
224	5.10	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
225		CONTENT]	280
226	5.11	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	281
227	5.12	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	282
228	5.13	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	283

229 List of Figures

230	3.1 Hierarchy of algebraic object classes in GraphBLAS.	45
231	4.1 Flowchart for the GraphBLAS operations.	124
232	B.1 Data layout for CSR format.	291
233	B.2 Data layout for CSC format.	292
234	B.3 Data layout for COO format.	292

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 2.1 are (in alphabetical order):

- Raye Kimmerer (MIT)
- Jim Kitchen (Anaconda)
- Manoj Kumar (?)
- Timothy G. Mattson (Intel Corporation)
- Erik Welch (Nvidia Corporation)

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, Jim 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*

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

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

2.2 Notation

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

2.3 Mathematical foundations

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

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

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

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

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

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

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

Table 2.1: Types of GraphBLAS opaque objects.

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

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

2.4 GraphBLAS opaque objects

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

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

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

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

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

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

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

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

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

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

2.5 Execution model

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

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

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

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

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

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

2.5.1 Execution modes

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

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

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

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

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

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

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

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

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

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

2.5.2 Multi-threaded execution

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

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

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

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

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

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

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

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

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

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

Method	Section
GrB_Vector_nvals	4.2.4.6
GrB_Vector_extractElement	4.2.4.10
GrB_Vector_extractTuples	4.2.4.11
GrB_Matrix_nvals	4.2.5.8
GrB_Matrix_extractElement	4.2.5.12
GrB_Matrix_extractTuples	4.2.5.13
GrB_reduce (vector-scalar value variant)	4.3.10.2
GrB_reduce (matrix-scalar value variant)	4.3.10.3

2.6 Error model

All GraphBLAS methods return a value of type `GrB_Info` (an enum) to provide information available to the system at the time the method returns. The returned value will be one of the defined values shown in Table 3.16. The return values fall into three groups: informational, API errors, and execution errors. While API and execution errors take on negative values, informational return values listed in Table 3.16(a) are non-negative and include `GrB_SUCCESS` (a value of 0) and `GrB_NO_VALUE`.

An API error (listed in Table 3.16(b)) means that a GraphBLAS method was called with parameters that violate the rules for that method. These errors are restricted to those that can be determined by inspecting the dimensions and domains of GraphBLAS objects, GraphBLAS operators, or the values of scalar parameters fixed at the time a method is called. API errors are deterministic and consistent across platforms and implementations. API errors are never deferred, even in nonblocking mode. That is, if a method is called in a manner that would generate an API error, it always returns with the appropriate API error value. If a GraphBLAS method returns with an API error, it is guaranteed that none of the arguments to the method (or any other program data) have been modified. The informational return value, `GrB_NO_VALUE`, is also deterministic and never deferred in nonblocking mode.

Execution errors (listed in Table 3.16(c)) indicate that something went wrong during the execution of a legal GraphBLAS method invocation. Their occurrence may depend on specifics of the execution environment and data values being manipulated. This does not mean that execution errors are the fault of the GraphBLAS implementation. For example, a memory leak could arise from an error in an application’s source code (a “program error”), but it may manifest itself in different points of a program’s execution (or not at all) depending on the platform, problem size, or what else is running at that time. Index out-of-bounds errors, for example, always indicate a program error.

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

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

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

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

762 After a call to any GraphBLAS method that modifies an opaque object, the program can re-
763 trieve additional error information (beyond the error code returned by the method) though a call
764 to the function `GrB_error()`, passing the method's output object as described in Section 4.2.9.
765 The function returns a pointer to a NULL-terminated string, and the contents of that string are
766 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error
767 string. `GrB_error()` is a thread-safe function, in the sense that multiple threads can call it simul-
768 taneously and each will get its own error string back, referring to the object passed as an input
769 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:

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

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

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

806 3.3 Types (domains)

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

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

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

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

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

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

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

GrB_Type	GrB_Type_Code	Suffix	C type	Domain
-	GrB_UDT_CODE=0	UDT	-	-
GrB_BOOL	GrB_BOOL_CODE=1	BOOL	bool	{false, true}
GrB_INT8	GrB_INT8_CODE=2	INT8	int8_t	$\mathbb{Z} \cap [-2^7, 2^7)$
GrB_UINT8	GrB_UINT8_CODE=3	UINT8	uint8_t	$\mathbb{Z} \cap [0, 2^8)$
GrB_INT16	GrB_INT16_CODE=4	INT16	int16_t	$\mathbb{Z} \cap [-2^{15}, 2^{15})$
GrB_UINT16	GrB_UINT16_CODE=5	UINT16	uint16_t	$\mathbb{Z} \cap [0, 2^{16})$
GrB_INT32	GrB_INT32_CODE=6	INT32	int32_t	$\mathbb{Z} \cap [-2^{31}, 2^{31})$
GrB_UINT32	GrB_UINT32_CODE=7	UINT32	uint32_t	$\mathbb{Z} \cap [0, 2^{32})$
GrB_INT64	GrB_INT64_CODE=8	INT64	int64_t	$\mathbb{Z} \cap [-2^{63}, 2^{63})$
GrB_UINT64	GrB_UINT64_CODE=9	UINT64	uint64_t	$\mathbb{Z} \cap [0, 2^{64})$
GrB_FP32	GrB_FP32_CODE=10	FP32	float	IEEE 754 binary32
GrB_FP64	GrB_FP64_CODE=11	FP64	double	IEEE 754 binary64

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

3.4 Algebraic objects, operators and associated functions

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

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

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

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

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

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

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

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

3.4.1 Operators

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

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

(a) Properties of algebraic objects.

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

(b) Recipes for algebraic objects.

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

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

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

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

853 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} ,
854 D_{in_2} , and an operation $\odot : D_{in_1} \times D_{in_2} \rightarrow D_{out}$. For a given GraphBLAS binary operator $F_b =$
855 $\langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$, 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 $\odot(F_b) =$
856 \odot . Note that \odot could be used in place of either \oplus or \otimes in other methods and operations.

857 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
858 domains, D_{out} , D_{in_1} , D_{in_2} , the domain of GraphBLAS indices, and an operation $f_i : D_{in_1} \times I_{U64}^2 \times$
859 $D_{in_2} \rightarrow D_{out}$ (where I_{U64} corresponds to the domain of a `GrB_Index`). For a given GraphBLAS
860 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 $\mathbf{f}(F_i) = f_i$.

861 User-defined operators can be created with calls to `GrB_UnaryOp_new`, `GrB_BinaryOp_new`, and
862 `GrB_IndexUnaryOp_new`, respectively. See Section 4.2.2 for information on these methods. The
863 GraphBLAS C API predefines a number of these operators. These are listed in Tables 3.5 and 3.6.
864 Note that most entries in these tables represent a “family” of predefined operators for a set of
865 different types represented by the T , I , or F in their names. For example, the multiplicative
866 inverse (`GrB_MINV_F`) function is only defined for floating-point types ($F = \text{FP32}$ or FP64). The
867 division (`GrB_DIV_T`) function is defined for all types, but only if $y \neq 0$ for integral and floating
868 point types and $y \neq \text{false}$ for the Boolean type.

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

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

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

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

3.4.2 Monoids

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

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

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

3.4.3 Semirings

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

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

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

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

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

User-defined semirings can be created with calls to `GrB_Semiring_new` (see Section 4.2.2). A list of predefined true semirings and convenience semirings can be found in Tables 3.8 and 3.9, respectively. Predefined semirings are named `GrB_add_mul_SEMIRING_T`, where *add* is the semiring additive operation, *mul* is the semiring multiplicative operation and *T* is the domain (type) of the semiring.

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

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

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

Table 3.8: Predefined true semirings for GraphBLAS in C where the additive identity is the multiplicative annihilator. The x can be one of 8, 16, 32, or 64 in `UINT x` or `INT x` , and can be 32 or 64 in `FP x` .

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

Table 3.9: Other useful predefined semirings for GraphBLAS in C that don't have a multiplicative annihilator. The x can be one of 8, 16, 32, or 64 in $\text{UINT}x$ or $\text{INT}x$, and can be 32 or 64 in $\text{FP}x$.

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

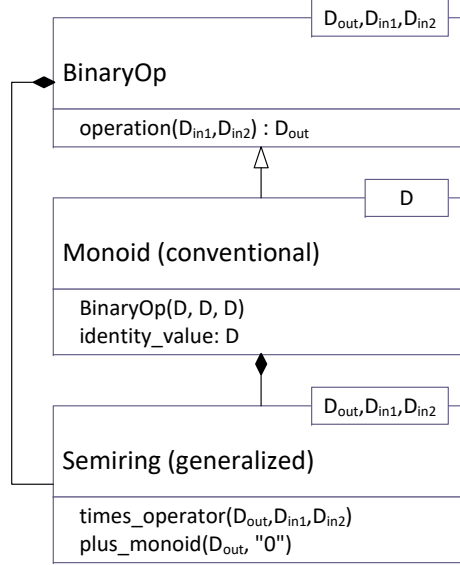


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

3.5 Collections

3.5.1 Scalars

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

3.5.2 Vectors

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

3.5.3 Matrices

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

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

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

3.5.3.1 External matrix formats

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

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

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

3.5.4 Masks

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

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

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

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

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

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

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

3.6 Descriptors

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

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

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

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

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

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

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

3.7 Fields

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

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

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

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

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

(a) Types used with GraphBLAS descriptors.

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

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

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

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

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

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

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

1019 3.7.1 Input Types

1020 Allowable types used in `GrB_get` and `GrB_set` are `ENUM`, `GrB_Scalar`, `char*`, and `void*`. Each
1021 `GrB_Field` is associated with exactly one of these types as defined in Table 3.13. Implementations
1022 that add additional `GrB_Fields` must document the type associated with each `GrB_Field`.

1023 3.7.1.1 ENUM Handling

1024 `ENUM` types use standard `int` enumerations defined in C. User code should use the enum name
1025 rather than the integer value directly when getting or setting a field value.

1026 3.7.1.2 GrB_Scalar Handling

1027 When calling `GrB_get`, the user must provide an already initialized `GrB_Scalar` object to which
1028 the implementation will write a value of the correct element type. When calling `GrB_set`, the
1029 `GrB_Scalar` must not be empty, otherwise a `GrB_EMPTY_OBJECT` error is raised.

1030 3.7.1.3 String (char*) Handling

1031 When the input to `GrB_set` is a `char*` the input array is null terminated. The GraphBLAS imple-
1032 mentation must copy this array into internal data structures. Using `GrB_get` for strings requires
1033 two calls. First, call `GrB_get` with the field and object, but pass `int*` as the last argument. The
1034 implementation will return the size of the string buffer that the user must create. Second, call
1035 `GrB_get` with the field and object, this time passing a pointer to the newly created string buffer.
1036 The GraphBLAS implementation will write to this buffer, including a trailing null terminator. The
1037 size returned in the first call will include enough bytes for the null terminator.

1038 3.7.1.4 void* Handling

1039 When the input to `GrB_set` is a `void*`, an extra `int` argument is passed to indicate the size of the
1040 buffer. The GraphBLAS implementation must copy this many bytes from the buffer into internal
1041 data structures. Similar to reading strings, `GrB_get` must be called twice for `void*`. The first
1042 call passes `int*` to find the required buffer size. The user must create a buffer and then pass the
1043 pointer to `GrB_get`. The implementation will write to this buffer. No standard specification or
1044 protocol is required for the contents of `void*`. It is meant to be a mechanism to allow full freedom
1045 for GraphBLAS implementations with needs that cannot be handled using `ENUM`, `GrB_Scalar`, or
1046 `Strings`.

1047 3.7.2 Hints

1048 Several fields are *hints* (marked H in Table 3.13). Hints are used to represent intended use cases
1049 or best guesses, but do not determine strict behavior. When `GrB_set` is called with a hint, the

1050 GraphBLAS implementation should return `GrB_SUCCESS`, but is free to use or ignore the hint.
1051 When `GrB_get` is called, the implementation should return a best guess on the correct answer. If
1052 there is no clear answer, the implementation should return `GrB_UNKNOWN`.

1053 3.7.3 `GrB_NAME`

1054 The `GrB_NAME` field is a special case regarding writability. All objects which have a `GrB_NAME`
1055 field default to an empty string. Collections and `GrB_Descriptors` may have their `GrB_NAME` set
1056 at any time. User-defined algebraic objects and `GrB_Types` may only have their `GrB_NAME` set
1057 once to a globally unique value. Attempting to set this field after it has already been set will return
1058 a `GrB_ALREADY_SET` error code.

1059 Built-in algebraic objects and `GrB_Types` have names which can be read, but not written to. The
1060 name returned will be the string form of the `GrB_Type` listed in Table 3.2 or the GraphBLAS
1061 identifier listed in Tables 3.5, 3.6, 3.7, 3.8, and 3.9. For example, the name of `GrB_INT32` type is
1062 "`GrB_INT32`" (9 characters) and the name of `GrB_MIN_FP64` binary op is "`GrB_MIN_FP64`" (12
1063 characters).

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

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

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

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

3.8 GrB_Info return values

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

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

(a) Field values of type GrB_Orientation.

Value Name	Value	Description
GrB_ROWMAJOR	0	The majority of iteration over the object will be row-wise.
GrB_COLMAJOR	1	The majority of iteration over the object will be column-wise.
GrB_BOTH	2	Iteration may occur with equal frequency in both directions.
GrB_UNKNOWN	3	No indication is given or is unknown.

Table 3.16: Enumeration literals and corresponding values returned by GraphBLAS methods and operations.

(a) Informational return values

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

(b) API errors

Symbol	Value	Description
GrB_UNINITIALIZED_OBJECT	-1	A GraphBLAS object is passed to a method before <code>new</code> was called on it.
GrB_NULL_POINTER	-2	A NULL is passed for a pointer parameter.
GrB_INVALID_VALUE	-3	Miscellaneous incorrect values.
GrB_INVALID_INDEX	-4	Indices passed are larger than dimensions of the matrix or vector being accessed.
GrB_DOMAIN_MISMATCH	-5	A mismatch between domains of collections and operations when user-defined domains are in use.
GrB_DIMENSION_MISMATCH	-6	Operations on matrices and vectors with incompatible dimensions.
GrB_OUTPUT_NOT_EMPTY	-7	An attempt was made to build a matrix or vector using an output object that already contains valid tuples (elements).
GrB_NOT_IMPLEMENTED	-8	An attempt was made to call a GraphBLAS method for a combination of input parameters that is not supported by a particular implementation.
GrB_ALREADY_SET	-9	An attempt was made to write to a field which may only be written to once.

(c) Execution errors

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

Chapter 4

Methods

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

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

4.1 Context methods

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

4.1.1 `init`: Initialize a GraphBLAS context

Creates and initializes a GraphBLAS C API context.

C Syntax

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

Parameters

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

1086 **Return Values**

1087 `GrB_SUCCESS` operation completed successfully.

1088 `GrB_PANIC` unknown internal error.

1089 `GrB_INVALID_VALUE` invalid mode specified, or method called multiple times.

1090 **Description**

1091 The `init` method creates and initializes a GraphBLAS C API context. The argument to `GrB_init`
1092 defines the mode for the context. The two available modes are:

- 1093 • `GrB_BLOCKING`: In this mode, each method in a sequence returns after its computations have
1094 completed and output arguments are available to subsequent statements in an application.
1095 When executing in `GrB_BLOCKING` mode, the methods execute in program order.
- 1096 • `GrB_NONBLOCKING`: In this mode, methods in a sequence may return after arguments in
1097 the method have been tested for dimension and domain compatibility within the method
1098 but potentially before their computations complete. Output arguments are available to sub-
1099 sequent GraphBLAS methods in an application. When executing in `GrB_NONBLOCKING`
1100 mode, the methods in a sequence may execute in any order that preserves the mathematical
1101 result defined by the sequence.

1102 An application can only create one context per execution instance. An application may only call
1103 `GrB_Init` once. Calling `GrB_Init` more than once results in undefined behavior.

1104 **4.1.2 finalize: Finalize a GraphBLAS context**

1105 Terminates and frees any internal resources created to support the GraphBLAS C API context.

1106 **C Syntax**

1107 `GrB_Info GrB_finalize();`

1108 **Return Values**

1109 `GrB_SUCCESS` operation completed successfully.

1110 `GrB_PANIC` unknown internal error.

1111 **Description**

1112 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS
1113 C API context. `GrB_finalize` may only be called after a context has been initialized by calling
1114 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-
1115 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined
1116 behavior.

1117 **4.1.3 getVersion: Get the version number of the standard.**

1118 Query the library for the version number of the standard that this library implements.

1119 **C Syntax**

```
1120         GrB_Info GrB_getVersion(unsigned int *version,  
1121                                unsigned int *subversion);
```

1122 **Parameters**

1123 version (OUT) On successful return will hold the value of the major version number.

1124 version (OUT) On successful return will hold the value of the subversion number.

1125 **Return Values**

1126 GrB_SUCCESS operation completed successfully.

1127 GrB_PANIC unknown internal error.

1128 **Description**

1129 The `getVersion` method is used to query the major and minor version number of the GraphBLAS
1130 C API specification that the library implements at runtime. To support compile time queries the
1131 following two macros shall also be defined by the library.

```
1132         #define GRB_VERSION      2  
1133         #define GRB_SUBVERSION  0
```

1134 **4.2 Object methods**

1135 This section describes methods that setup and operate on GraphBLAS opaque objects but are not
1136 part of the the GraphBLAS math specification.

1137 4.2.1 Get and Set methods

1138 The methods in this section query and, optionally, set internal fields of GraphBLAS objects.

1139 4.2.1.1 get: Query the value of an object

1140 C Syntax

```
1141 GrB_Info GrB_get(GrB_<OBJ> o, <type> value, GrB_Field field);
```

1142 Parameters

1143 OBJ (IN) An existing, valid GraphBLAS object (collection, operation, type) which is
1144 being queried. To indicate the global context, the constant `GrB_Global` is used.

1145 value (OUT) A pointer to or `GrB_Scalar` containing a value whose type is dependent
1146 on field which will be filled with the current value of the field. type may be `int*`,
1147 `size_t*`, `GrB_Scalar`, `char*` or `void*`.

1148 field (IN) The field being queried.

1149 Return Value

1150 `GrB_SUCCESS` The method completed successfully.

1151 `GrB_PANIC` unknown internal error.

1152 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1153 `GrB_UNINITIALIZED_OBJECT` the value parameter is `GrB_Scalar` and has not been initialized by
1154 a call to `new`.

1155 `GrB_INVALID_VALUE` invalid value type provided for the field or invalid field.

1156 Description

1157 Queries a field of an existing GraphBLAS object. The type of the argument is uniquely determined
1158 by field. For the case of `char*` and `void*`, the type can be replaced with `size_t*` to get the required
1159 buffer size to hold the response. Fields marked as hints in Table 3.13 will return a hint on how
1160 best to use the object.

1161 4.2.1.2 set: Set field of an object

1162 Set the content for a field for an existing GraphBLAS object.

1163 C Syntax

```
1164     GrB_Info GrB_set(GrB_<OBJ> o, <type> value, GrB_Field field);
1165     GrB_Info GrB_set(GrB_<OBJ> o, <type> value, GrB_Field field, size_t voidSize);
```

1166 Parameters

1167 OBJ (IN) The GraphBLAS object which is having field set. To indicate
1168 the global context, the constant `GrB_Global` is used.

1169 value (IN) A value whose type is dependent on field. type may be a int,
1170 `GrB_Scalar`, `char*` or `void*`.

1171 field (IN) The field being set.

1172 voidSize (IN) The size of the `void*` buffer. Note that a size is not needed for
1173 `char*` because the string is assumed null-terminated.

1174 Return Values

1175 `GrB_SUCCESS` The method completed successfully.

1176 `GrB_PANIC` unknown internal error.

1177 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1178 `GrB_UNINITIALIZED_OBJECT` the `GrB_Scalar` parameter has not been initialized by a call to `new`.

1179 `GrB_INVALID_VALUE` invalid value set on the field, invalid field, or field is read-only.

1180 `GrB_ALREADY_SET` this field has already been set, and may only be set once.

1181 Description

1182 Set a field of OBJ or the Global context to a new value.

1183 4.2.2 Algebra methods

1184 4.2.2.1 Type_new: Construct a new GraphBLAS (user-defined) type

1185 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,
1186 monoids, semirings, vectors and matrices.

1187 C Syntax

```
1188         GrB_Info GrB_Type_new(GrB_Type  *utype,  
1189                               size_t     sizeof(ctype));
```

1190 Parameters

1191 **utype** (INOUT) On successful return, contains a handle to the newly created user-defined
1192 GraphBLAS type object.

1193 **ctype** (IN) A C type that defines the new GraphBLAS user-defined type.

1194 Return Values

1195 **GrB_SUCCESS** operation completed successfully.

1196 **GrB_PANIC** unknown internal error.

1197 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1198 **GrB_NULL_POINTER** utype pointer is NULL.

1199 Description

1200 Given a C type **ctype**, the **Type_new** method returns in **utype** a handle to a new GraphBLAS type
1201 that is equivalent to the C type. Variables of this **ctype** must be a struct, union, or fixed-size array.
1202 In particular, given two variables, **src** and **dst**, of type **ctype**, the following operation must be a
1203 valid way to copy the contents of **src** to **dst**:

```
1204         memcpy(&dst, &src, sizeof(ctype))
```

1205 A new, user-defined type **utype** should be destroyed with a call to **GrB_free(utype)** when no longer
1206 needed.

1207 It is not an error to call this method more than once on the same variable; however, the handle to
1208 the previously created object will be overwritten.

1209 4.2.2.2 UnaryOp_new: Construct a new GraphBLAS unary operator

1210 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types
1211 (domains).

1212 C Syntax

```
1213         GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1214                                 void          (*unary_func)(void*, const void*),  
1215                                 GrB_Type      d_out,  
1216                                 GrB_Type      d_in);
```

1217 Parameters

1218 **unary_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1219 unary operator object.

1220 **unary_func** (IN) a pointer to a user-defined function that takes one input parameter of **d_in**'s
1221 type and returns a value of **d_out**'s type, both passed as **void** pointers. Specifically
1222 the signature of the function is expected to be of the form:

```
1223         void func(void *out, const void *in);
```

1225 **d_out** (IN) The **GrB_Type** of the return value of the unary operator being created. Should
1226 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1227 BLAS type.

1228 **d_in** (IN) The **GrB_Type** of the input argument of the unary operator being created.
1229 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1230 GraphBLAS type.

1231 Return Values

1232 **GrB_SUCCESS** operation completed successfully.

1233 **GrB_PANIC** unknown internal error.

1234 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1235 **GrB_UNINITIALIZED_OBJECT** any **GrB_Type** parameter (for user-defined types) has not been ini-
1236 tialized by a call to **GrB_Type_new**.

1237 **GrB_NULL_POINTER** **unary_op** or **unary_func** pointers are **NULL**.

1238 Description

1239 The **UnaryOp_new** method creates a new GraphBLAS unary operator

1240 $f_u = \langle \mathbf{D}(\mathbf{d_out}), \mathbf{D}(\mathbf{d_in}), \text{unary_func} \rangle$

1241 and returns a handle to it in `unary_op`.

1242 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments
1243 are aliased. In other words, for all invocations of the function:

```
1244     unary_func(out,in);
```

1245 the value of `out` must be the same as if the following code was executed:

```
1246     D(d_in) *tmp = malloc(sizeof(D(d_in)));  
1247     memcpy(tmp,in,sizeof(D(d_in)));  
1248     unary_func(out,tmp);  
1249     free(tmp);
```

1250 It is not an error to call this method more than once on the same variable; however, the handle to
1251 the previously created object will be overwritten.

1252 4.2.2.3 BinaryOp_new: Construct a new GraphBLAS binary operator

1253 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types
1254 (domains).

1255 C Syntax

```
1256     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
1257                               void          (*binary_func)(void*,  
1258                                                         const void*,  
1259                                                         const void*),  
1260                               GrB_Type      d_out,  
1261                               GrB_Type      d_in1,  
1262                               GrB_Type      d_in2);
```

1263 Parameters

1264 `binary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1265 binary operator object.

1266 `binary_func` (IN) A pointer to a user-defined function that takes two input parameters of types
1267 `d_in1` and `d_in2` and returns a value of type `d_out`, all passed as void pointers.
1268 Specifically the signature of the function is expected to be of the form:

```
1269         void func(void *out, const void *in1, const void *in2);
```

1270

1271 **d_out** (IN) The **GrB_Type** of the return value of the binary operator being created. Should
1272 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1273 BLAS type.

1274 **d_in1** (IN) The **GrB_Type** of the left hand argument of the binary operator being created.
1275 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1276 GraphBLAS type.

1277 **d_in2** (IN) The **GrB_Type** of the right hand argument of the binary operator being cre-
1278 ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-
1279 defined GraphBLAS type.

1280 **Return Values**

1281 **GrB_SUCCESS** operation completed successfully.

1282 **GrB_PANIC** unknown internal error.

1283 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1284 **GrB_UNINITIALIZED_OBJECT** the **GrB_Type** (for user-defined types) has not been initialized by a
1285 call to **GrB_Type_new**.

1286 **GrB_NULL_POINTER** **binary_op** or **binary_func** pointer is **NULL**.

1287 **Description**

1288 The **BinaryOp_new** methods creates a new GraphBLAS binary operator

1289 $f_b = \langle \mathbf{D}(\mathbf{d_out}), \mathbf{D}(\mathbf{d_in1}), \mathbf{D}(\mathbf{d_in2}), \mathbf{binary_func} \rangle$

1290 and returns a handle to it in **binary_op**.

1291 The implementation of **binary_func** must be such that it works even if any of the **d_out**, **d_in1**, and
1292 **d_in2** arguments are aliased to each other. In other words, for all invocations of the function:

1293 **binary_func(out, in1, in2);**

1294 the value of **out** must be the same as if the following code was executed:

```
1295        D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1296        D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1297        memcpy(tmp1, in1, sizeof(D(d_in1)));
1298        memcpy(tmp2, in2, sizeof(D(d_in2)));
1299        binary_func(out, tmp1, tmp2);
1300        free(tmp2);
1301        free(tmp1);
```

1302 It is not an error to call this method more than once on the same variable; however, the handle to
1303 the previously created object will be overwritten.

1304 4.2.2.4 Monoid_new: Construct a new GraphBLAS monoid

1305 Creates a new monoid with specified binary operator and identity value.

1306 C Syntax

```
1307         GrB_Info GrB_Monoid_new(GrB_Monoid    *monoid,  
1308                                GrB_BinaryOp    binary_op,  
1309                                <type>          identity);
```

1310 Parameters

1311 monoid (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1312 monoid object.

1313 binary_op (IN) An existing GraphBLAS associative binary operator whose input and output
1314 types are the same.

1315 identity (IN) The value of the identity element of the monoid. Must be the same type as
1316 the type used by the **binary_op** operator.

1317 Return Values

1318 GrB_SUCCESS operation completed successfully.

1319 GrB_PANIC unknown internal error.

1320 GrB_OUT_OF_MEMORY not enough memory available for operation.

1321 GrB_UNINITIALIZED_OBJECT the GrB_BinaryOp (for user-defined operators) has not been initial-
1322 ized by a call to GrB_BinaryOp_new.

1323 GrB_NULL_POINTER monoid pointer is NULL.

1324 GrB_DOMAIN_MISMATCH all three argument types of the binary operator and the type of the
1325 identity value are not the same.

1326 Description

1327 The **Monoid_new** method creates a new monoid $M = \langle \mathbf{D}(\text{binary_op}), \text{binary_op}, \text{identity} \rangle$ and re-
1328 turns a handle to it in **monoid**.

1329 If `binary_op` is not associative, the results of GraphBLAS operations that require associativity of
1330 this monoid will be undefined.

1331 It is not an error to call this method more than once on the same variable; however, the handle to
1332 the previously created object will be overwritten.

1333 4.2.2.5 Semiring_new: Construct a new GraphBLAS semiring

1334 Creates a new semiring with specified domain, operators, and elements.

1335 C Syntax

```
1336         GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1337                                   GrB_Monoid    add_op,  
1338                                   GrB_BinaryOp   mul_op);
```

1339 Parameters

1340 `semiring` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1341 `semiring`.

1342 `add_op` (IN) An existing GraphBLAS commutative monoid that specifies the addition op-
1343 erator and its identity.

1344 `mul_op` (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-
1345 plication operator. In addition, `mul_op`'s output domain, $\mathbf{D}_{out}(\text{mul_op})$, must be
1346 the same as the `add_op`'s domain $\mathbf{D}(\text{add_op})$.

1347 Return Values

1348 `GrB_SUCCESS` operation completed successfully.

1349 `GrB_PANIC` unknown internal error.

1350 `GrB_OUT_OF_MEMORY` not enough memory available for this method to complete.

1351 `GrB_UNINITIALIZED_OBJECT` the `add_op` (for user-define monoids) object has not been initialized
1352 with a call to `GrB_Monoid_new` or the `mul_op` (for user-defined
1353 operators) object has not been not been initialized by a call to
1354 `GrB_BinaryOp_new`.

1355 `GrB_NULL_POINTER` `semiring` pointer is NULL.

1356 `GrB_DOMAIN_MISMATCH` the output domain of `mul_op` does not match the domain of the
1357 `add_op` monoid.

Description

The `Semiring_new` method creates a new semiring:

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

and returns a handle to it in `semiring`. Note that $\mathbf{D}_{out}(\text{mul_op})$ must be the same as $\mathbf{D}(\text{add_op})$. If `add_op` is not commutative, then GraphBLAS operations using this semiring will be undefined. It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

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

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

C Syntax

```
GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp *index_unary_op,
                              void (*index_unary_func)(void*,
                                                         const void*,
                                                         GrB_Index,
                                                         GrB_Index,
                                                         const void*),
                              GrB_Type          d_out,
                              GrB_Type          d_in1,
                              GrB_Type          d_in2);
```

Parameters

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

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

```
void func(void *out,
           const void *in1,
           GrB_Index row_index,
           GrB_Index col_index,
```

1390 `const void *in2);`
 1391

1392 **d_out** (IN) The `GrB_Type` of the return value of the index unary operator being created.
 1393 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
 1394 GraphBLAS type.

1395 **d_in1** (IN) The `GrB_Type` of the first input argument of the index unary operator being
 1396 created and corresponds to the stored values of the `GrB_Vector` or `GrB_Matrix`
 1397 being operated on. Should be one of the predefined GraphBLAS types in Ta-
 1398 ble 3.2, or a user-defined GraphBLAS type.

1399 **d_in2** (IN) The `GrB_Type` of the last input argument of the index unary operator be-
 1400 ing created and corresponds to a scalar provided by the GraphBLAS operation
 1401 that uses this operator. Should be one of the predefined GraphBLAS types in
 1402 Table 3.2, or a user-defined GraphBLAS type.

1403 Return Values

1404 `GrB_SUCCESS` operation completed successfully.

1405 `GrB_PANIC` unknown internal error.

1406 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1407 `GrB_UNINITIALIZED_OBJECT` the `GrB_Type` (for user-defined types) has not been initialized by a
 1408 call to `GrB_Type_new`.

1409 `GrB_NULL_POINTER` `index_unary_op` or `index_unary_func` pointer is `NULL`.

1410 Description

1411 The `IndexUnaryOp_new` methods creates a new GraphBLAS index unary operator

1412
$$f_i = \langle \mathbf{D}(\mathbf{d_out}), \mathbf{D}(\mathbf{d_in1}), \mathbf{D}(\mathbf{GrB_Index}), \mathbf{D}(\mathbf{GrB_Index}), \mathbf{D}(\mathbf{d_in2}), \text{index_unary_func} \rangle$$

1413 and returns a handle to it in `index_unary_op`.

1414 The implementation of `index_unary_func` must be such that it works even if any of the `d_out`,
 1415 `d_in1`, and `d_in2` arguments are aliased to each other. In other words, for all invocations of the
 1416 function:

1417 `index_unary_func(out, in1, row_index, col_index, n, in2);`

1418 the value of `out` must be the same as if the following code was executed (shown here for matrices):

```

1419     GrB_Index row_index = ...;
1420     GrB_Index col_index = ...;
1421     D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1422     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1423     memcpy(tmp1,in1,sizeof(D(d_in1)));
1424     memcpy(tmp2,in2,sizeof(D(d_in2)));
1425     index_unary_func(out,tmp1,row_index,col_index,tmp2);
1426     free(tmp2);
1427     free(tmp1);

```

1428 It is not an error to call this method more than once on the same variable; however, the handle to
1429 the previously created object will be overwritten.

1430 4.2.3 Scalar methods

1431 4.2.3.1 Scalar_new: Construct a new scalar

1432 Creates a new empty scalar with specified domain.

1433 C Syntax

```

1434     GrB_Info GrB_Scalar_new(GrB_Scalar *s,
1435                             GrB_Type    d);

```

1436 Parameters

1437 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1438 scalar.

1439 **d** (IN) The type corresponding to the domain of the scalar being created. Can be
1440 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1441 GraphBLAS type.

1442 Return Values

1443 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1444 blocking mode, this indicates that the API checks for the input
1445 arguments passed successfully. Either way, output scalar **s** is ready
1446 to be used in the next method of the sequence.

1447 **GrB_PANIC** Unknown internal error.

1448 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1449 GraphBLAS objects (input or output) is in an invalid state caused

1450 by a previous execution error. Call `GrB_error()` to access any error
1451 messages generated by the implementation.

1452 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1453 **GrB_UNINITIALIZED_OBJECT** The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`
1454 (needed for user-defined types).

1455 **GrB_NULL_POINTER** The `s` pointer is `NULL`.

1456 Description

1457 Creates a new GraphBLAS scalar s of domain $\mathbf{D}(d)$ and empty $\mathbf{L}(s)$. The method returns a handle
1458 to the new scalar in `s`.

1459 It is not an error to call this method more than once on the same variable; however, the handle to
1460 the previously created object will be overwritten.

1461 4.2.3.2 Scalar_dup: Construct a copy of a GraphBLAS scalar

1462 Creates a new scalar with the same domain and contents as another scalar.

1463 C Syntax

```
1464 GrB_Info GrB_Scalar_dup(GrB_Scalar      *t,  
1465                        const GrB_Scalar s);
```

1466 Parameters

1467 `t` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1468 scalar.

1469 `s` (IN) The GraphBLAS scalar to be duplicated.

1470 Return Values

1471 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1472 blocking mode, this indicates that the API checks for the input
1473 arguments passed successfully. Either way, output scalar `t` is ready
1474 to be used in the next method of the sequence.

1475 **GrB_PANIC** Unknown internal error.

1476 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1477 GraphBLAS objects (input or output) is in an invalid state caused

1478 by a previous execution error. Call `GrB_error()` to access any error
1479 messages generated by the implementation.

1480 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1481 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to
1482 `Scalar_new` or `Scalar_dup`.

1483 `GrB_NULL_POINTER` The `t` pointer is `NULL`.

1484 Description

1485 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
1486 scalar in `t`.

1487 It is not an error to call this method more than once with the same output variable; however, the
1488 handle to the previously created object will be overwritten.

1489 4.2.3.3 `Scalar_clear`: Clear/remove a stored value from a scalar

1490 Removes the stored value from a scalar.

1491 C Syntax

1492 `GrB_Info GrB_Scalar_clear(GrB_Scalar s);`

1493 Parameters

1494 `s` (INOUT) An existing GraphBLAS scalar to clear.

1495 Return Values

1496 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1497 blocking mode, this indicates that the API checks for the input
1498 arguments passed successfully. Either way, output scalar `s` is ready
1499 to be used in the next method of the sequence.

1500 `GrB_PANIC` Unknown internal error.

1501 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1502 GraphBLAS objects (input or output) is in an invalid state caused
1503 by a previous execution error. Call `GrB_error()` to access any error
1504 messages generated by the implementation.

1505 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1506 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to
1507 Scalar_new or Scalar_dup.

1508 Description

1509 Removes the stored value from an existing scalar. After the call, *L(s)* is empty. The size of the
1510 scalar does not change.

1511 4.2.3.4 Scalar_nvals: Number of stored elements in a scalar

1512 Retrieve the number of stored elements in a scalar (either zero or one).

1513 C Syntax

```
1514         GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,  
1515                                   const GrB_Scalar s);
```

1516 Parameters

1517 *nvals* (OUT) On successful return, this is set to the number of stored elements in the
1518 scalar (zero or one).

1519 *s* (IN) An existing GraphBLAS scalar being queried.

1520 Return Values

1521 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1522 cessfully and the value of *nvals* has been set.

1523 GrB_PANIC Unknown internal error.

1524 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1525 GraphBLAS objects (input or output) is in an invalid state caused
1526 by a previous execution error. Call GrB_error() to access any error
1527 messages generated by the implementation.

1528 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1529 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to
1530 Scalar_new or Scalar_dup.

1531 GrB_NULL_POINTER The *nvals* pointer is NULL.

1532 **Description**

1533 Return **nvals(s)** in **nvals**. This is the number of stored elements in scalar **s**, which is the size of
1534 **L(s)**, and can only be either zero or one (see Section 3.5.1).

1535 **4.2.3.5 Scalar_setElement: Set the single element in a scalar**

1536 Set the single element of a scalar to a given value.

1537 **C Syntax**

```
1538         GrB_Info GrB_Scalar_setElement(GrB_Scalar    s,  
1539                                         <type>      val);
```

1540 **Parameters**

1541 **s** (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1542 **val** (IN) Scalar value to assign. The type must be compatible with the domain of **s**.

1543 **Return Values**

1544 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1545 blocking mode, this indicates that the compatibility tests on in-
1546 dex/dimensions and domains for the input arguments passed suc-
1547 cessfully. Either way, the output scalar **s** is ready to be used in the
1548 next method of the sequence.

1549 **GrB_PANIC** Unknown internal error.

1550 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1551 GraphBLAS objects (input or output) is in an invalid state caused
1552 by a previous execution error. Call **GrB_error()** to access any error
1553 messages generated by the implementation.

1554 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1555 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to
1556 **Scalar_new** or **Scalar_dup**.

1557 **GrB_DOMAIN_MISMATCH** The domains of **s** and **val** are incompatible.

Description

First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}(\mathbf{s})$. 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_Scalar_setElement` ends and the domain mismatch error listed above is returned.

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

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

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

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

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

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

C Syntax

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

Parameters

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

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

Return Values

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

`GrB_PANIC` Unknown internal error.

1619 Parameters

- 1620 **v** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1621 vector.
- 1622 **d** (IN) The type corresponding to the domain of the vector being created. Can be
1623 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1624 GraphBLAS type.
- 1625 **nsz** (IN) The size of the vector being created.

1626 Return Values

- 1627 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1628 blocking mode, this indicates that the API checks for the input
1629 arguments passed successfully. Either way, output vector **v** is ready
1630 to be used in the next method of the sequence.
- 1631 **GrB_PANIC** Unknown internal error.
- 1632 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1633 GraphBLAS objects (input or output) is in an invalid state caused
1634 by a previous execution error. Call **GrB_error()** to access any error
1635 messages generated by the implementation.
- 1636 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
- 1637 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
1638 (needed for user-defined types).
- 1639 **GrB_NULL_POINTER** The **v** pointer is **NULL**.
- 1640 **GrB_INVALID_VALUE** **nsz** is zero or outside the range of the type **GrB_Index**.

1641 Description

- 1642 Creates a new vector **v** of domain **D(d)**, size **nsz**, and empty **L(v)**. The method returns a handle
1643 to the new vector in **v**.
- 1644 It is not an error to call this method more than once on the same variable; however, the handle to
1645 the previously created object will be overwritten.

1646 4.2.4.2 Vector_dup: Construct a copy of a GraphBLAS vector

- 1647 Creates a new vector with the same domain, size, and contents as another vector.

1648 C Syntax

```
1649         GrB_Info GrB_Vector_dup(GrB_Vector      *w,  
1650                                 const GrB_Vector  u);
```

1651 Parameters

1652 **w** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1653 vector.

1654 **u** (IN) The GraphBLAS vector to be duplicated.

1655 Return Values

1656 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1657 blocking mode, this indicates that the API checks for the input
1658 arguments passed successfully. Either way, output vector **w** is ready
1659 to be used in the next method of the sequence.

1660 **GrB_PANIC** Unknown internal error.

1661 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1662 GraphBLAS objects (input or output) is in an invalid state caused
1663 by a previous execution error. Call **GrB_error()** to access any error
1664 messages generated by the implementation.

1665 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1666 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **u**, has not been initialized by a call to
1667 **Vector_new** or **Vector_dup**.

1668 **GrB_NULL_POINTER** The **w** pointer is **NULL**.

1669 Description

1670 Creates a new vector **w** of domain **D(u)**, size **size(u)**, and contents **L(u)**. The method returns a
1671 handle to the new vector in **w**.

1672 It is not an error to call this method more than once on the same variable; however, the handle to
1673 the previously created object will be overwritten.

1674 4.2.4.3 Vector_resize: Resize a vector

1675 Changes the size of an existing vector.

1676 C Syntax

```
1677         GrB_Info GrB_Vector_resize(GrB_Vector  w,  
1678                                   GrB_Index   nsize);
```

1679 Parameters

1680 **w** (INOUT) An existing Vector object that is being resized.

1681 **nsize** (IN) The new size of the vector. It can be smaller or larger than the current size.

1682 Return Values

1683 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1684 blocking mode, this indicates that the API checks for the input
1685 arguments passed successfully. Either way, output vector **w** is ready
1686 to be used in the next method of the sequence.

1687 **GrB_PANIC** Unknown internal error.

1688 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1689 GraphBLAS objects (input or output) is in an invalid state caused
1690 by a previous execution error. Call **GrB_error()** to access any error
1691 messages generated by the implementation.

1692 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1693 **GrB_NULL_POINTER** The **w** pointer is NULL.

1694 **GrB_INVALID_VALUE** **nsize** is zero or outside the range of the type **GrB_Index**.

1695 Description

1696 Changes the size of **w** to **nsize**. The domain **D(w)** of vector **w** remains the same. The contents **L(w)**
1697 are modified as described below.

1698 Let $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$ when the method is called. When the method returns, $w = \langle \mathbf{D}(w), \mathbf{nsize}, \mathbf{L}'(w) \rangle$
1699 where $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < \mathbf{nsize})\}$. That is, all elements of **w** with index greater
1700 than or equal to the new vector size (**nsize**) are dropped.

1701 4.2.4.4 Vector_clear: Clear a vector

1702 Removes all the elements (tuples) from a vector.

1703 C Syntax

1704 `GrB_Info GrB_Vector_clear(GrB_Vector v);`

1705 Parameters

1706 `v` (INOUT) An existing GraphBLAS vector to clear.

1707 Return Values

1708 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1709 blocking mode, this indicates that the API checks for the input
1710 arguments passed successfully. Either way, output vector `v` is ready
1711 to be used in the next method of the sequence.

1712 `GrB_PANIC` Unknown internal error.

1713 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1714 GraphBLAS objects (input or output) is in an invalid state caused
1715 by a previous execution error. Call `GrB_error()` to access any error
1716 messages generated by the implementation.

1717 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1718 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS vector, `v`, has not been initialized by a call to
1719 `Vector_new` or `Vector_dup`.

1720 Description

1721 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,
1722 $L(v) = \emptyset$. The size of the vector does not change.

1723 4.2.4.5 Vector_size: Size of a vector

1724 Retrieve the size of a vector.

1725 C Syntax

1726 `GrB_Info GrB_Vector_size(GrB_Index *nsize,`
1727 `const GrB_Vector v);`

1728 **Parameters**

1729 **nsz** (OUT) On successful return, is set to the size of the vector.

1730 **v** (IN) An existing GraphBLAS vector being queried.

1731 **Return Values**

1732 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1733 cessfully and the value of **nsz** has been set.

1734 **GrB_PANIC** Unknown internal error.

1735 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1736 GraphBLAS objects (input or output) is in an invalid state caused
1737 by a previous execution error. Call **GrB_error()** to access any error
1738 messages generated by the implementation.

1739 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to
1740 **Vector_new** or **Vector_dup**.

1741 **GrB_NULL_POINTER** **nsz** pointer is NULL.

1742 **Description**

1743 Return **size(v)** in **nsz**.

1744 **4.2.4.6 Vector_nvals: Number of stored elements in a vector**

1745 Retrieve the number of stored elements (tuples) in a vector.

1746 **C Syntax**

1747 **GrB_Info GrB_Vector_nvals**(**GrB_Index** ***nvals**,
1748 **const GrB_Vector** **v**);

1749 **Parameters**

1750 **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)
1751 in the vector.

1752 **v** (IN) An existing GraphBLAS vector being queried.

1753 Return Values

1754 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1755 cessfully and the value of `nvals` has been set.

1756 GrB_PANIC Unknown internal error.

1757 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1758 GraphBLAS objects (input or output) is in an invalid state caused
1759 by a previous execution error. Call `GrB_error()` to access any error
1760 messages generated by the implementation.

1761 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1762 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to
1763 Vector_new or Vector_dup.

1764 GrB_NULL_POINTER The `nvals` pointer is NULL.

1765 Description

1766 Return `nvals(v)` in `nvals`. This is the number of stored elements in vector `v`, which is the size of
1767 `L(v)` (see Section 3.5.2).

1768 4.2.4.7 Vector_build: Store elements from tuples into a vector

1769 C Syntax

```
1770                   GrB_Info GrB_Vector_build(GrB_Vector                   w,  
1771                                           const GrB_Index               *indices,  
1772                                           const <type>                *values,  
1773                                           GrB_Index                    n,  
1774                                           const GrB_BinaryOp           dup);
```

1775 Parameters

1776 w (INOUT) An existing Vector object to store the result.

1777 indices (IN) Pointer to an array of indices.

1778 values (IN) Pointer to an array of scalars of a type that is compatible with the domain of
1779 vector `w`.

1780 n (IN) The number of entries contained in each array (the same for `indices` and `values`).

1781 **dup** (IN) An associative and commutative binary operator to apply when duplicate
 1782 values for the same location are present in the input arrays. All three domains of
 1783 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
 1784 then duplicate locations will result in an error.

1785 **Return Values**

1786 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 1787 blocking mode, this indicates that the API checks for the input
 1788 arguments passed successfully. Either way, output vector **w** is
 1789 ready to be used in the next method of the sequence.

1790 **GrB_PANIC** Unknown internal error.

1791 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
 1792 opaque GraphBLAS objects (input or output) is in an invalid
 1793 state caused by a previous execution error. Call **GrB_error()** to
 1794 access any error messages generated by the implementation.

1795 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1796 **GrB_UNINITIALIZED_OBJECT** Either **w** has not been initialized by a call to **GrB_Vector_new**
 1797 or by **GrB_Vector_dup**, or **dup** has not been initialized by a call
 1798 to **GrB_BinaryOp_new**.

1799 **GrB_NULL_POINTER** indices or values pointer is **NULL**.

1800 **GrB_INDEX_OUT_OF_BOUNDS** A value in indices is outside the allowed range for **w**.

1801 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
 1802 not all the same, or the domains of **values** and **w** are incompatible
 1803 with each other or D_{dup} .

1804 **GrB_OUTPUT_NOT_EMPTY** Output vector **w** already contains valid tuples (elements). In
 1805 other words, **GrB_Vector_nvals(C)** returns a positive value.

1806 **GrB_INVALID_VALUE** indices contains a duplicate location and **dup** is **GrB_NULL**.

1807 **Description**

1808 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
 1809 from **w** in its domain; otherwise, $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$.

1810 Each tuple $\{\text{indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the output in the form of

$$1811 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \mathbf{dup} \neq \mathbf{GrB_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1812 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,
 1813 `dup` is used to reduce the values before assignment into $\tilde{\mathbf{w}}$ as follows:

$$1814 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1815 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{w}}$ is copied into `w` via typecasting its
 1816 values to $\mathbf{D}(\mathbf{w})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

1817 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1818 It is an error to call this function on an output object with existing elements. In other words,
 1819 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1820 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`
 1821 or `values`.

1822 4.2.4.8 Vector_setElement: Set a single element in a vector

1823 Set one element of a vector to a given value.

1824 C Syntax

```
1825 // scalar value
1826 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1827                               <type>         val,
1828                               GrB_Index       index);
1829
1830 // GraphBLAS scalar
1831 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1832                               const GrB_Scalar s,
1833                               GrB_Index       index);
```

1834 Parameters

1835 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1836 `val` or `s` (IN) Scalar assign. Its domain (type) must be compatible with the domain of `w`.

1837 `index` (IN) The location of the element to be assigned.

1838 Return Values

1839 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 1840 blocking mode, this indicates that the compatibility tests on in-
 1841 dex/dimensions and domains for the input arguments passed suc-

cessfully. Either way, the 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 The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been initialized by a call to a respective constructor.

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

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

Description

First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)** must be compatible with **D(w)**. 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_Vector_setElement** ends and the domain mismatch error listed above is returned.

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

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

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

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

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

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

In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new contents of **w** is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new contents 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.

1875 4.2.4.9 Vector_removeElement: Remove an element from a vector

1876 Remove (annihilate) one stored element from a vector.

1877 C Syntax

```
1878         GrB_Info GrB_Vector_removeElement(GrB_Vector  w,  
1879                                           GrB_Index    index);
```

1880 Parameters

1881 w (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1882 index (IN) The location of the element to be removed.

1883 Return Values

1884 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1885 blocking mode, this indicates that the compatibility tests on in-
1886 dex/dimensions and domains for the input arguments passed suc-
1887 cessfully. Either way, the output vector w is ready to be used in
1888 the next method of the sequence.

1889 GrB_PANIC Unknown internal error.

1890 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1891 GraphBLAS objects (input or output) is in an invalid state caused
1892 by a previous execution error. Call GrB_error() to access any error
1893 messages generated by the implementation.

1894 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1895 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, w, has not been initialized by a call to
1896 Vector_new or Vector_dup.

1897 GrB_INVALID_INDEX index specifies a location that is outside the dimensions of w.

1898 Description

1899 First, the index parameter is checked for a valid value where the following condition must hold:

$$1900 \quad 0 \leq \text{index} < \text{size}(w)$$

1901 If this condition is violated, execution of GrB_Vector_removeElement ends and the invalid index
1902 error listed above is returned.

1903 We are now ready to carry out the removal of a value that may be stored at the location specified
 1904 by `index`. If a value does not exist at the specified location in `w`, no error is reported and the
 1905 operation has no effect on the state of `w`. In either case, the following will be true on return from
 1906 the method: `index` \notin `ind(w)`.

1907 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
 1908 of `w` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
 1909 return value `GrB_SUCCESS` and the new content of vector `w` is as defined above but may not be
 1910 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1911 4.2.4.10 `Vector_extractElement`: Extract a single element from a vector.

1912 Extract one element of a vector into a scalar.

1913 C Syntax

```
1914 // scalar value
1915 GrB_Info GrB_Vector_extractElement(<type>          *val,
1916                                   const GrB_Vector  u,
1917                                   GrB_Index          index);
1918
1919 // GraphBLAS scalar
1920 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1921                                   const GrB_Vector  u,
1922                                   GrB_Index          index);
```

1923 Parameters

1924 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector
 1925 `u`. On successful return, this scalar holds the result of the extract. Any previous
 1926 value stored in `val` or `s` is overwritten.

1927 `u` (IN) The GraphBLAS vector from which an element is extracted.

1928 `index` (IN) The location in `u` to extract.

1929 Return Values

1930 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
 1931 cessfully. This indicates that the compatibility tests on dimensions
 1932 and domains for the input arguments passed successfully, and the
 1933 output scalar, `val` or `s`, has been computed and is ready to be used
 1934 in the next method of the sequence.

1935 GrB_NO_VALUE When using the transparent scalar, `val`, this is returned when there
1936 is no stored value at specified location.

1937 GrB_PANIC Unknown internal error.

1938 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1939 GraphBLAS objects (input or output) is in an invalid state caused
1940 by a previous execution error. Call `GrB_error()` to access any error
1941 messages generated by the implementation.

1942 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1943 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `u`, or scalar, `s`, has not been initialized by
1944 a call to a corresponding constructor.

1945 GrB_NULL_POINTER `val` pointer is NULL.

1946 GrB_INVALID_INDEX `index` specifies a location that is outside the dimensions of `w`.

1947 GrB_DOMAIN_MISMATCH The domains of the vector and scalar are incompatible.

1948 Description

1949 First, the scalar and input vector are tested for domain compatibility as follows: **D**(`val`) or **D**(`s`)
1950 must be compatible with **D**(`u`). Two domains are compatible with each other if values from
1951 one domain can be cast to values in the other domain as per the rules of the C language. In
1952 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1953 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1954 `GrB_Vector_extractElement` ends and the domain mismatch error listed above is returned.

1955 Then, the `index` parameter is checked for a valid value where the following condition must hold:

$$1956 \qquad 0 \leq \text{index} < \text{size}(u)$$

1957 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the invalid index
1958 error listed above is returned.

1959 We are now ready to carry out the extract into the output scalar; that is:

$$1960 \qquad \left. \begin{array}{l} \mathbf{L}(s) \\ \text{val} \end{array} \right\} = u(\text{index})$$

1961 If `index` \in **ind**(`u`), then the corresponding value from `u` is copied into `s` or `val` with casting as
1962 necessary. If `index` \notin **ind**(`u`), then one of the follow occurs depending on output scalar type:

- 1963 • The GraphBLAS scalar, `s`, is cleared and `GrB_SUCCESS` is returned.
- 1964 • The non-opaque scalar, `val`, is unchanged, and `GrB_NO_VALUE` is returned.

1965 When using the non-opaque scalar variant (`val`) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
 1966 mode, the new contents of `val` are as defined above if the method exits with return value `GrB_SUCCESS`
 1967 or `GrB_NO_VALUE`.

1968 When using the GraphBLAS scalar variant (`s`) with a `GrB_SUCCESS` return value, the method
 1969 exits and the new contents of `s` is as defined above and fully computed in `GrB_BLOCKING` mode.
 1970 In `GrB_NONBLOCKING` mode, the new contents of `s` is as defined above but may not be fully
 1971 computed; however, it can be used in the next GraphBLAS method call in a sequence.

1972 4.2.4.11 `Vector_extractTuples`: Extract tuples from a vector

1973 Extract the contents of a GraphBLAS vector into non-opaque data structures.

1974 C Syntax

```
1975      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,
1976                                     <type>          *values,
1977                                     GrB_Index        *n,
1978                                     const GrB_Vector  v);
1979
```

1980 `indices` (OUT) Pointer to an array of indices that is large enough to hold all of the stored
 1981 values' indices.

1982 `values` (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
 1983 the stored values whose type is compatible with `D(v)`.

1984 `n` (INOUT) Pointer to a value indicating (on input) the number of elements the
 1985 `values` and `indices` arrays can hold. Upon return, it will contain the number of
 1986 values written to the arrays.

1987 `v` (IN) An existing GraphBLAS vector.

1988 Return Values

1989 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
 1990 cessfully. This indicates that the compatibility tests on the input
 1991 argument passed successfully, and the output arrays, `indices` and
 1992 `values`, have been computed.

1993 `GrB_PANIC` Unknown internal error.

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

2057 4.2.5.2 Matrix_dup: Construct a copy of a GraphBLAS matrix

2058 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

2059 C Syntax

```
2060         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
2061                                const GrB_Matrix A);
```

2062 Parameters

2063 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2064 matrix.

2065 A (IN) The GraphBLAS matrix to be duplicated.

2066 Return Values

2067 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2068 blocking mode, this indicates that the API checks for the input
2069 arguments passed successfully. Either way, output matrix C is ready
2070 to be used in the next method of the sequence.

2071 GrB_PANIC Unknown internal error.

2072 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2073 GraphBLAS objects (input or output) is in an invalid state caused
2074 by a previous execution error. Call GrB_error() to access any error
2075 messages generated by the implementation.

2076 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2077 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2078 any matrix constructor.

2079 GrB_NULL_POINTER The C pointer is NULL.

2080 Description

2081 Creates a new matrix **C** of domain **D(A)**, size **nrows(A) × ncols(A)**, and contents **L(A)**. It returns
2082 a handle to it in C.

2083 It is not an error to call this method more than once on the same variable; however, the handle to
2084 the previously created object will be overwritten.

2085 4.2.5.3 Matrix_diag: Construct a diagonal GraphBLAS matrix

2086 Creates a new matrix with the same domain and contents as a GrB_Vector, and square dimensions
2087 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

2088 C Syntax

```
2089         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
2090                                 const GrB_Vector  v,  
2091                                 int64_t          k);
```

2092 Parameters

2093 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2094 matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

2095 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the
2096 matrix.

2097 k (IN) The diagonal to which the vector is assigned. k = 0 represents the main
2098 diagonal, k > 0 is above the main diagonal, and k < 0 is below.

2099 Return Values

2100 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2101 blocking mode, this indicates that the API checks for the input
2102 arguments passed successfully. Either way, output matrix C is ready
2103 to be used in the next method of the sequence.

2104 GrB_PANIC Unknown internal error.

2105 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2106 GraphBLAS objects (input or output) is in an invalid state caused
2107 by a previous execution error. Call GrB_error() to access any error
2108 messages generated by the implementation.

2109 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2110 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to
2111 Vector_new or Vector_dup.

2112 GrB_NULL_POINTER The C pointer is NULL.

2113 Description

2114 Creates a new matrix **C** of domain **D(v)**, size $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$, and contents

$$2115 \quad \mathbf{L}(\mathbf{C}) = \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or}$$

$$2116 \quad \mathbf{L}(\mathbf{C}) = \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0.$$

2117 It returns a handle to it in **C**. It is not an error to call this method more than once on the same
2118 variable; however, the handle to the previously created object will be overwritten.

2119 4.2.5.4 Matrix_resize: Resize a matrix

2120 Changes the dimensions of an existing matrix.

2121 C Syntax

```
2122      GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
2123                               GrB_Index  nrows,  
2124                               GrB_Index  ncols);
```

2125 Parameters

2126 **C** (INOUT) An existing Matrix object that is being resized.

2127 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the
2128 current number of rows.

2129 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than
2130 the current number of columns.

2131 Return Values

2132 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2133 blocking mode, this indicates that the API checks for the input
2134 arguments passed successfully. Either way, output matrix **C** is ready
2135 to be used in the next method of the sequence.

2136 **GrB_PANIC** Unknown internal error.

2137 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2138 GraphBLAS objects (input or output) is in an invalid state caused
2139 by a previous execution error. Call **GrB_error()** to access any error
2140 messages generated by the implementation.

2141 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2142 GrB_NULL_POINTER The C pointer is NULL.

2143 GrB_INVALID_VALUE nrows or ncols is zero or outside the range of the type GrB_Index.

2144 Description

2145 Changes the number of rows and columns of C to nrows and ncols, respectively. The domain $\mathbf{D}(\mathbf{C})$
2146 of matrix C remains the same. The contents $\mathbf{L}(\mathbf{C})$ are modified as described below.

2147 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
2148 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
2149 greater than or equal to ncols are dropped.
2150

2151 4.2.5.5 Matrix_clear: Clear a matrix

2152 Removes all elements (tuples) from a matrix.

2153 C Syntax

2154 GrB_Info GrB_Matrix_clear(GrB_Matrix A);

2155 Parameters

2156 A (IN) An existing GraphBLAS matrix to clear.

2157 Return Values

2158 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2159 blocking mode, this indicates that the API checks for the input ar-
2160 guments passed successfully. Either way, output matrix A is ready
2161 to be used in the next method of the sequence.

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_OUT_OF_MEMORY Not enough memory available for operation.

2168 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2169 any matrix constructor.

2170 **Description**

2171 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,
2172 $\mathbf{L}(\mathbf{A}) = \emptyset$. The dimensions of the matrix do not change.

2173 **4.2.5.6 Matrix_nrows: Number of rows in a matrix**

2174 Retrieve the number of rows in a matrix.

2175 **C Syntax**

```
2176         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2177                                   const GrB_Matrix A);
```

2178 **Parameters**

2179 nrows (OUT) On successful return, contains the number of rows in the matrix.

2180 A (IN) An existing GraphBLAS matrix being queried.

2181 **Return Values**

2182 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2183 cessfully and the value of `nrows` has been set.

2184 GrB_PANIC Unknown internal error.

2185 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2186 GraphBLAS objects (input or output) is in an invalid state caused
2187 by a previous execution error. Call `GrB_error()` to access any error
2188 messages generated by the implementation.

2189 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
2190 any matrix constructor.

2191 GrB_NULL_POINTER `nrows` pointer is NULL.

2192 **Description**

2193 Return `nrows(A)` in `nrows` (the number of rows).

2194 **4.2.5.7 Matrix_ncols: Number of columns in a matrix**

2195 Retrieve the number of columns in a matrix.

2196 C Syntax

```
2197         GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2198                                   const GrB_Matrix A);
```

2199 Parameters

2200 ncols (OUT) On successful return, contains the number of columns in 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 ncols 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_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2211 any matrix constructor.

2212 GrB_NULL_POINTER ncols pointer is NULL.

2213 Description

2214 Return **ncols(A)** in ncols (the number of columns).

2215 4.2.5.8 Matrix_nvals: Number of stored elements in a matrix

2216 Retrieve the number of stored elements (tuples) in a matrix.

2217 C Syntax

```
2218         GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2219                                   const GrB_Matrix A);
```

2220 **Parameters**

2221 **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in
2222 the matrix.

2223 **A** (IN) An existing GraphBLAS matrix being queried.

2224 **Return Values**

2225 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2226 cessfully and the value of **nvals** has been set.

2227 **GrB_PANIC** Unknown internal error.

2228 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2229 GraphBLAS objects (input or output) is in an invalid state caused
2230 by a previous execution error. Call **GrB_error()** to access any error
2231 messages generated by the implementation.

2232 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2233 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2234 any matrix constructor.

2235 **GrB_NULL_POINTER** The **nvals** pointer is **NULL**.

2236 **Description**

2237 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of
2238 **L(A)** (see Section 3.5.3).

2239 **4.2.5.9 Matrix_build: Store elements from tuples into a matrix**

2240 **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);
```

2241 **Parameters**

2242 **C** (INOUT) An existing Matrix object to store the result.

2243 **row_indices** (IN) Pointer to an array of row indices.

2244 **col_indices** (IN) Pointer to an array of column indices.

2245 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of
2246 matrix, **C**.

2247 **n** (IN) The number of entries contained in each array (the same for **row_indices**,
2248 **col_indices**, and **values**).

2249 **dup** (IN) An associative and commutative binary operator to apply when duplicate
2250 values for the same location are present in the input arrays. All three domains of
2251 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
2252 then duplicate locations will result in an error.

2253 Return Values

2254 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2255 blocking mode, this indicates that the API checks for the input
2256 arguments passed successfully. Either way, output matrix **C** is
2257 ready to be used in the next method of the sequence.

2258 **GrB_PANIC** Unknown internal error.

2259 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2260 opaque GraphBLAS objects (input or output) is in an invalid
2261 state caused by a previous execution error. Call **GrB_error()** to
2262 access any error messages generated by the implementation.

2263 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2264 **GrB_UNINITIALIZED_OBJECT** Either **C** has not been initialized by a call to any matrix construc-
2265 tor, or **dup** has not been initialized by a call to **GrB_BinaryOp_new**.

2266 **GrB_NULL_POINTER** **row_indices**, **col_indices** or **values** pointer is **NULL**.

2267 **GrB_INDEX_OUT_OF_BOUNDS** A value in **row_indices** or **col_indices** is outside the allowed range
2268 for **C**.

2269 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
2270 not all the same, or the domains of **values** and **C** are incompatible
2271 with each other or D_{dup} .

2272 **GrB_OUTPUT_NOT_EMPTY** Output matrix **C** already contains valid tuples (elements). In
2273 other words, **GrB_Matrix_nvals(C)** returns a positive value.

2274 **GrB_INVALID_VALUE** indices contains a duplicate location and **dup** is **GrB_NULL**.

2275 Description

2276 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
 2277 only differs from \mathbf{C} in its domain; otherwise, $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$.

2278 Each tuple $\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the
 2279 output in the form of

$$2280 \quad \tilde{\mathbf{C}}(\text{row_indices}[k], \text{col_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2281 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,
 2282 `dup` is used to reduce the values before assignment into $\tilde{\mathbf{C}}$ as follows:

$$2283 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row_indices}[k]=i \wedge \text{col_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2284 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{C}}$ is copied into \mathbf{C} via typecasting its
 2285 values to $\mathbf{D}(\mathbf{C})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

2286 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2287 It is an error to call this function on an output object with existing elements. In other words,
 2288 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2289 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,
 2290 `col_indices`, or `values`.

2291 4.2.5.10 Matrix_setElement: Set a single element in matrix

2292 Set one element of a matrix to a given value.

2293 C Syntax

```
2294 // scalar value
2295 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2296                               <type>          val,
2297                               GrB_Index         row_index,
2298                               GrB_Index         col_index);
2299
2300 // GraphBLAS scalar
2301 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2302                               const GrB_Scalar s,
2303                               GrB_Index         row_index,
2304                               GrB_Index         col_index);
```

2305 Parameters

2306 **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.
2307 **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of
2308 **C**.
2309 **row_index** (IN) Row index of element to be assigned
2310 **col_index** (IN) Column index of element to be assigned

2311 Return Values

2312 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2313 blocking mode, this indicates that the compatibility tests on in-
2314 dex/dimensions and domains for the input arguments passed suc-
2315 cessfully. Either way, the output matrix **C** is ready to be used in
2316 the next method of the sequence.
2317 **GrB_PANIC** Unknown internal error.
2318 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2319 GraphBLAS objects (input or output) is in an invalid state caused
2320 by a previous execution error. Call **GrB_error()** to access any error
2321 messages generated by the implementation.
2322 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
2323 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been
2324 initialized by a call to a respective constructor.
2325 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e., not less
2326 than **nrows(C)** or **ncols(C)**, respectively).
2327 **GrB_DOMAIN_MISMATCH** The domains of the matrix and the scalar are incompatible.

2328 Description

2329 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or
2330 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from
2331 one domain can be cast to values in the other domain as per the rules of the C language. In
2332 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
2333 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
2334 **GrB_Matrix_setElement** ends and the domain mismatch error listed above is returned.

2335 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2336 \quad & 0 \leq \text{row_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2337 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid
 2338 index error listed above is returned.

2339 We are now ready to carry out the assignment; that is:

$$2340 \quad C(\text{row_index}, \text{col_index}) = \begin{cases} \mathbf{L}(\mathbf{s}), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

2341 In the case of a transparent scalar or if $\mathbf{L}(\mathbf{s})$ is not empty, then a value will be stored at the
 2342 specified location in \mathbf{C} , overwriting any value that may have been stored there before. In the case
 2343 of a GraphBLAS scalar and if $\mathbf{L}(\mathbf{s})$ is empty, then any value stored at the specified location in \mathbf{C}
 2344 will be removed.

2345 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
 2346 of \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
 2347 return value `GrB_SUCCESS` and the new content of vector \mathbf{C} is as defined above but may not be
 2348 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2349 **4.2.5.11 Matrix_removeElement: Remove an element from a matrix**

2350 Remove (annihilate) one stored element from a matrix.

2351 **C Syntax**

```
2352      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2353                                       GrB_Index   row_index,
2354                                       GrB_Index   col_index);
```

2355 **Parameters**

2356 `C` (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2357 `row_index` (IN) Row index of element to be removed

2358 `col_index` (IN) Column index of element to be removed

2359 **Return Values**

2360 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 2361 blocking mode, this indicates that the compatibility tests on in-
 2362 dex/dimensions and domains for the input arguments passed suc-
 2363 cessfully. Either way, the output matrix \mathbf{C} is ready to be used in
 2364 the next method of the sequence.

2365 `GrB_PANIC` Unknown internal error.

2366 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 2367 GraphBLAS objects (input or output) is in an invalid state caused
 2368 by a previous execution error. Call GrB_error() to access any error
 2369 messages generated by the implementation.

2370 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2371 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to
 2372 any matrix constructor.

2373 GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e., not less
 2374 than **nrows(C)** or **ncols(C)**, respectively).

2375 Description

2376 First, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2377 \quad & 0 \leq \text{row_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2378 If either of these conditions is violated, execution of GrB_Matrix_removeElement ends and the
 2379 invalid index error listed above is returned.

2380 We are now ready to carry out the removal of a value that may be stored at the location specified by
 2381 (row_index, col_index). If a value does not exist at the specified location in C, no error is reported
 2382 and the operation has no effect on the state of C. In either case, the following will be true on return
 2383 from this method: (row_index, col_index) \notin ind(C)

2384 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
 2385 of C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
 2386 return value GrB_SUCCESS and the new content of vector C is as defined above but may not be
 2387 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2388 4.2.5.12 Matrix_extractElement: Extract a single element from a matrix

2389 Extract one element of a matrix into a scalar.

2390 C Syntax

```
2391 // scalar value
2392 GrB_Info GrB_Matrix_extractElement(<type>          *val,
2393                                   const GrB_Matrix A,
2394                                   GrB_Index         row_index,
2395                                   GrB_Index         col_index);
2396
2397 // GraphBLAS scalar
```

```

2398         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2399                                           const GrB_Matrix A,
2400                                           GrB_Index      row_index,
2401                                           GrB_Index      col_index);
2402

```

2403 Parameters

2404 **val or s** (INOUT) An existing scalar whose domain is compatible with the domain of matrix
2405 **A**. On successful return, this scalar holds the result of the extract. Any previous
2406 value stored in **val** or **s** is overwritten.

2407 **A** (IN) The GraphBLAS matrix from which an element is extracted.

2408 **row_index** (IN) The row index of location in **A** to extract.

2409 **col_index** (IN) The column index of location in **A** to extract.

2410 Return Values

2411 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2412 cessfully. This indicates that the compatibility tests on dimensions
2413 and domains for the input arguments passed successfully, and the
2414 output scalar, **val** or **s**, has been computed and is ready to be used
2415 in the next method of the sequence.

2416 **GrB_NO_VALUE** When using the transparent scalar, **val**, this is returned when there
2417 is no stored value at specified location.

2418 **GrB_PANIC** Unknown internal error.

2419 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2420 GraphBLAS objects (input or output) is in an invalid state caused
2421 by a previous execution error. Call **GrB_error()** to access any error
2422 messages generated by the implementation.

2423 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2424 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by
2425 a call to a corresponding constructor.

2426 **GrB_NULL_POINTER** **val** pointer is **NULL**.

2427 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e. less than
2428 zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-
2429 tively).

2430 **GrB_DOMAIN_MISMATCH** The domains of the matrix and scalar are incompatible.

2431 Description

2432 First, the scalar and input matrix are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ or $\mathbf{D}(\mathbf{s})$
 2433 must be compatible with $\mathbf{D}(\mathbf{A})$. Two domains are compatible with each other if values from
 2434 one domain can be cast to values in the other domain as per the rules of the C language. In
 2435 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
 2436 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
 2437 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2438 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2439 \quad & 0 \leq \text{row_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2440 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index
 2441 error listed above is returned.

2442 We are now ready to carry out the extract into the output scalar; that is,

$$2443 \quad \left. \begin{array}{l} \mathbf{L}(\mathbf{s}) \\ \text{val} \end{array} \right\} = \mathbf{A}(\text{row_index}, \text{col_index})$$

2444 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
 2445 with casting as necessary. If $(\text{row_index}, \text{col_index}) \notin \mathbf{ind}(\mathbf{A})$, then one of the follow occurs
 2446 depending on output scalar type:

- 2447 • The GraphBLAS scalar, \mathbf{s} , is cleared and `GrB_SUCCESS` is returned.
- 2448 • The non-opaque scalar, val , is unchanged, and `GrB_NO_VALUE` is returned.

2449 When using the non-opaque scalar variant (val) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
 2450 mode, the new contents of val are as defined above if the method exits with return value `GrB_SUCCESS`
 2451 or `GrB_NO_VALUE`.

2452 When using the GraphBLAS scalar variant (\mathbf{s}) with a `GrB_SUCCESS` return value, the method
 2453 exits and the new contents of \mathbf{s} is as defined above and fully computed in `GrB_BLOCKING` mode.
 2454 In `GrB_NONBLOCKING` mode, the new contents of \mathbf{s} is as defined above but may not be fully
 2455 computed; however, it can be used in the next GraphBLAS method call in a sequence.

2456 4.2.5.13 Matrix_extractTuples: Extract tuples from a matrix

2457 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

2458 C Syntax

```
2459         GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2460                                         GrB_Index      *col_indices,
```

```

2461                                     <type>          *values,
2462                                     GrB_Index         *n,
2463                                     const GrB_Matrix   A);

```

2464 Parameters

2465 **row_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the
2466 row indices.

2467 **col_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the
2468 column indices.

2469 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
2470 the stored values whose type is compatible with $\mathbf{D}(\mathbf{A})$.

2471 **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,
2472 **row_indices**, and **col_indices** arrays can hold. Upon return, it will contain the
2473 number of values written to the arrays.

2474 **A** (IN) An existing GraphBLAS matrix.

2475 Return Values

2476 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2477 cessfully. This indicates that the compatibility tests on the input
2478 argument passed successfully, and the output arrays, **indices** and
2479 **values**, have been computed.

2480 **GrB_PANIC** Unknown internal error.

2481 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2482 GraphBLAS objects (input or output) is in an invalid state caused
2483 by a previous execution error. Call **GrB_error()** to access any error
2484 messages generated by the implementation.

2485 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2486 **GrB_INSUFFICIENT_SPACE** Not enough space in **row_indices**, **col_indices**, and **values** (as indi-
2487 cated by the **n** parameter) to hold all of the tuples that will be
2488 extracted.

2489 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2490 any matrix constructor.

2491 **GrB_NULL_POINTER** **row_indices**, **col_indices**, **values** or **n** pointer is NULL.

2492 **GrB_DOMAIN_MISMATCH** The domains of the **A** matrix and **values** array are incompatible
2493 with one another.

2494 Description

2495 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with
2496 those tuples are placed in the **values** array, the column indices are placed in the **col_indices** array,
2497 and the row indices are placed in the **row_indices** array. These output arrays are pre-allocated by
2498 the user before calling this function such that each output array has enough space to hold at least
2499 **GrB_Matrix_nvals(A)** elements.

2500 Upon return of this function, a pair of $\{\text{row_indices}[k], \text{col_indices}[k]\}$ are unique for every valid
2501 k , but they are not required to be sorted in any particular order. Each tuple (i, j, A_{ij}) in **A** is
2502 unzipped and copied into a distinct k th location in output vectors:

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

2503 where $0 \leq k < \text{GrB_Matrix_nvals}(v)$. No gaps in output vectors are allowed; that is, if **row_indices**[k],
2504 **col_indices**[k] and **values**[k] exist upon return, so does **row_indices**[j], **col_indices**[j] and **values**[j] for
2505 all j such that $0 \leq j < k$.

2506 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,
2507 then a **GrB_INSUFFICIENT_SPACE** error is returned since it is undefined which subset of values
2508 would be extracted.

2509 In both **GrB_BLOCKING** mode **GrB_NONBLOCKING** mode if the method exits with return value
2510 **GrB_SUCCESS**, the new contents of the arrays **row_indices**, **col_indices** and **values** are as defined
2511 above.

2512 4.2.5.14 Matrix_exportHint: Provide a hint as to which storage format might be most 2513 efficient for exporting a matrix

2514 C Syntax

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

2515 Parameters

2516 hint (OUT) Pointer to a value of type **GrB_Format**.

2517 A (IN) A GraphBLAS matrix object.

2518 Return Values

2519 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2520 cessfully and the value of **hint** has been set.

2521 **GrB_PANIC** Unknown internal error.

2522 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
2523 opaque GraphBLAS objects (input or output) is in an invalid
2524 state caused by a previous execution error. Call GrB_error() to
2525 access any error messages generated by the implementation.

2526 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2527 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2528 any matrix constructor.

2529 GrB_NULL_POINTER hint is NULL.

2530 GrB_NO_VALUE If the implementation does not have a preferred format, it may
2531 return the value GrB_NO_VALUE.

2532 Description

2533 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for
2534 exporting the matrix A. GraphBLAS implementations might return the current storage format of
2535 the matrix, or the format to which it could most efficiently be exported. However, implementations
2536 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is
2537 free to refuse to provide a format hint, returning GrB_NO_VALUE.

2538 4.2.5.15 Matrix_exportSize: Return the array sizes necessary to export a GraphBLAS 2539 matrix object

2540 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);

```

2541 Parameters

2542 n_indptr (OUT) Pointer to a value of type GrB_Index.

2543 n_indices (OUT) Pointer to a value of type GrB_Index.

2544 n_values (OUT) Pointer to a value of type GrB_Index.

2545 format (IN) a value indicating the format in which the matrix will be exported, as defined
2546 in Section 3.5.3.1.

2547 A (IN) A GraphBLAS matrix object.

2548 Return Values

2549 GrB_SUCCESS In blocking mode or non-blocking mode, the operation com-
2550 pleted successfully. This indicates that the API checks for the
2551 input arguments passed successfully, and the number of elements
2552 necessary for the export buffers have been written to `n_indptr`,
2553 `n_indices`, and `n_values`, respectively.

2554 GrB_PANIC Unknown internal error.

2555 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
2556 opaque GraphBLAS objects (input or output) is in an invalid
2557 state caused by a previous execution error. Call `GrB_error()` to
2558 access any error messages generated by the implementation.

2559 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2560 GrB_UNINITIALIZED_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to
2561 any matrix constructor.

2562 GrB_NULL_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

2563 Description

2564 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to
2565 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and
2566 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be
2567 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user
2568 as one of the values defined in Section 3.5.3.1.

2569 4.2.5.16 Matrix_export: Export a GraphBLAS matrix to a pre-defined format

2570 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);
```

2571 Parameters

2572 **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-
2573 dices, depending on the value of **format**. It must be large enough to hold at
2574 least **n_indptr** elements of type **GrB_Index**, where **n_indices** was returned from
2575 **GrB_Matrix_exportSize()** method.

2576 **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements
2577 in **values**, depending on the value of **format**. It must be large enough to hold at
2578 least **n_indices** elements of type **GrB_Index**, where **n_indices** was returned from
2579 **GrB_Matrix_exportSize()** method.

2580 **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-
2581 ment must match the type of the values stored in **A**. It must be large enough
2582 to hold at least **n_values** elements of that type, where **n_values** was returned from
2583 **GrB_Matrix_exportSize**.

2584 **n_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**
2585 array can hold. Upon return, it will contain the number of elements written to the
2586 array.

2587 **n_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**
2588 array can hold. Upon return, it will contain the number of elements written to the
2589 array.

2590 **n_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**
2591 array can hold. Upon return, it will contain the number of elements written to the
2592 array.

2593 **format** (IN) a value indicating the format in which the matrix will be exported, as defined
2594 in Section 3.5.3.1.

2595 **A** (IN) A GraphBLAS matrix object.

2596 Return Values

2597 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2598 cessfully. This indicates that the compatibility tests on the input
2599 argument passed successfully, and the output arrays, **indptr**, **in-**
2600 **dices** and **values**, have been computed.

2601 **GrB_PANIC** Unknown internal error.

2602 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2603 opaque GraphBLAS objects (input or output) is in an invalid
2604 state caused by a previous execution error. Call **GrB_error()** to
2605 access any error messages generated by the implementation.

2606 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2631 **nrows** (IN) Integer value holding the number of rows in the matrix.

2632 **ncols** (IN) Integer value holding the number of columns in the matrix.

2633 **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the
2634 value of **format**.

2635 **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending
2636 on the value of **format**.

2637 **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2638 **n_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2639 **n_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2640 **n_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2641 **format** (IN) a value indicating the format of the matrix being imported, as defined in
2642 Section 3.5.3.1.

2643 **Return Values**

2644 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2645 blocking mode, this indicates that the API checks for the input
2646 arguments passed successfully and the input arrays have been
2647 consumed. Either way, output matrix **A** is ready to be used in
2648 the next method of the sequence.

2649 **GrB_PANIC** Unknown internal error.

2650 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2651 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
2652 (needed for user-defined types).

2653 **GrB_NULL_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2654 **GrB_INDEX_OUT_OF_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices
2655 in **A** and or the size of **values**, **n_values**, depending on the value
2656 of **format**.

2657 **GrB_INVALID_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB_Index**.

2658 **GrB_DOMAIN_MISMATCH** The domain given in parameter **d** does not match the element
2659 type of **values**.

2660 Description

2661 Creates a new matrix **A** of domain **D**(d) and dimension **nrows** \times **ncols**. The new GraphBLAS
2662 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.
2663 The method returns a handle to the new matrix in **A**. The structure of the data being imported is
2664 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of
2665 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2666 It is not an error to call this method more than once on the same output matrix; however, the
2667 handle to the previously created object will be overwritten.

2668 4.2.5.18 Matrix_serializeSize: Compute the serialize buffer size

2669 Compute the buffer size (in bytes) necessary to serialize a GrB_Matrix using GrB_Matrix_serialize.

2670 C Syntax

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

2671 Parameters

2672 size (OUT) Pointer to GrB_Index value where size in bytes of serialized object will be
2673 written.

2674 A (IN) A GraphBLAS matrix object.

2675 Return Values

2676 GrB_SUCCESS The operation completed successfully and the value pointed to
2677 by *size has been computed and is ready to use.

2678 GrB_PANIC Unknown internal error.

2679 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2680 GrB_NULL_POINTER size is NULL.

2681 Description

2682 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object A.
2683 Users may then allocate a buffer of size bytes to pass as a parameter to GrB_Matrix_serialize.

2684 **4.2.5.19 Matrix_serialize: Serialize a GraphBLAS matrix.**

2685 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2686 **C Syntax**

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

2687 **Parameters**

2688 **serialized_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be
2689 written.

2690 **serialized_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized_data**.
2691 On output, the number of bytes written to **serialized_data**.

2692 **A** (IN) A GraphBLAS matrix object.

2693 **Return Values**

2694 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2695 cessfully. This indicates that the compatibility tests on the in-
2696 put argument passed successfully, and the output buffer **serial-
2697 ized_data** and **serialized_size**, have been computed and are ready
2698 to use.

2699 **GrB_PANIC** Unknown internal error.

2700 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2701 opaque GraphBLAS objects (input or output) is in an invalid
2702 state caused by a previous execution error. Call **GrB_error()** to
2703 access any error messages generated by the implementation.

2704 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2705 **GrB_NULL_POINTER** **serialized_data** or **serialize_size** is NULL.

2706 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2707 any matrix constructor.

2708 **GrB_INSUFFICIENT_SPACE** The size of the buffer **serialized_data** (provided as an input **seri-
2709 alized_size**) was not large enough.

2710 Description

2711 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,
2712 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must
2713 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the
2714 serialized matrix written to `serialized_data` is provided upon completion as an output written to
2715 `serialized_size`.

2716 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created
2717 with one library implementation is not necessarily valid for deserialization with another implemen-
2718 tation.

2719 4.2.5.20 Matrix_deserialize: Deserialize a GraphBLAS matrix.

2720 Construct a new GraphBLAS matrix from a serialized object.

2721 C Syntax

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

2722 Parameters

2723 A (INOUT) On a successful return, contains a handle to the newly created Graph-
2724 BLAS matrix.

2725 d (IN) the type of the matrix that was serialized in `serialized_data`.

2726 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2727 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

2728 Return Values

2729 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2730 blocking mode, this indicates that the API checks for the input
2731 arguments passed successfully. Either way, output matrix A is
2732 ready to be used in the next method of the sequence.

2733 GrB_PANIC Unknown internal error.

2734 GrB_INVALID_OBJECT This is returned if `serialized_data` is invalid or corrupted.

2735 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2736 GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to GrB_Type_new
2737 (needed for user-defined types).

2738 GrB_NULL_POINTER serialized_data or A is NULL.

2739 GrB_DOMAIN_MISMATCH The type given in d does not match the type of the matrix
2740 serialized in serialized_data.

2741 Description

2742 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object
2743 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The
2744 domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized
2745 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2746 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix
2747 serialized in one library implementation can be deserialized by another.

2748 It is not an error to call this method more than once on the same output matrix; however, the
2749 handle to the previously created object will be overwritten.

2750 4.2.6 Descriptor methods

2751 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-
2752 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

2753 4.2.6.1 Descriptor_new: Create new descriptor

2754 Creates a new (empty or default) descriptor.

2755 C Syntax

2756 GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);

2757 Parameters

2758 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2759 descriptor.

2760 Return Value

2761 GrB_SUCCESS The method completed successfully.

2762 GrB_PANIC unknown internal error.

2763 GrB_OUT_OF_MEMORY not enough memory available for operation.

2764 GrB_NULL_POINTER desc pointer is NULL.

2765 **Description**

2766 Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can
2767 be populated by calls to Descriptor_set.

2768 It is not an error to call this method more than once on the same variable; however, the handle to
2769 the previously created object will be overwritten.

2770 **4.2.6.2 Descriptor_set: Set content of descriptor**

2771 Sets the content for a field for an existing descriptor.

2772 **C Syntax**

```
2773        GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2774                                    GrB_Desc_Field        field,  
2775                                    GrB_Desc_Value        val);
```

2776 **Parameters**

2777 desc (IN) An existing GraphBLAS descriptor to be modified.

2778 field (IN) The field being set.

2779 val (IN) New value for the field being set.

2780 **Return Values**

2781 GrB_SUCCESS operation completed successfully.

2782 GrB_PANIC unknown internal error.

2783 GrB_OUT_OF_MEMORY not enough memory available for operation.

2784 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to new.

2785 GrB_INVALID_VALUE invalid value set on the field, or invalid field.

2786 Description

2787 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor
2788 to set the value associated with that field. Valid values for the `field` parameter include the following:

2789 `GrB_OUTP` refers to the output parameter (result) of the operation.

2790 `GrB_MASK` refers to the mask parameter of the operation.

2791 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2792 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2793 Valid values for the `val` parameter are:

2794 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask
2795 (`GrB_MASK`) parameter.

2796 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-
2797 eter. When combined with `GrB_STRUCTURE`, the complement of the
2798 structure of the mask is used without evaluating the values stored.

2799 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input
2800 matrix parameters only).

2801 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear
2802 the matrix first (or clear the non-masked entries). The default behavior
2803 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`
2804 parameter only.

2805 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,
2806 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).
2807 A value for a given field may be set multiple times but will have no additional effect. Fields that
2808 have no values set result in their default behavior, as defined in Section 3.6.

2809 4.2.7 free: Destroy an object and release its resources

2810 Destroys a previously created GraphBLAS object and releases any resources associated with the
2811 object.

2812 C Syntax

2813 `GrB_Info GrB_free(<GrB_Object> *obj);`

2814 Parameters

2815 obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have
2816 been created by an explicit call to a GraphBLAS constructor. It can be any of the
2817 opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,
2818 binary op, unary op, or type. On successful completion of GrB_free, obj behaves
2819 as an uninitialized object.

2820 Return Values

2821 GrB_SUCCESS operation completed successfully

2822 GrB_PANIC unknown internal error. If this return value is encountered when
2823 in nonblocking mode, the error responsible for the panic condition
2824 could be from any method involved in the computation of the input
2825 object. The GrB_error() method should be called for additional
2826 information.

2827 Description

2828 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime
2829 system. A call to GrB_free frees those resources so they are available for use by other GraphBLAS
2830 objects.

2831 The parameter passed into GrB_free is a handle referencing a GraphBLAS opaque object of a data
2832 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-
2833 structor. The behavior of a program that calls GrB_free on a pre-defined object is implementation
2834 defined.

2835 After the GrB_free method returns, the object referenced by the input handle is destroyed and the
2836 handle has the value GrB_INVALID_HANDLE. The handle can be used in subsequent GraphBLAS
2837 methods but only after the handle has been reinitialized with a call the the appropriate _new or
2838 _dup method.

2839 Note that unlike other GraphBLAS methods, calling GrB_free with an object with an invalid handle
2840 is legal. The system may attempt to free resources that might be associated with that object, if
2841 possible, and return normally.

2842 When using GrB_free it is possible to create a dangling reference to an object. This would occur
2843 when a handle is assigned to a second variable of the same opaque type. This creates two handles
2844 that reference the same object. If GrB_free is called with one of the variables, the object is destroyed
2845 and the handle associated with the other variable no longer references a valid object. This is not an
2846 error condition that the implementation of the GraphBLAS API can be expected to catch, hence
2847 programmers must take care to prevent this situation from occurring.

2848 **4.2.8 wait: Return once an object is either *complete* or *materialized***

2849 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2850 **C Syntax**

2851 `GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);`

2852 **Parameters**

2853 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an
2854 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2855 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2856 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another
2857 thread (completion) or all computing to produce `obj` by all GraphBLAS operations
2858 in its sequence have finished (materialization).

2859 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the
2860 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or
2861 `GrB_MATERIALIZE`.

2862 **Return values**

2863 `GrB_SUCCESS` operation completed successfully.

2864 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-
2865 pletion of pending operations.

2866 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion
2867 of pending operations.

2868 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,
2869 or other constructor, method.

2870 `GrB_PANIC` unknown internal error.

2871 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`
2872 `GrB_MATERIALIZE`.

2873 **Description**

2874 On successful return from `GrB_wait()`, the input object, `obj` is in one of two states depending on
2875 the mode of `GrB_wait`:

- 2876 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program
2877 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another
2878 thread. This result occurs when the mode parameter is set to `GrB_COMPLETE`.
- 2879 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on
2880 behalf of `obj` and error information is available. This result occurs when the mode parameter
2881 is set to `GrB_MATERIALIZE`.

2882 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,
2883 `GrB_wait(obj,mode)` has no effect when called in blocking mode.

2884 In non-blocking mode, the status of any pending method calls, other than those associated with pro-
2885 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to `GrB_wait(obj,mode)`.
2886 Methods in the sequence for `obj`, however, most likely would be impacted by a call to `GrB_wait(obj,mode)`;
2887 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be
2888 finished prior to the return from `GrB_wait(obj,mode)`.

2889 4.2.9 error: Retrieve an error string

2890 Retrieve an error-message about any errors encountered during the processing associated with an
2891 object.

2892 C Syntax

```
2893         GrB_Info GrB_error(const char          **error,
2894                           const GrB_Object      obj);
```

2895 Parameters

2896 `error` (OUT) A pointer to a null-terminated string. The contents of the string are im-
2897 plementation defined.

2898 `obj` (IN) An existing GraphBLAS object. The object must have been created by an
2899 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2900 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2901 or type.

2902 Return value

2903 `GrB_SUCCESS` operation completed successfully.

2904 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,
2905 or other constructor, method.

2906 `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	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

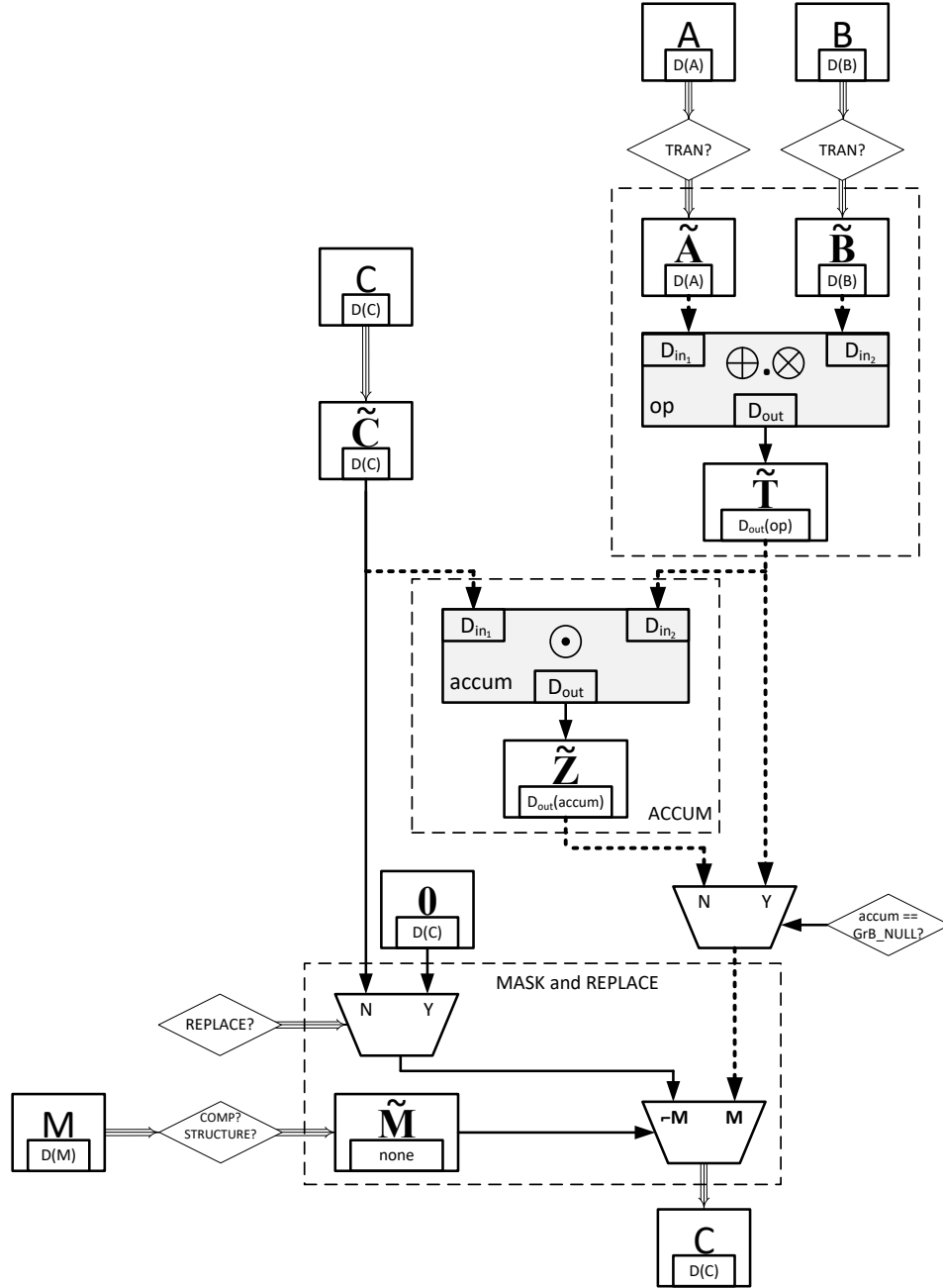


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows (\Rightarrow) denote where “as if copy” takes place (including both collections and descriptor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

2976 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-
 2977 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at
 2978 the point of their declaration, as shown in the following examples:

```
2979         GrB_Type          type = GrB_INVALID_HANDLE;
2980         GrB_Semiring      semiring = GrB_INVALID_HANDLE;
2981         GrB_Matrix        matrix = GrB_INVALID_HANDLE;
```

2982 Compliance

2983 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.
 2984 That is, for each operation we give a recipe for producing its outcome. Any implementation that
 2985 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error
 2986 model (Section 2.6) is a conforming implementation.

2987 4.3.1 mxm: Matrix-matrix multiply

2988 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

2989 C Syntax

```
2990         GrB_Info GrB_mxm(GrB_Matrix          C,
2991                         const GrB_Matrix      Mask,
2992                         const GrB_BinaryOp     accum,
2993                         const GrB_Semiring     op,
2994                         const GrB_Matrix      A,
2995                         const GrB_Matrix      B,
2996                         const GrB_Descriptor   desc);
```

2997 Parameters

2998 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 2999 that may be accumulated with the result of the matrix product. On output, the
 3000 matrix holds the results of the operation.

3001 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 3002 stored into the output matrix C. The mask dimensions must match those of the
 3003 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
 3004 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
 3005 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
 3006 dimensions of C), `GrB_NULL` should be specified.

3007 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
 3008 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 3009 specified.

3010 **op** (IN) The semiring used in the matrix-matrix multiply.

3011 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3012 multiplication.

3013 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3014 multiplication.

3015 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 3016 should be specified. Non-default field/value pairs are listed as follows:
 3017

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.

3019 **Return Values**

3020 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 3021 blocking mode, this indicates that the compatibility tests on di-
 3022 mensions and domains for the input arguments passed successfully.
 3023 Either way, output matrix **C** is ready to be used in the next method
 3024 of the sequence.

3025 **GrB_PANIC** Unknown internal error.

3026 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 3027 GraphBLAS objects (input or output) is in an invalid state caused
 3028 by a previous execution error. Call **GrB_error()** to access any error
 3029 messages generated by the implementation.

3030 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

3031 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 3032 a call to **new** (or **Matrix_dup** for matrix parameters).

3033 **GrB_DIMENSION_MISMATCH** Mask and/or matrix dimensions are incompatible.

3034 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3035 corresponding domains of the semiring or accumulation operator,
 3036 or the mask's domain is not compatible with `bool` (in the case where
 3037 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3038 Description

3039 GrB_mxm computes the matrix product $C = A \oplus . \otimes B$ or, if an optional binary accumulation operator
 3040 (\odot) is provided, $C = C \odot (A \oplus . \otimes B)$ (where matrices A and B can be optionally transposed).
 3041 Logically, this operation occurs in three steps:

3042 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3043 and dimensions are tested for compatibility.

3044 **Compute** The indicated computations are carried out.

3045 **Output** The result is written into the output matrix, possibly under control of a mask.

3046 Up to four argument matrices are used in the GrB_mxm operation:

- 3047 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3048 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)
- 3049 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3050 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3051 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for
 3052 domain compatibility as follows:

- 3053 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3054 must be from one of the pre-defined types of Table 3.2.
- 3055 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 3056 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 3057 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 3058 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3059 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 3060 of the accumulation operator.

3061 Two domains are compatible with each other if values from one domain can be cast to values in
 3062 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3063 all compatible with each other. A domain from a user-defined type is only compatible with itself.

3064 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch
 3065 error listed above is returned.

3066 From the argument matrices, the internal matrices and mask used in the computation are formed
 3067 (\leftarrow denotes copy):

- 3068 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3069 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3070 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3071 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3072 (b) If `Mask \neq GrB_NULL`,
 - 3073 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3074 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3075 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3076 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 3077 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3078 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3079 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3080 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3081 must hold:

- 3082 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 3083 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 3084 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 3085 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.
- 3086 5. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.

3087 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch
 3088 error listed above is returned.

3089 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3090 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3091 We are now ready to carry out the matrix multiplication and any additional associated operations.
 3092 We describe this in terms of two intermediate matrices:

- 3093 • $\tilde{\mathbf{T}}$: The matrix holding the product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3094 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3095 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

$$3097 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3098 where \oplus and \otimes are the additive and multiplicative operators of semiring \mathbf{op} , respectively.

3099 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3100 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3101 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3102 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3103 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
3104 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 3105 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3106 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3107 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3108 \quad & \\ 3109 \end{aligned}$$

3110 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3111 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
3112 using what is called a *standard matrix mask and replace*. This is carried out under control of the
3113 mask which acts as a “write mask”.

- 3114 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
3115 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3116 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3117 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
3118 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
3119 mask are unchanged:

$$3120 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3121 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
3122 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
3123 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
3124 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3125 sequence.

3126 4.3.2 vxm: Vector-matrix multiply

3127 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

3128 C Syntax

```
3129         GrB_Info GrB_vxm(GrB_Vector          w,  
3130                           const GrB_Vector    mask,  
3131                           const GrB_BinaryOp   accum,  
3132                           const GrB_Semiring   op,  
3133                           const GrB_Vector    u,  
3134                           const GrB_Matrix    A,  
3135                           const GrB_Descriptor desc);
```

3136 Parameters

3137 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3138 that may be accumulated with the result of the vector-matrix product. On output,
3139 this vector holds the results of the operation.

3140 **mask** (IN) An optional “write” mask that controls which results from this operation are
3141 stored into the output vector **w**. The mask dimensions must match those of the
3142 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3143 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3144 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3145 dimensions of **w**), **GrB_NULL** should be specified.

3146 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3147 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
3148 specified.

3149 **op** (IN) Semiring used in the vector-matrix multiply.

3150 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3151 multiplication.

3152 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
3153 multiplication.

3154 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
3155 should be specified. Non-default field/value pairs are listed as follows:
3156

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.

3184 **Output** The result is written into the output vector, possibly under control of a mask.

3185 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3186 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3187 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3188 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3189 4. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3190 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
 3191 tested for domain compatibility as follows:

- 3192 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 3193 must be from one of the pre-defined types of Table 3.2.
- 3194 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 3195 3. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 3196 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.
- 3197 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3198 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$
 3199 of the accumulation operator.

3200 Two domains are compatible with each other if values from one domain can be cast to values in
 3201 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3202 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 3203 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch
 3204 error listed above is returned.

3205 From the argument vectors and matrices, the internal matrices and mask used in the computation
 3206 are formed (\leftarrow denotes copy):

- 3207 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3208 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3209 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3210 (b) If `mask` \neq `GrB_NULL`,
 - 3211 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3212 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$.
 - 3213 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3214 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3215 4. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

3216 The internal matrices and masks are checked for shape compatibility. The following conditions
3217 must hold:

3218 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.

3219 2. $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$.

3220 3. $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$.

3221 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch
3222 error listed above is returned.

3223 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3224 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3225 We are now ready to carry out the vector-matrix multiplication and any additional associated
3226 operations. We describe this in terms of two intermediate vectors:

- 3227 • $\tilde{\mathbf{t}}$: The vector holding the product of vector $\tilde{\mathbf{u}}^T$ and matrix $\tilde{\mathbf{A}}$.
- 3228 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3229 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.
3230 The value of each of its elements is computed by

$$3231 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3232 where \oplus and \otimes are the additive and multiplicative operators of semiring `op`, respectively.

3233 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3234 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3235 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3236 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3237 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3238 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3239 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3240 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3241 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3242 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3243 \end{aligned}$$

3244 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3245 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3246 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3247 mask which acts as a “write mask”.

- 3248 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 3249 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3250 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3251 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3252 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3253 mask are unchanged:

$$3254 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3255 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 3256 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 3257 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 3258 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3259 sequence.

3260 4.3.3 mxv: Matrix-vector multiply

3261 Multiplies a matrix by a vector on a semiring. The result is a vector.

3262 C Syntax

```
3263     GrB_Info GrB_mxv(GrB_Vector      w,
3264                     const GrB_Vector mask,
3265                     const GrB_BinaryOp accum,
3266                     const GrB_Semiring op,
3267                     const GrB_Matrix A,
3268                     const GrB_Vector u,
3269                     const GrB_Descriptor desc);
```

3270 Parameters

3271 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 3272 that may be accumulated with the result of the matrix-vector product. On output,
 3273 this vector holds the results of the operation.

3274 **mask** (IN) An optional “write” mask that controls which results from this operation are
 3275 stored into the output vector \mathbf{w} . The mask dimensions must match those of the
 3276 vector \mathbf{w} . If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain

3277 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types
 3278 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
 3279 dimensions of `w`), `GrB_NULL` should be specified.

3280 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`
 3281 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 3282 specified.

3283 `op` (IN) Semiring used in the vector-matrix multiply.

3284 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3285 multiplication.

3286 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3287 multiplication.

3288 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 3289 should be specified. Non-default field/value pairs are listed as follows:
 3290

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.

3292 Return Values

3293 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 3294 blocking mode, this indicates that the compatibility tests on di-
 3295 mensions and domains for the input arguments passed successfully.
 3296 Either way, output vector `w` is ready to be used in the next method
 3297 of the sequence.

3298 `GrB_PANIC` Unknown internal error.

3299 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 3300 GraphBLAS objects (input or output) is in an invalid state caused
 3301 by a previous execution error. Call `GrB_error()` to access any error
 3302 messages generated by the implementation.

3303 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3304 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by
 3305 a call to `new` (or `dup` for matrix or vector parameters).

3306 GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3307 GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with
3308 the corresponding domains of the semiring or accumulation opera-
3309 tor, or the mask's domain is not compatible with **bool** (in the case
3310 where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3311 Description

3312 GrB_mxv computes the matrix-vector product $w = A \oplus . \otimes u$, or, if an optional binary accumulation
3313 operator (\odot) is provided, $w = w \odot (A \oplus . \otimes u)$ (where matrix A can be optionally transposed).
3314 Logically, this operation occurs in three steps:

3315 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
3316 domains/dimensions are tested for compatibility.

3317 **Compute** The indicated computations are carried out.

3318 **Output** The result is written into the output vector, possibly under control of a mask.

3319 Up to four argument vectors or matrices are used in the GrB_mxv operation:

- 3320 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3321 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3322 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3323 4. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3324 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
3325 tested for domain compatibility as follows:

- 3326 1. If **mask** is not GrB_NULL, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3327 must be from one of the pre-defined types of Table 3.2.
- 3328 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 3329 3. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 3330 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 3331 5. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3332 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
3333 of the accumulation operator.

3334 Two domains are compatible with each other if values from one domain can be cast to values in
 3335 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3336 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 3337 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the domain mismatch
 3338 error listed above is returned.

3339 From the argument vectors and matrices, the internal matrices and mask used in the computation
 3340 are formed (\leftarrow denotes copy):

- 3341 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3342 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3343 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3344 (b) If `mask \neq GrB_NULL`,
 - 3345 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3346 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3347 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3348 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3349 4. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3350 The internal matrices and masks are checked for shape compatibility. The following conditions
 3351 must hold:

- 3352 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.
- 3353 2. $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}(\tilde{\mathbf{A}})$.
- 3354 3. $\text{size}(\tilde{\mathbf{u}}) = \text{ncols}(\tilde{\mathbf{A}})$.

3355 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the dimension mismatch
 3356 error listed above is returned.

3357 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3358 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3359 We are now ready to carry out the matrix-vector multiplication and any additional associated
 3360 operations. We describe this in terms of two intermediate vectors:

- 3361 • $\tilde{\mathbf{t}}$: The vector holding the product of matrix $\tilde{\mathbf{A}}$ and vector $\tilde{\mathbf{u}}$.
- 3362 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3363 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.
 3364 The value of each of its elements is computed by

$$3365 \quad t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3366 where \oplus and \otimes are the additive and multiplicative operators of semiring **op**, respectively.
 3367 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3368 • If **accum** = **GrB_NULL**, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3369 • If **accum** is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3370 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3371 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3372 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3373 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3374 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3375 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3376 \end{aligned}$$

3377 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.
 3378

3379 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector **w**,
 3380 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3381 mask which acts as a “write mask”.

- 3382 • If **desc[GrB_OUTP].GrB_REPLACE** is set, then any values in **w** on input to this operation are
 3383 deleted and the content of the new output vector, **w**, is defined as,

$$3384 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3385 • If **desc[GrB_OUTP].GrB_REPLACE** is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3386 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the
 3387 mask are unchanged:

$$3388 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3389 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new content
 3390 of vector **w** is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method
 3391 exits with return value **GrB_SUCCESS** and the new content of vector **w** is as defined above but
 3392 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3393 sequence.

3394 4.3.4 eWiseMult: Element-wise multiplication

3395 **Note:** The difference between **eWiseAdd** and **eWiseMult** is not about the element-wise operation
 3396 but how the index sets are treated. **eWiseAdd** returns an object whose indices are the “union” of
 3397 the indices of the inputs whereas **eWiseMult** returns an object whose indices are the “intersection”
 3398 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
 3399 the set of values from the resulting index set.

3400 4.3.4.1 eWiseMult: Vector variant

3401 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-
3402 ducing a third vector as result.

3403 C Syntax

```
3404     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3405                           const GrB_Vector mask,  
3406                           const GrB_BinaryOp accum,  
3407                           const GrB_Semiring op,  
3408                           const GrB_Vector u,  
3409                           const GrB_Vector v,  
3410                           const GrB_Descriptor desc);  
3411  
3412     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3413                           const GrB_Vector mask,  
3414                           const GrB_BinaryOp accum,  
3415                           const GrB_Monoid op,  
3416                           const GrB_Vector u,  
3417                           const GrB_Vector v,  
3418                           const GrB_Descriptor desc);  
3419  
3420     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3421                           const GrB_Vector mask,  
3422                           const GrB_BinaryOp accum,  
3423                           const GrB_BinaryOp op,  
3424                           const GrB_Vector u,  
3425                           const GrB_Vector v,  
3426                           const GrB_Descriptor desc);
```

3427 Parameters

3428 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3429 that may be accumulated with the result of the element-wise operation. On output,
3430 this vector holds the results of the operation.

3431 **mask** (IN) An optional “write” mask that controls which results from this operation are
3432 stored into the output vector **w**. The mask dimensions must match those of the
3433 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3434 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3435 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3436 dimensions of **w**), **GrB_NULL** should be specified.

3437 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3438 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3439 specified.

3440 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3441 operation. Depending on which type is passed, the following defines the binary
3442 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3443 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3444 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-
3445 nored.

3446 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
3447 is ignored.

3448 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3449 operation.

3450 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
3451 operation.

3452 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
3453 should be specified. Non-default field/value pairs are listed as follows:
3454

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.

3456 Return Values

3457 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3458 blocking mode, this indicates that the compatibility tests on di-
3459 mensions and domains for the input arguments passed successfully.
3460 Either way, output vector w is ready to be used in the next method
3461 of the sequence.

3462 GrB_PANIC Unknown internal error.

3463 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3464 GraphBLAS objects (input or output) is in an invalid state caused
3465 by a previous execution error. Call GrB_error() to access any error
3466 messages generated by the implementation.

3467 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3468 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3469 a call to `new` (or `dup` for vector parameters).

3470 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3471 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 3472 responding domains of the binary operator (`op`) or accumulation
 3473 operator, or the mask's domain is not compatible with `bool` (in the
 3474 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3475 Description

3476 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:
 3477 $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$.
 3478 Logically, this operation occurs in three steps:

3479 **Setup** The internal vectors and mask used in the computation are formed and their domains
 3480 and dimensions are tested for compatibility.

3481 **Compute** The indicated computations are carried out.

3482 **Output** The result is written into the output vector, possibly under control of a mask.

3483 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3484 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3485 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3486 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3487 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3488 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)
 3489 are tested for domain compatibility as follows:

- 3490 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 3491 must be from one of the pre-defined types of Table 3.2.
- 3492 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 3493 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 3494 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3495 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3496 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
 3497 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_eWiseMult` ends and the domain mismatch error listed above is returned.

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

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

The internal vectors and mask are checked for dimension compatibility. The following conditions must hold:

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

If any compatibility rule above is violated, execution of `GrB_eWiseMult` 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 element-wise “product” and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$: The vector holding the element-wise “product” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

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 value of each of its elements is computed by:

$$t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))$$

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

3529 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3530 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

3531
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3532 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3533 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

3534
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3535

3536
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3537

3538
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3539 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3540 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3541 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3542 mask which acts as a “write mask”.

3543 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in \mathbf{w} on input to this operation are
 3544 deleted and the content of the new output vector, \mathbf{w} , is defined as,

3545
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3546 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3547 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3548 mask are unchanged:

3549
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3550 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new content
 3551 of vector \mathbf{w} is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method
 3552 exits with return value **GrB_SUCCESS** and the new content of vector \mathbf{w} is as defined above but
 3553 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3554 sequence.

3555 4.3.4.2 eWiseMult: Matrix variant

3556 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-
 3557 ducing a third matrix as result.

3558 C Syntax

```

3559         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3560                                const GrB_Matrix Mask,
3561                                const GrB_BinaryOp accum,
3562                                const GrB_Semiring op,
3563                                const GrB_Matrix A,
3564                                const GrB_Matrix B,
3565                                const GrB_Descriptor desc);
3566
3567         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3568                                const GrB_Matrix Mask,
3569                                const GrB_BinaryOp accum,
3570                                const GrB_Monoid op,
3571                                const GrB_Matrix A,
3572                                const GrB_Matrix B,
3573                                const GrB_Descriptor desc);
3574
3575         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3576                                const GrB_Matrix Mask,
3577                                const GrB_BinaryOp accum,
3578                                const GrB_BinaryOp op,
3579                                const GrB_Matrix A,
3580                                const GrB_Matrix B,
3581                                const GrB_Descriptor desc);

```

3582 Parameters

3583 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3584 that may be accumulated with the result of the element-wise operation. On output,
3585 the matrix holds the results of the operation.

3586 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3587 stored into the output matrix C. The mask dimensions must match those of the
3588 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3589 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types
3590 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
3591 dimensions of C), `GrB_NULL` should be specified.

3592 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3593 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3594 specified.

3595 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3596 operation. Depending on which type is passed, the following defines the binary
3597 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3598 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 3599 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-
 3600 nored.
 3601 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
 3602 is ignored.

3603 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3604 operation.

3605 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3606 operation.

3607 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3608 should be specified. Non-default field/value pairs are listed as follows:
 3609

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.

3611 Return Values

3612 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3613 blocking mode, this indicates that the compatibility tests on di-
 3614 mensions and domains for the input arguments passed successfully.
 3615 Either way, output matrix C is ready to be used in the next method
 3616 of the sequence.

3617 GrB_PANIC Unknown internal error.

3618 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3619 GraphBLAS objects (input or output) is in an invalid state caused
 3620 by a previous execution error. Call GrB_error() to access any error
 3621 messages generated by the implementation.

3622 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3623 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3624 a call to new (or Matrix_dup for matrix parameters).

3625 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3626 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3627 corresponding domains of the binary operator (\otimes) or accumulation
 3628 operator, or the mask's domain is not compatible with `bool` (in the
 3629 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3630 Description

3631 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:
 3632 $C = A \otimes B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \otimes B)$.
 3633 Logically, this operation occurs in three steps:

3634 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3635 and dimensions are tested for compatibility.

3636 **Compute** The indicated computations are carried out.

3637 **Output** The result is written into the output matrix, possibly under control of a mask.

3638 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3639 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3640 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)
- 3641 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3642 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3643 The argument matrices, the “product” operator (\otimes), and the accumulation operator (if provided)
 3644 are tested for domain compatibility as follows:

- 3645 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3646 must be from one of the pre-defined types of Table 3.2.
- 3647 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\otimes)$.
- 3648 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\otimes)$.
- 3649 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\otimes)$.
- 3650 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3651 of the accumulation operator and $\mathbf{D}_{out}(\otimes)$ of \otimes must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3652 the accumulation operator.

3653 Two domains are compatible with each other if values from one domain can be cast to values in
 3654 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3655 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3656 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3657 error listed above is returned.

3658 From the argument matrices, the internal matrices and mask used in the computation are formed
 3659 (\leftarrow denotes copy):

- 3660 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3661 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3662 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3663 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3664 (b) If `Mask \neq GrB_NULL`,
 - 3665 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3666 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3667 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3668 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$.
 - 3669 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3670 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3671 4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3672 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3673 must hold:

- 3674 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3675 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3676 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3677 mismatch error listed above is returned.

3678 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3679 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3680 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3681 We describe this in terms of two intermediate matrices:

- 3682 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise product of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3683 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3684 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$
 3685 is created. The value of each of its elements is computed by

$$3686 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3687 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

3688 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

3689 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3690 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3691 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3692 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3693 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3694 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3695 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3696 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3697 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 3700 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 3701 mask which acts as a “write mask”.

3702 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 3703 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3704 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3705 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 3706 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 3707 mask are unchanged:

$$3708 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3709 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3710 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3711 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 3712 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3713 sequence.

3714 4.3.5 eWiseAdd: Element-wise addition

3715 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation
 3716 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of
 3717 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”
 3718 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
 3719 the set of values from the resulting index set.

3720 4.3.5.1 eWiseAdd: Vector variant

3721 Perform element-wise (general) addition on the elements of two vectors, producing a third vector
3722 as result.

3723 C Syntax

```
3724     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3725                          const GrB_Vector mask,  
3726                          const GrB_BinaryOp accum,  
3727                          const GrB_Semiring op,  
3728                          const GrB_Vector u,  
3729                          const GrB_Vector v,  
3730                          const GrB_Descriptor desc);  
3731  
3732     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3733                          const GrB_Vector mask,  
3734                          const GrB_BinaryOp accum,  
3735                          const GrB_Monoid op,  
3736                          const GrB_Vector u,  
3737                          const GrB_Vector v,  
3738                          const GrB_Descriptor desc);  
3739  
3740     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3741                          const GrB_Vector mask,  
3742                          const GrB_BinaryOp accum,  
3743                          const GrB_BinaryOp op,  
3744                          const GrB_Vector u,  
3745                          const GrB_Vector v,  
3746                          const GrB_Descriptor desc);
```

3747 Parameters

3748 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3749 that may be accumulated with the result of the element-wise operation. On output,
3750 this vector holds the results of the operation.

3751 **mask** (IN) An optional “write” mask that controls which results from this operation are
3752 stored into the output vector **w**. The mask dimensions must match those of the
3753 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3754 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3755 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3756 dimensions of **w**), **GrB_NULL** should be specified.

3757 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3758 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 3759 specified.

3760 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
 3761 operation. Depending on which type is passed, the following defines the binary
 3762 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3763 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3764 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-
 3765 nored.

3766 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-
 3767 nary op and additive identity are ignored.

3768 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 3769 operation.

3770 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3771 operation.

3772 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3773 should be specified. Non-default field/value pairs are listed as follows:
 3774

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.

3776 Return Values

3777 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3778 blocking mode, this indicates that the compatibility tests on di-
 3779 mensions and domains for the input arguments passed successfully.
 3780 Either way, output vector w is ready to be used in the next method
 3781 of the sequence.

3782 GrB_PANIC Unknown internal error.

3783 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3784 GraphBLAS objects (input or output) is in an invalid state caused
 3785 by a previous execution error. Call GrB_error() to access any error
 3786 messages generated by the implementation.

3787 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3788 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3789 a call to `new` (or `dup` for vector parameters).

3790 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3791 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
3792 responding domains of the binary operator (`op`) or accumulation
3793 operator, or the mask's domain is not compatible with `bool` (in the
3794 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3795 Description

3796 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors: $w =$
3797 $u \oplus v$, or, if an optional binary accumulation operator (\odot) is provided, $w = w \odot (u \oplus v)$. Logically,
3798 this operation occurs in three steps:

3799 **Setup** The internal vectors and mask used in the computation are formed and their domains
3800 and dimensions are tested for compatibility.

3801 **Compute** The indicated computations are carried out.

3802 **Output** The result is written into the output vector, possibly under control of a mask.

3803 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3804 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3805 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3806 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3807 4. $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3808 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are
3809 tested for domain compatibility as follows:

- 3810 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3811 must be from one of the pre-defined types of Table 3.2.
- 3812 2. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3813 3. $\mathbf{D}(v)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3814 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3815 5. $\mathbf{D}(u)$ and $\mathbf{D}(v)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3816 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3817 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
3818 the accumulation operator.

3819 Two domains are compatible with each other if values from one domain can be cast to values in
 3820 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3821 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3822 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch
 3823 error listed above is returned.

3824 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3825 denotes copy):

- 3826 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3827 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3828 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3829 (b) If `mask \neq GrB_NULL`,
 - 3830 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3831 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3832 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3833 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3834 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3835 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3836 must hold:

- 3837 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$.

3838 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 3839 mismatch error listed above is returned.

3840 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3841 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3842 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3843 We describe this in terms of two intermediate vectors:

- 3844 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “sum” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3845 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3846 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
 3847 value of each of its elements is computed by:

$$\begin{aligned}
 3848 \quad t_i &= (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})) \\
 3849 \quad t_i &= \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))) \\
 3850
 \end{aligned}$$

3851
3852

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3853

where the difference operator in the previous expressions refers to set difference.

3854

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

3855

- If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3856

- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

3857

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

3858

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 $\tilde{\mathbf{t}}$.

3859

3860

$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

3861

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3862

3863

3864

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3865

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

3866

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”.

3867

3868

3869

- 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,

3870

3871

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

3872

- 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:

3873

3874

3875

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

3876

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.

3877

3878

3879

3880

3881

4.3.5.2 eWiseAdd: Matrix variant

3882

Perform element-wise (general) addition on the elements of two matrices, producing a third matrix as result.

3883

3884 C Syntax

```

3885     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3886                           const GrB_Matrix Mask,
3887                           const GrB_BinaryOp accum,
3888                           const GrB_Semiring op,
3889                           const GrB_Matrix A,
3890                           const GrB_Matrix B,
3891                           const GrB_Descriptor desc);
3892
3893     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3894                           const GrB_Matrix Mask,
3895                           const GrB_BinaryOp accum,
3896                           const GrB_Monoid op,
3897                           const GrB_Matrix A,
3898                           const GrB_Matrix B,
3899                           const GrB_Descriptor desc);
3900
3901     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3902                           const GrB_Matrix Mask,
3903                           const GrB_BinaryOp accum,
3904                           const GrB_BinaryOp op,
3905                           const GrB_Matrix A,
3906                           const GrB_Matrix B,
3907                           const GrB_Descriptor desc);

```

3908 Parameters

3909 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3910 that may be accumulated with the result of the element-wise operation. On output,
3911 the matrix holds the results of the operation.

3912 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3913 stored into the output matrix C. The mask dimensions must match those of the
3914 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3915 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types
3916 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
3917 dimensions of C), `GrB_NULL` should be specified.

3918 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3919 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3920 specified.

3921 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
3922 operation. Depending on which type is passed, the following defines the binary
3923 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3924
3925
3926
3927
3928

3929
3930

3931
3932

3933
3934
3935

3936

3937

3938
3939
3940
3941
3942

3943

3944
3945
3946
3947

3948

3949
3950

3951

BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ignored.
 Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus(\text{op}) \rangle$; the multiplicative binary op and additive identity are ignored.

A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the operation.

B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the operation.

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

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for the operation.

Return Values

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

GrB_PANIC Unknown internal error.

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

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

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

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3952 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3953 corresponding domains of the binary operator (op) or accumulation
 3954 operator, or the mask's domain is not compatible with `bool` (in the
 3955 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3956 Description

3957 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:
 3958 $C = A \oplus B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \oplus B)$.
 3959 Logically, this operation occurs in three steps:

3960 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3961 and dimensions are tested for compatibility.

3962 **Compute** The indicated computations are carried out.

3963 **Output** The result is written into the output matrix, possibly under control of a mask.

3964 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3965 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3966 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)
- 3967 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3968 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3969 The argument matrices, the “sum” operator (op), and the accumulation operator (if provided) are
 3970 tested for domain compatibility as follows:

- 3971 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3972 must be from one of the pre-defined types of Table 3.2.
- 3973 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3974 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3975 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3976 5. $\mathbf{D}(A)$ and $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3977 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3978 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3979 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_eWiseAdd` 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 $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

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

1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

If any compatibility rule above is violated, execution of `GrB_eWiseAdd` 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 element-wise “sum” and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\tilde{\mathbf{T}}$: The matrix holding the element-wise sum of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4011 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$
 4012 is created. The value of each of its elements is computed by

$$\begin{aligned} 4013 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\ 4014 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\ 4015 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \end{aligned}$$

4018 where the difference operator in the previous expressions refers to set difference.

4019 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4020 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 4021 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4022 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4023 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4024 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 4025 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4026 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4027 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \end{aligned}$$

4030 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4031 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4032 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4033 mask which acts as a “write mask”.

- 4034 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 4035 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4036 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4037 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 4038 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 4039 mask are unchanged:

$$4040 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4041 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4042 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4043 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 4044 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4045 sequence.

4046 4.3.6 extract: Selecting sub-graphs

4047 Extract a subset of a matrix or vector.

4048 4.3.6.1 extract: Standard vector variant

4049 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector
4050 whose size is equal to the number of indices.

4051 C Syntax

```
4052         GrB_Info GrB_extract(GrB_Vector          w,  
4053                             const GrB_Vector    mask,  
4054                             const GrB_BinaryOp   accum,  
4055                             const GrB_Vector    u,  
4056                             const GrB_Index     *indices,  
4057                             GrB_Index           nindices,  
4058                             const GrB_Descriptor desc);
```

4059 Parameters

4060 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4061 that may be accumulated with the result of the extract operation. On output, this
4062 vector holds the results of the operation.

4063 **mask** (IN) An optional “write” mask that controls which results from this operation are
4064 stored into the output vector **w**. The mask dimensions must match those of the
4065 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4066 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4067 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4068 dimensions of **w**), **GrB_NULL** should be specified.

4069 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4070 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4071 specified.

4072 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4073 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of
4074 elements from **u** that are extracted. If all elements of **u** are to be extracted in order
4075 from 0 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution
4076 mode and return value, this array may be manipulated by the caller after this
4077 operation returns without affecting any deferred computations for this operation.

4078 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4079 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4080 should be specified. Non-default field/value pairs are listed as follows:

4081

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 .

4082

4083 Return Values

4084 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 4085 blocking mode, this indicates that the compatibility tests on
 4086 dimensions and domains for the input arguments passed suc-
 4087 cessfully. Either way, output vector **w** is ready to be used in the
 4088 next method of the sequence.

4089 GrB_PANIC Unknown internal error.

4090 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 4091 opaque GraphBLAS objects (input or output) is in an invalid
 4092 state caused by a previous execution error. Call GrB_error() to
 4093 access any error messages generated by the implementation.

4094 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4095 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
 4096 by a call to **new** (or **dup** for vector parameters).

4097 GrB_INDEX_OUT_OF_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-
 4098 blocking mode, this error can be deferred.

4099 GrB_DIMENSION_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices** \neq **size(w)**.

4100 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each
 4101 other or the corresponding domains of the accumulation oper-
 4102 ator, or the mask's domain is not compatible with **bool** (in the
 4103 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4104 GrB_NULL_POINTER Argument **row_indices** is a NULL pointer.

4105 Description

4106 This variant of GrB_extract computes the result of extracting a subset of locations from a Graph-
 4107 BLAS vector in a specific order: **w** = **u(indices)**; or, if an optional binary accumulation operator

4108 (\odot) is provided, $w = w \odot u(\text{indices})$. More explicitly:

$$4109 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4110 Logically, this operation occurs in three steps:

4111 **Setup** The internal vectors and mask used in the computation are formed and their domains
4112 and dimensions are tested for compatibility.

4113 **Compute** The indicated computations are carried out.

4114 **Output** The result is written into the output vector, possibly under control of a mask.

4115 Up to three argument vectors are used in this GrB_extract operation:

- 4116 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4117 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4118 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4119 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4120 bility as follows:

- 4121 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
4122 must be from one of the pre-defined types of Table 3.2.
- 4123 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(u)$.
- 4124 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4125 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4126 mulation operator.

4127 Two domains are compatible with each other if values from one domain can be cast to values in
4128 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4129 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4130 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
4131 error listed above is returned.

4132 From the arguments, the internal vectors, mask, and index array used in the computation are
4133 formed (\leftarrow denotes copy):

- 4134 1. Vector $\tilde{w} \leftarrow w$.
- 4135 2. One-dimensional mask, \tilde{m} , is computed from argument **mask** as follows:
4136 (a) If **mask** = GrB_NULL, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.

- 4137 (b) If $\text{mask} \neq \text{GrB_NULL}$,
- 4138 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
- 4139 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 4140 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
- 4141 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 4142 4. The internal index array, $\widetilde{\mathbf{I}}$, is computed from argument indices as follows:
- 4143 (a) If $\text{indices} = \text{GrB_ALL}$, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.
- 4144 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

4145 The internal vectors and mask are checked for dimension compatibility. The following conditions
4146 must hold:

- 4147 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4148 2. $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$.

4149 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
4150 match error listed above is returned.

4151 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4152 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4153 We are now ready to carry out the extract and any additional associated operations. We describe
4154 this in terms of two intermediate vectors:

- 4155 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{w}}$.
- 4156 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4157 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4158 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4159 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
4160 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`
4161 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4162 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4163 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 4164 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 4165 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$4166 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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.

4233 Return Values

4234	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
4235		blocking mode, this indicates that the compatibility tests on
4236		dimensions and domains for the input arguments passed suc-
4237		cessfully. Either way, output matrix C is ready to be used in the
4238		next method of the sequence.
4239	GrB_PANIC	Unknown internal error.
4240	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
4241		opaque GraphBLAS objects (input or output) is in an invalid
4242		state caused by a previous execution error. Call GrB_error() to
4243		access any error messages generated by the implementation.
4244	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
4245	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
4246		by a call to new (or Matrix_dup for matrix parameters).
4247	GrB_INDEX_OUT_OF_BOUNDS	A value in row_indices is greater than or equal to nrows(A) , or
4248		a value in col_indices is greater than or equal to ncols(A) . In
4249		non-blocking mode, this error can be deferred.
4250	GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, nrows \neq nrows(C) , or
4251		ncols \neq ncols(C) .
4252	GrB_DOMAIN_MISMATCH	The domains of the various matrices are incompatible with each
4253		other or the corresponding domains of the accumulation oper-
4254		ator, or the mask's domain is not compatible with bool (in the
4255		case where desc[GrB_MASK].GrB_STRUCTURE is not set).
4256	GrB_NULL_POINTER	Either argument row_indices is a NULL pointer, argument col_indices
4257		is a NULL pointer, or both.

4258 Description

4259 This variant of **GrB_extract** computes the result of extracting a subset of locations from specified
 4260 rows and columns of a GraphBLAS matrix in a specific order: $\mathbf{C} = \mathbf{A}(\text{row_indices}, \text{col_indices})$; or,
 4261 if an optional binary accumulation operator (\odot) is provided, $\mathbf{C} = \mathbf{C} \odot \mathbf{A}(\text{row_indices}, \text{col_indices})$.
 4262 More explicitly (not accounting for an optional transpose of **A**):

$$\begin{aligned}
 &\mathbf{C}(i, j) = \mathbf{A}(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &\mathbf{C}(i, j) = \mathbf{C}(i, j) \odot \mathbf{A}(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4264 Logically, this operation occurs in three steps:

4265 **Setup** The internal matrices and mask used in the computation are formed and their domains
 4266 and dimensions are tested for compatibility.

4267 **Compute** The indicated computations are carried out.

4268 **Output** The result is written into the output matrix, possibly under control of a mask.

4269 Up to three argument matrices are used in the GrB_extract operation:

- 4270 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
4271 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)
4272 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4273 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
4274 ibility as follows:

- 4275 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
4276 must be from one of the pre-defined types of Table 3.2.
4277 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
4278 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4279 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4280 mulation operator.

4281 Two domains are compatible with each other if values from one domain can be cast to values in
4282 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4283 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4284 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
4285 error listed above is returned.

4286 From the arguments, the internal matrices, mask, and index arrays used in the computation are
4287 formed (\leftarrow denotes copy):

- 4288 1. Matrix $\tilde{C} \leftarrow C$.
4289 2. Two-dimensional mask, \tilde{M} , is computed from argument Mask as follows:
4290 (a) If Mask = GrB_NULL, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
4291 $j < \mathbf{ncols}(C)\} \rangle$.
4292 (b) If Mask \neq GrB_NULL,
4293 i. If desc[GrB_MASK].GrB_STRUCTURE is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
4294 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
4295 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
4296 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
4297 (c) If desc[GrB_MASK].GrB_COMP is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
4298 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

- 4299 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4300 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 4301 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \text{nrows}$.
- 4302 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4303 (a) If `col_indices` = `GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$.
- 4304 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \text{ncols}$.

4305 The internal matrices and mask are checked for dimension compatibility. The following conditions
4306 must hold:

- 4307 1. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$.
- 4308 2. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$.
- 4309 3. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$.
- 4310 4. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$.

4311 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
4312 match error listed above is returned.

4313 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4314 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4315 We are now ready to carry out the extract and any additional associated operations. We describe
4316 this in terms of two intermediate matrices:

- 4317 • $\tilde{\mathbf{T}}$: The matrix holding the extraction from $\tilde{\mathbf{A}}$.
- 4318 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4319 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

4320
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{ (i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}}) \} \rangle.$$

4321 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 $\tilde{\mathbf{J}}$
4322 array is not in the range $[0, \text{ncols}(\tilde{\mathbf{A}}))$, the execution of `GrB_extract` ends and the index out-of-
4323 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
4324 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
4325 this point forward in the sequence.

4326 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4327 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

- If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

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

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.

4.3.6.3 extract: Column (and row) variant

Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the source matrix, elements of an arbitrary row of the matrix can be extracted with this function as well.

4357 C Syntax

```
4358      GrB_Info GrB_extract(GrB_Vector      w,  
4359                          const GrB_Vector mask,  
4360                          const GrB_BinaryOp accum,  
4361                          const GrB_Matrix  A,  
4362                          const GrB_Index  *row_indices,  
4363                          GrB_Index        nrows,  
4364                          GrB_Index        col_index,  
4365                          const GrB_Descriptor desc);
```

4366 Parameters

4367 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4368 that may be accumulated with the result of the extract operation. On output, this
4369 vector holds the results of the operation.

4370 **mask** (IN) An optional “write” mask that controls which results from this operation are
4371 stored into the output vector **w**. The mask dimensions must match those of the
4372 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4373 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4374 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4375 dimensions of **w**), **GrB_NULL** should be specified.

4376 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4377 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4378 specified.

4379 **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4380 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations
4381 within the specified column of **A** from which elements are extracted. If elements in
4382 all rows of **A** are to be extracted in order, **GrB_ALL** should be specified. Regardless
4383 of execution mode and return value, this array may be manipulated by the caller
4384 after this operation returns without affecting any deferred computations for this
4385 operation.

4386 **nrows** (IN) The number of indices in the **row_indices** array. Must be equal to **size(w)**.

4387 **col_index** (IN) The index of the column of **A** from which to extract values. It must be in the
4388 range $[0, \mathbf{ncols}(A))$.

4389 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4390 should be specified. Non-default field/value pairs are listed as follows:
4391

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

4419 an optional binary accumulation operator (\odot) is provided, $w = w \odot A(:, \text{col_index})(\text{row_indices})$.
 4420 More explicitly:

$$4421 \quad \begin{aligned} w(i) &= A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4422 Logically, this operation occurs in three steps:

4423 **Setup** The internal matrices, vectors, and mask used in the computation are formed and their
 4424 domains and dimensions are tested for compatibility.

4425 **Compute** The indicated computations are carried out.

4426 **Output** The result is written into the output vector, possibly under control of a mask.

4427 Up to three argument vectors and matrices are used in this GrB_extract operation:

- 4428 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4429 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4430 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4431 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain
 4432 compatibility as follows:

- 4433 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
 4434 must be from one of the pre-defined types of Table 3.2.
- 4435 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(A)$.
- 4436 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4437 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4438 mulation operator.

4439 Two domains are compatible with each other if values from one domain can be cast to values in
 4440 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4441 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4442 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
 4443 error listed above is returned.

4444 From the arguments, the internal vector, matrix, mask, and index array used in the computation
 4445 are formed (\leftarrow denotes copy):

- 4446 1. Vector $\tilde{w} \leftarrow w$.
- 4447 2. One-dimensional mask, \tilde{m} , is computed from argument **mask** as follows:
 4448 (a) If **mask** = GrB_NULL, then $\tilde{m} = \langle \mathbf{size}(w), \{i, \forall i : 0 \leq i < \mathbf{size}(w)\} \rangle$.

4449 (b) If $\text{mask} \neq \text{GrB_NULL}$,
 4450 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 4451 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 4452 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
 4453 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
 4454 4. The internal row index array, $\widetilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
 4455 (a) If `indices = GrB_ALL`, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
 4456 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$.

4457 The internal vector, `mask`, and index array are checked for dimension compatibility. The following
 4458 conditions must hold:

- 4459 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4460 2. $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$.

4461 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
 4462 match error listed above is returned.

4463 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4464 1. $0 \leq \text{col_index} < \text{ncols}(\mathbf{A})$

4465 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above
 4466 is returned.

4467 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4468 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4469 We are now ready to carry out the extract and any additional associated operations. We describe
 4470 this in terms of two intermediate vectors:

- 4471 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from a column of $\widetilde{\mathbf{A}}$.
- 4472 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4473 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4474 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4475 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`
 4476 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,
 4477 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result
 4478 vector, \mathbf{w} , is invalid from this point forward in the sequence.

4479 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

4480 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

4481 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4482 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4483 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4484 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4485 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

$$4486 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4487 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4490 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4491 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 4492 using what is called a *standard vector mask and replace*. This is carried out under control of the
 4493 mask which acts as a “write mask”.

4494 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 4495 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4496 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4497 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 4498 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 4499 mask are unchanged:

$$4500 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4501 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4502 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4503 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
 4504 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4505 sequence.

4506 4.3.7 assign: Modifying sub-graphs

4507 Assign the contents of a subset of a matrix or vector.

4508 4.3.7.1 assign: Standard vector variant

4509 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.
 4510 The size of the input vector is the same size as the index array provided.

4511 C Syntax

```
4512         GrB_Info GrB_assign(GrB_Vector      w,  
4513                             const GrB_Vector mask,  
4514                             const GrB_BinaryOp accum,  
4515                             const GrB_Vector u,  
4516                             const GrB_Index *indices,  
4517                             GrB_Index      nindices,  
4518                             const GrB_Descriptor desc);
```

4519 Parameters

4520 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4521 that may be accumulated with the result of the assign operation. On output, this
4522 vector holds the results of the operation.

4523 **mask** (IN) An optional “write” mask that controls which results from this operation are
4524 stored into the output vector **w**. The mask dimensions must match those of the
4525 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4526 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4527 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4528 dimensions of **w**), **GrB_NULL** should be specified.

4529 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4530 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4531 specified.

4532 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4533 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4534 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0
4535 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4536 and return value, this array may be manipulated by the caller after this operation
4537 returns without affecting any deferred computations for this operation. If this
4538 array contains duplicate values, it implies in assignment of more than one value to
4539 the same location which leads to undefined results.

4540 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4541 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4542 should be specified. Non-default field/value pairs are listed as follows:
4543

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:

$$\begin{aligned}
 w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
 \end{aligned}$$

4572 Logically, this operation occurs in three steps:

4573 **Setup** The internal vectors and mask used in the computation are formed and their domains
4574 and dimensions are tested for compatibility.

4575 **Compute** The indicated computations are carried out.

4576 **Output** The result is written into the output vector, possibly under control of a mask.

4577 Up to three argument vectors are used in the `GrB_assign` operation:

- 4578 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4579 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4580 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4581 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4582 bility as follows:

- 4583 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
4584 must be from one of the pre-defined types of Table 3.2.
- 4585 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 4586 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
4587 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
4588 mulation operator.

4589 Two domains are compatible with each other if values from one domain can be cast to values in
4590 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4591 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4592 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4593 error listed above is returned.

4594 From the arguments, the internal vectors, mask and index array used in the computation are formed
4595 (\leftarrow denotes copy):

- 4596 1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 4597 2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 4598 (a) If `mask` = `GrB_NULL`, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 4599 (b) If `mask` \neq `GrB_NULL`,
 - 4600 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 4601 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 4602 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.

4603 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4604 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:

4605 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i$, $\forall i : 0 \leq i < \text{nindices}$.

4606 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i]$, $\forall i : 0 \leq i < \text{nindices}$.

4607 The internal vector and mask are checked for dimension compatibility. The following conditions
4608 must hold:

4609 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4610 2. $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$.

4611 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4612 match error listed above is returned.

4613 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4614 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4615 We are now ready to carry out the assign and any additional associated operations. We describe
4616 this in terms of two intermediate vectors:

- 4617 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.
- 4618 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4619 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4620 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4621 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{w}}$, computation
4622 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`
4623 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4624 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4625 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4626 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4627 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4628 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4629 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
4630 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4631 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4632 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4633 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4634 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4636 where the difference operator refers to set difference.

4637 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4638 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4639 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4640 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 4641 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 4642 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4643 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4644 \end{aligned}$$

4645 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4647 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
4648 using what is called a *standard vector mask and replace*. This is carried out under control of the
4649 mask which acts as a “write mask”.

4650 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
4651 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4652 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4653 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4654 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
4655 mask are unchanged:

$$4656 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4657 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4658 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4659 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
4660 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4661 sequence.

4662 4.3.7.2 assign: Standard matrix variant

4663 Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices.
4664 The dimensions of the input matrix are the same size as the row and column index arrays provided.

4665 C Syntax

```
4666      GrB_Info GrB_assign(GrB_Matrix      C,
4667                          const GrB_Matrix Mask,
4668                          const GrB_BinaryOp accum,
4669                          const GrB_Matrix A,
```

```

4670         const GrB_Index      *row_indices,
4671         GrB_Index             nrows,
4672         const GrB_Index      *col_indices,
4673         GrB_Index             ncols,
4674         const GrB_Descriptor desc);

```

4675 Parameters

4676 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
4677 that may be accumulated with the result of the assign operation. On output, the
4678 matrix holds the results of the operation.

4679 **Mask** (IN) An optional “write” mask that controls which results from this operation are
4680 stored into the output matrix **C**. The mask dimensions must match those of the
4681 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4682 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
4683 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4684 dimensions of **C**), **GrB_NULL** should be specified.

4685 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4686 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4687 specified.

4688 **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4689 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
4690 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
4691 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
4692 this array may be manipulated by the caller after this operation returns without
4693 affecting any deferred computations for this operation. If this array contains du-
4694 plicate values, it implies assignment of more than one value to the same location
4695 which leads to undefined results.

4696 **nrows** (IN) The number of values in the **row_indices** array. Must be equal to **nrows(A)**
4697 if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4698 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns
4699 of **C** that are assigned. If all columns of **C** are to be assigned in order from 0
4700 to **ncols** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4701 and return value, this array may be manipulated by the caller after this operation
4702 returns without affecting any deferred computations for this operation. If this
4703 array contains duplicate values, it implies assignment of more than one value to
4704 the same location which leads to undefined results.

4705 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **ncols(A)** if **A** is
4706 not transposed, or equal to **nrows(A)** if **A** is transposed.

4707
4708
4709

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:

4710

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.

4711 Return Values

4712
4713
4714
4715
4716

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.

4717

GrB_PANIC Unknown internal error.

4718
4719
4720
4721

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.

4722

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

4723
4724

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).

4725
4726
4727

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.

4728
4729

GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrow \neq nrow(A), or ncols \neq ncols(A).

4730
4731
4732
4733

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).

4734
4735

GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices is a NULL pointer, or both.

4736 Description

4737 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows
 4738 and columns in `C` in a specified order: $C(\text{row_indices}, \text{col_indices}) = A$; or, if an optional binary
 4739 accumulation operator (\odot) is provided, $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot$
 4740 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4741 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot A(i, j), \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4742 Logically, this operation occurs in three steps:

4743 Setup The internal matrices and mask used in the computation are formed and their domains
 4744 and dimensions are tested for compatibility.

4745 Compute The indicated computations are carried out.

4746 Output The result is written into the output matrix, possibly under control of a mask.

4747 Up to three argument matrices are used in the `GrB_assign` operation:

- 4748 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4749 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)
- 4750 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4751 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
 4752 ibility as follows:

- 4753 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 4754 must be from one of the pre-defined types of Table 3.2.
- 4755 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 4756 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4757 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4758 mulation operator.

4759 Two domains are compatible with each other if values from one domain can be cast to values in
 4760 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4761 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4762 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4763 error listed above is returned.

4764 From the arguments, the internal matrices, mask, and index arrays used in the computation are
 4765 formed (\leftarrow denotes copy):

- 4766 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4767 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
- 4768 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
4769 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 4770 (b) If `Mask \neq GrB_NULL`,
- 4771 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
4772 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
- 4773 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
4774 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
- 4775 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4776 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 4777 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4778 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
- 4779 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 4780 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4781 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
- 4782 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4783 The internal matrices and mask are checked for dimension compatibility. The following conditions
4784 must hold:

- 4785 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 4786 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 4787 3. $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$.
- 4788 4. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$.

4789 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4790 match error listed above is returned.

4791 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4792 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4793 We are now ready to carry out the assign and any additional associated operations. We describe
4794 this in terms of two intermediate vectors:

- 4795 • $\tilde{\mathbf{T}}$: The matrix holding the contents from $\tilde{\mathbf{A}}$ in their destination locations relative to $\tilde{\mathbf{C}}$.
- 4796 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4797 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4798 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \} \}.$$

4799 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
4800 $\tilde{\mathbf{J}}$ array is not in the range $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-
4801 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
4802 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
4803 this point forward in the sequence.

4804 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 4805 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$4806 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4807 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})) \} \}.$$

4808 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
4809 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
4810 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

4811 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4812 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4813 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4814 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \\ 4815$$

4816 where the difference operator refers to set difference.

- 4817 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4818 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}}) \} \}.$$

4819 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4820 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4821 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4822 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4823 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4824 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4825$$

4826 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4827 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
4828 using what is called a *standard matrix mask and replace*. This is carried out under control of the
4829 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

4863 bool or any of the predefined “built-in” types in Table 3.2. If the default mask
 4864 is desired (i.e., a mask that is all true with the dimensions of a column of C),
 4865 GrB_NULL should be specified.

4866 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 4867 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 4868 specified.

4869 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column
 4870 of C.

4871 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
 4872 the specified column of C that are to be assigned. If all elements of the column
 4873 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB_ALL should
 4874 be specified. Regardless of execution mode and return value, this array may be
 4875 manipulated by the caller after this operation returns without affecting any de-
 4876 ferred computations for this operation. If this array contains duplicate values, it
 4877 implies in assignment of more than one value to the same location which leads to
 4878 undefined results.

4879 **nrows** (IN) The number of values in **row_indices** array. Must be equal to **size(u)**.

4880 **col_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4881 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4882 should be specified. Non-default field/value pairs are listed as follows:

4883

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.

4884

4885 Return Values

4886 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 4887 blocking mode, this indicates that the compatibility tests on
 4888 dimensions and domains for the input arguments passed suc-
 4889 cessfully. Either way, output matrix C is ready to be used in the
 4890 next method of the sequence.

4891 **GrB_PANIC** Unknown internal error.

4923 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4924 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
4925 compatibility as follows:

4926 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4927 must be from one of the pre-defined types of Table 3.2.

4928 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}(\mathbf{u})$.

4929 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4930 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4931 mulation operator.

4932 Two domains are compatible with each other if values from one domain can be cast to values in
4933 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4934 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4935 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4936 error listed above is returned.

4937 The `col_index` parameter is checked for a valid value. The following condition must hold:

4938 1. $0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{C})$

4939 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
4940 is returned.

4941 From the arguments, the internal vectors, `mask`, and index array used in the computation are
4942 formed (\leftarrow denotes copy):

4943 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a column of \mathbf{C} as follows:

4944
$$\tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4945 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

4946 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$.

4947 (b) If `mask` \neq `GrB_NULL`,

4948 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

4949 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

4950 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

4951 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4952 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:

4953 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.

4954 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i]$, $\forall i : 0 \leq i < \text{nrows}$.

4955 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
4956 conditions must hold:

- 4957 1. $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4958 2. $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$.

4959 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4960 match error listed above is returned.

4961 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4962 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4963 We are now ready to carry out the assign and any additional associated operations. We describe
4964 this in terms of two intermediate vectors:

- 4965 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 4966 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4967 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4968 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4969 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
4970 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
4971 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4972 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4973 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4974 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4975 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4976 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4977 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
4978 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4979 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4980 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$4981 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4982 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4984 where the difference operator refers to set difference.

4985 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4986 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4987 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4988 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4989 \quad z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

4990

$$4991 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

4992

$$4993 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

4994 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4995 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
 4996 result matrix, $\mathbf{C}(:, \text{col_index})$. This is carried out under control of the mask which acts as a “write
 4997 mask”.

4998 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(:, \text{col_index})$ on input to this
 4999 operation are deleted and the new contents of the column is given by:

$$5000 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5001 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 5002 copied into the column of the final result matrix, $\mathbf{C}(:, \text{col_index})$, and elements of this column
 5003 that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} 5004 \quad \mathbf{L}(\mathbf{C}) &= \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \\ 5005 &\quad \{(i, \text{col_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ 5006 &\quad \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

5007 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 5008 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 5009 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may
 5010 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5011 4.3.7.4 assign: Row variant

5012 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the
 5013 output cannot be transposed, a different variant of `assign` is provided to assign to a column of a
 5014 matrix.

5015 C Syntax

```
5016         GrB_Info GrB_assign(GrB_Matrix      C,  
5017                             const GrB_Vector mask,  
5018                             const GrB_BinaryOp accum,  
5019                             const GrB_Vector u,  
5020                             GrB_Index      row_index,  
5021                             const GrB_Index *col_indices,  
5022                             GrB_Index      ncols,  
5023                             const GrB_Descriptor desc);
```

5024 Parameters

5025 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values
5026 that may be accumulated with the result of the assign operation. On output, this
5027 matrix holds the results of the operation.

5028 **mask** (IN) An optional “write” mask that controls which results from this operation are
5029 stored into the specified row of the output matrix **C**. The mask dimensions must
5030 match those of a single row of the matrix **C**. If the **GrB_STRUCTURE** descriptor
5031 is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or
5032 any of the predefined “built-in” types in Table 3.2. If the default mask is desired
5033 (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB_NULL** should
5034 be specified.

5035 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5036 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5037 specified.

5038 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of
5039 **C**.

5040 **row_index** (IN) The index of the row in **C** to assign. Must be in the range $[0, \mathbf{nrows}(\mathbf{C})]$.

5041 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
5042 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to
5043 be assigned in order from index 0 to $\mathbf{ncols} - 1$, then **GrB_ALL** should be specified.
5044 Regardless of execution mode and return value, this array may be manipulated by
5045 the caller after this operation returns without affecting any deferred computations
5046 for this operation. If this array contains duplicate values, it implies in assignment
5047 of more than one value to the same location which leads to undefined results.

5048 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **size(u)**.

5049 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5050 should be specified. Non-default field/value pairs are listed as follows:

5051

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:

5080 $C(\text{row_index}, :) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(\text{row_index}, :$
5081 $) = C(\text{row_index}, :) \odot u$. Taking order of `col_indices` into account it is more explicitly written as:

5082 $C(\text{row_index}, \text{col_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$
5083 $C(\text{row_index}, \text{col_indices}[j]) = C(\text{row_index}, \text{col_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$

5083 Logically, this operation occurs in three steps:

5084 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
5085 domains and dimensions are tested for compatibility.

5086 **Compute** The indicated computations are carried out.

5087 **Output** The result is written into the output matrix, possibly under control of a mask.

5088 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5089 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
5090 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
5091 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5092 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
5093 compatibility as follows:

- 5094 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5095 must be from one of the pre-defined types of Table 3.2.
5096 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(u)$.
5097 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5098 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
5099 mulation operator.

5100 Two domains are compatible with each other if values from one domain can be cast to values in
5101 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5102 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5103 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
5104 error listed above is returned.

5105 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5106 1. $0 \leq \text{row_index} < \mathbf{nrows}(C)$

5107 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
5108 is returned.

5109 From the arguments, the internal vectors, mask, and index array used in the computation are
5110 formed (\leftarrow denotes copy):

5111 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a row of \mathbf{C} as follows:

$$5112 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5113 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

5114 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$.

5115 (b) If `mask \neq GrB_NULL`,

5116 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

5117 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

5118 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

5119 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5120 4. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:

5121 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.

5122 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

5123 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
5124 conditions must hold:

5125 1. $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5126 2. $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$.

5127 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
5128 match error listed above is returned.

5129 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5130 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5131 We are now ready to carry out the assign and any additional associated operations. We describe
5132 this in terms of two intermediate vectors:

- 5133 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 5134 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5135 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$5136 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5137 At this point, if any value of $\tilde{\mathbf{J}}[j]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
5138 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
5139 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
5140 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

5141 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5142 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}}$ is defined as

$$5143 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5144 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
5145 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
5146 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5147 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5148 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$5149 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5150 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5152 where the difference operator refers to set difference.

5153 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5154 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5155 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5156 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5157 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5158 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5159 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5160 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5163 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
5164 result matrix, $\mathbf{C}(\text{row_index}, :)$. This is carried out under control of the mask which acts as a “write
5165 mask”.

5166 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in $\mathbf{C}(\text{row_index}, :)$ on input to this
5167 operation are deleted and the new contents of the column is given by:

$$5168 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5169 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
5170 copied into the column of the final result matrix, $\mathbf{C}(\text{row_index}, :)$, and elements of this column
5171 that fall outside the set indicated by the mask are unchanged:

$$5172 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup$$

$$5173 \quad \{(\text{row_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5174 \quad \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5175 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
5176 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
5177 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but may
5178 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5179 4.3.7.5 assign: Constant vector variant[Scott: NEW CONTENT]

5180 Assign the same value to a specified subset of vector elements. With the use of GrB_ALL, the entire
5181 destination vector can be filled with the constant.

5182 C Syntax

```
5183         GrB_Info GrB_assign(GrB_Vector          w,  
5184                             const GrB_Vector    mask,  
5185                             const GrB_BinaryOp    accum,  
5186                             <type>              val,  
5187                             const GrB_Index      *indices,  
5188                             GrB_Index            nindices,  
5189                             const GrB_Descriptor desc);
```

```
5190         GrB_Info GrB_assign(GrB_Vector          w,  
5191                             const GrB_Vector    mask,  
5192                             const GrB_BinaryOp    accum,  
5193                             const GrB_Scalar      s,  
5194                             const GrB_Index      *indices,  
5195                             GrB_Index            nindices,  
5196                             const GrB_Descriptor desc);
```

5197 Parameters

5198 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5199 that may be accumulated with the result of the assign operation. On output, this
5200 vector holds the results of the operation.

5201 **mask** (IN) An optional “write” mask that controls which results from this operation are
5202 stored into the output vector **w**. The mask dimensions must match those of the
5203 vector **w**. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5204 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
5205 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5206 dimensions of **w**), GrB_NULL should be specified.

5207 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5208 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5209 specified.

5210 **val** (IN) Scalar value to assign to (a subset of) **w**.

5211 **s** (IN) Scalar value to assign to (a subset of) **w**.

5212 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
5213 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0

5214 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode
5215 and return value, this array may be manipulated by the caller after this operation
5216 returns without affecting any deferred computations for this operation. In this
5217 variant, the specific order of the values in the array has no effect on the result.
5218 Unlike other variants, if there are duplicated values in this array the result is still
5219 defined.

5220 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If
5221 `nindices` is zero, the operation becomes a NO-OP.

5222 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
5223 should be specified. Non-default field/value pairs are listed as follows:

5224

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> .

5225

5226 Return Values

5227 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5228 blocking mode, this indicates that the compatibility tests on
5229 dimensions and domains for the input arguments passed suc-
5230 cessfully. Either way, output vector `w` is ready to be used in the
5231 next method of the sequence.

5232 **GrB_PANIC** Unknown internal error.

5233 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
5234 opaque GraphBLAS objects (input or output) is in an invalid
5235 state caused by a previous execution error. Call `GrB_error()` to
5236 access any error messages generated by the implementation.

5237 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5238 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized
5239 by a call to `new` (or `dup` for vector parameters).

5240 **GrB_INDEX_OUT_OF_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-
5241 blocking mode, this can be reported as an execution error.

5242 **GrB_DIMENSION_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less
5243 than `size(w)`.

5274 4. If **accum** is not **GrB_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the
 5275 method, must be compatible with **D_{in2}(accum)** of the accumulation operator.

5276 Two domains are compatible with each other if values from one domain can be cast to values in
 5277 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 5278 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5279 any compatibility rule above is violated, execution of **GrB_assign** ends and the domain mismatch
 5280 error listed above is returned.

5281 From the arguments, the internal vectors, mask and index array used in the computation are formed
 5282 (\leftarrow denotes copy):

- 5283 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5284 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument **mask** as follows:
 - 5285 (a) If **mask** = **GrB_NULL**, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 5286 (b) If **mask** \neq **GrB_NULL**,
 - 5287 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 5288 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$.
 - 5289 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5290 3. Scalar $\tilde{s} \leftarrow s$ (**GrB_Scalar** version only).
- 5291 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument **indices** as follows:
 - 5292 (a) If **indices** = **GrB_ALL**, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$.
 - 5293 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$.

5294 The internal vector and mask are checked for dimension compatibility. The following conditions
 5295 must hold:

- 5296 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5297 2. $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$.

5298 If any compatibility rule above is violated, execution of **GrB_assign** ends and the dimension mis-
 5299 match error listed above is returned.

5300 From this point forward, in **GrB_NONBLOCKING** mode, the method can optionally exit with
 5301 **GrB_SUCCESS** return code and defer any computation and/or execution error codes.

5302 We are now ready to carry out the assign and any additional associated operations. We describe
 5303 this in terms of two intermediate vectors:

- 5304 • $\tilde{\mathbf{t}}$: The vector holding the copies of the scalar, either **val** or \tilde{s} , in their destination locations
 5305 relative to $\tilde{\mathbf{w}}$.

5306 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5307 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows. If a non-opaque scalar \mathbf{val} is provided:

5308
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5309 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 1$):

5310
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5311 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 0$):

5312
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5313 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
 5314 is not in the range $[0, \mathbf{size}(\tilde{\mathbf{w}}))$, the execution of `GrB_assign` ends and the index out-of-bounds
 5315 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a
 5316 sequence-terminating `GrB_wait()` is called. Regardless, the result vector, \mathbf{w} , is invalid from this
 5317 point forward in the sequence.

5318 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5319 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

5320
$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5321 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 5322 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
 5323 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5324 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5325 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

5326
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5327
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5329 where the difference operator refers to set difference. We note that in this case of assigning
 5330 a constant, $\{\tilde{\mathbf{I}}[k], \forall k\}$ and $\mathbf{ind}(\tilde{\mathbf{t}})$ are identical.

5331 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

5332
$$\langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5333 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5334 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

5335
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

5336
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5337
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5340 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5341 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 5342 using what is called a *standard vector mask and replace*. This is carried out under control of the
 5343 mask which acts as a “write mask”.

- 5344 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 5345 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$5346 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5347 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 5348 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 5349 mask are unchanged:

$$5350 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5351 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 5352 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 5353 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 5354 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 5355 sequence.

5356 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5357 Assign the same value to a specified subset of matrix elements. With the use of GrB_ALL, the
 5358 entire destination matrix can be filled with the constant.

5359 C Syntax

```
5360      GrB_Info GrB_assign(GrB_Matrix      C,
5361                        const GrB_Matrix  Mask,
5362                        const GrB_BinaryOp accum,
5363                        <type>           val,
5364                        const GrB_Index   *row_indices,
5365                        GrB_Index         nrows,
5366                        const GrB_Index   *col_indices,
5367                        GrB_Index         ncols,
5368                        const GrB_Descriptor desc);
```

```
5369      GrB_Info GrB_assign(GrB_Matrix      C,
5370                        const GrB_Matrix  Mask,
5371                        const GrB_BinaryOp accum,
5372                        const GrB_Scalar   s,
5373                        const GrB_Index   *row_indices,
5374                        GrB_Index         nrows,
```

```

5375         const GrB_Index      *col_indices,
5376         GrB_Index            ncols,
5377         const GrB_Descriptor  desc);

```

5378 Parameters

5379 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5380 that may be accumulated with the result of the assign operation. On output, the
5381 matrix holds the results of the operation.

5382 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5383 stored into the output matrix **C**. The mask dimensions must match those of the
5384 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5385 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
5386 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5387 dimensions of **C**), **GrB_NULL** should be specified.

5388 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5389 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5390 specified.

5391 **val** (IN) Scalar value to assign to (a subset of) **C**.

5392 **s** (IN) Scalar value to assign to (a subset of) **C**.

5393 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
5394 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
5395 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
5396 this array may be manipulated by the caller after this operation returns without
5397 affecting any deferred computations for this operation. Unlike other variants, if
5398 there are duplicated values in this array the result is still defined.

5399 **nrows** (IN) The number of values in **row_indices** array. Must be in the range: [0, **nrows**(**C**)].
5400 If **nrows** is zero, the operation becomes a NO-OP.

5401 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**
5402 that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** – 1,
5403 then **GrB_ALL** should be specified. Regardless of execution mode and return value,
5404 this array may be manipulated by the caller after this operation returns without
5405 affecting any deferred computations for this operation. Unlike other variants, if
5406 there are duplicated values in this array the result is still defined.

5407 **ncols** (IN) The number of values in **col_indices** array. Must be in the range: [0, **ncols**(**C**)].
5408 If **ncols** is zero, the operation becomes a NO-OP.

5409 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5410 should be specified. Non-default field/value pairs are listed as follows:

5411

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`

5441 or $C(\text{row_indices}, \text{col_indices}) = s$ is performed. If an optional binary accumulation operator
 5442 (\odot) is provided, then either $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot \text{val}$ or
 5443 $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot s$ is performed. More explicitly, if a
 5444 non-opaque value val is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = \text{val}, \text{ or} \\ 5445 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5446 Correspondingly, if a `GrB_Scalar` s is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = s, \text{ or} \\ 5447 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5448 Logically, this operation occurs in three steps:

5449 Setup The internal vectors and mask used in the computation are formed and their domains
 5450 and dimensions are tested for compatibility.

5451 Compute The indicated computations are carried out.

5452 Output The result is written into the output matrix, possibly under control of a mask.

5453 Up to two argument matrices are used in the `GrB_assign` operation:

- 5454 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5455 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)

5456 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain
 5457 compatibility as follows:

- 5458 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 5459 must be from one of the pre-defined types of Table 3.2.
- 5460 2. $\mathbf{D}(C)$ must be compatible with either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5461 method.
- 5462 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5463 of the accumulation operator.
- 5464 4. If `accum` is not `GrB_NULL`, then either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5465 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.

5498 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5499 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5500 We are now ready to carry out the assign and any additional associated operations. We describe
 5501 this in terms of two intermediate matrices:

- 5502 • $\tilde{\mathbf{T}}$: The matrix holding the copies of the scalar, either `val` or \tilde{s} , in their destination locations
 5503 relative to $\tilde{\mathbf{C}}$.
- 5504 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5505 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows. If a non-opaque scalar `val` is provided:

$$\begin{aligned} 5506 \quad \tilde{\mathbf{T}} = & \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5507 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 1):

$$\begin{aligned} 5508 \quad \tilde{\mathbf{T}} = & \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5509 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 0):

$$5510 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5511 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
 5512 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
 5513 $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-bounds error listed above is
 5514 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating
 5515 `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from this point forward in the
 5516 sequence.

5517 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 5518 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$\begin{aligned} 5519 \quad \tilde{\mathbf{Z}} = & \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5520 \quad & \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle. \end{aligned}$$

5521 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 5522 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
 5523 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

5524 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 5525 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 5526 \quad Z_{ij} = & \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5527 \quad & \\ 5528 \quad & \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \end{aligned}$$

5529 where the difference operator refers to set difference. We note that, in this particular case of
 5530 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.

5531 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5532 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5533 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5534 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5535 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5536 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5537 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5540 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5541 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5542 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5543 mask which acts as a “write mask”.

5544 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5545 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5546 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5547 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
5548 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
5549 mask are unchanged:

$$5550 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5551 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5552 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5553 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
5554 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
5555 sequence.

5556 4.3.8 `apply`: Apply a function to the elements of an object

5557 Computes the transformation of the values of the elements of a vector or a matrix using a unary
5558 function, or a binary function where one argument is bound to a scalar.

5559 4.3.8.1 `apply`: Vector variant

5560 Computes the transformation of the values of the elements of a vector using a unary function.

5561 C Syntax

```

5562         GrB_Info GrB_apply(GrB_Vector          w,
5563                             const GrB_Vector    mask,
5564                             const GrB_BinaryOp   accum,
5565                             const GrB_UnaryOp    op,
5566                             const GrB_Vector    u,
5567                             const GrB_Descriptor desc);

```

5568 Parameters

5569 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5570 that may be accumulated with the result of the apply operation. On output, this
5571 vector holds the results of the operation.

5572 **mask** (IN) An optional “write” mask that controls which results from this operation are
5573 stored into the output vector **w**. The mask dimensions must match those of the
5574 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5575 of the mask vector must be of type **bool** or any of the predefined “built-in” types
5576 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5577 dimensions of **w**), **GrB_NULL** should be specified.

5578 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5579 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5580 specified.

5581 **op** (IN) A unary operator applied to each element of input vector **u**.

5582 **u** (IN) The GraphBLAS vector to which the unary function is applied.

5583 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5584 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 .

5587 Return Values

5588 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5589 blocking mode, this indicates that the compatibility tests on di-
5590 mensions and domains for the input arguments passed successfully.

5591 Either way, output vector w is ready to be used in the next method
 5592 of the sequence.

5593 **GrB_PANIC** Unknown internal error.

5594 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 5595 GraphBLAS objects (input or output) is in an invalid state caused
 5596 by a previous execution error. Call **GrB_error()** to access any error
 5597 messages generated by the implementation.

5598 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5599 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 5600 a call to **new** (or **dup** for vector parameters).

5601 **GrB_DIMENSION_MISMATCH** $mask$, w and/or u dimensions are incompatible.

5602 **GrB_DOMAIN_MISMATCH** The domains of the various vectors are incompatible with the corre-
 5603 sponding domains of the accumulation operator or unary function,
 5604 or the mask's domain is not compatible with **bool** (in the case where
 5605 $desc[GrB_MASK].GrB_STRUCTURE$ is not set).

5606 Description

5607 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
 5608 GraphBLAS vector: $w = f(u)$; or, if an optional binary accumulation operator (\odot) is provided,
 5609 $w = w \odot f(u)$.

5610 Logically, this operation occurs in three steps:

5611 **Setup** The internal vectors and mask used in the computation are formed and their domains
 5612 and dimensions are tested for compatibility.

5613 **Compute** The indicated computations are carried out.

5614 **Output** The result is written into the output vector, possibly under control of a mask.

5615 Up to three argument vectors are used in this **GrB_apply** operation:

- 5616 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5617 2. $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$ (optional)
- 5618 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5619 The argument vectors, unary operator and the accumulation operator (if provided) are tested for
 5620 domain compatibility as follows:

- 5621 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5622 must be from one of the pre-defined types of Table 3.2.
- 5623 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5624 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5625 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5626 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5627 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in}(\text{op})$.

5628 Two domains are compatible with each other if values from one domain can be cast to values in
5629 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5630 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5631 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
5632 error listed above is returned.

5633 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
5634 denotes copy):

- 5635 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5636 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 5637 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 5638 (b) If `mask` \neq `GrB_NULL`,
 - 5639 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 5640 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 5641 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5642 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5643 The internal vectors and masks are checked for dimension compatibility. The following conditions
5644 must hold:

- 5645 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5646 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

5647 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5648 error listed above is returned.

5649 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5650 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5651 We are now ready to carry out the apply and any additional associated operations. We describe
5652 this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$: The vector holding the result from applying the unary operator to the input vector $\tilde{\mathbf{u}}$.
- $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle,$$

where $f = \mathbf{f}(\text{op})$.

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

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

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

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

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

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

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

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

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

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

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

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

4.3.8.2 apply: Matrix variant

Computes the transformation of the values of the elements of a matrix using a unary function.

C Syntax

```
GrB_Info GrB_apply(GrB_Matrix      C,
                  const GrB_Matrix  Mask,
                  const GrB_BinaryOp accum,
                  const GrB_UnaryOp  op,
                  const GrB_Matrix  A,
                  const GrB_Descriptor desc);
```

Parameters

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

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

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

op (IN) A unary operator applied to each element of input matrix A.

A (IN) The GraphBLAS matrix to which the unary function is applied.

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

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

5713 Return Values

5714	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
5715		blocking mode, this indicates that the compatibility tests on
5716		dimensions and domains for the input arguments passed suc-
5717		cessfully. Either way, output matrix C is ready to be used in the
5718		next method of the sequence.
5719	GrB_PANIC	Unknown internal error.
5720	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
5721		opaque GraphBLAS objects (input or output) is in an invalid
5722		state caused by a previous execution error. Call GrB_error() to
5723		access any error messages generated by the implementation.
5724	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
5725	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
5726		by a call to new (or Matrix_dup for matrix parameters).
5727	GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, nrows \neq nrows (C), or
5728		ncols \neq ncols (C).
5729	GrB_DOMAIN_MISMATCH	The domains of the various matrices are incompatible with the
5730		corresponding domains of the accumulation operator or unary
5731		function, or the mask's domain is not compatible with bool (in
5732		the case where desc [GrB_MASK]. GrB_STRUCTURE is not set).

5733 Description

5734 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
 5735 GraphBLAS matrix: $C = f(A)$; or, if an optional binary accumulation operator (\odot) is provided,
 5736 $C = C \odot f(A)$.

5737 Logically, this operation occurs in three steps:

5738 **Setup** The internal matrices and mask used in the computation are formed and their domains
 5739 and dimensions are tested for compatibility.

5740 **Compute** The indicated computations are carried out.

5741 **Output** The result is written into the output matrix, possibly under control of a mask.

5742 Up to three argument matrices are used in the **GrB_apply** operation:

- 5743 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5744 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)

5745 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5746 The argument matrices, unary operator and the accumulation operator (if provided) are tested for
5747 domain compatibility as follows:

- 5748 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\text{Mask})$
5749 must be from one of the pre-defined types of Table 3.2.
- 5750 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5751 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5752 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5753 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5754 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in}(\text{op})$ of the unary operator.

5755 Two domains are compatible with each other if values from one domain can be cast to values in
5756 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5757 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5758 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
5759 error listed above is returned.

5760 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
5761 are formed (\leftarrow denotes copy):

- 5762 1. Matrix $\tilde{C} \leftarrow C$.
- 5763 2. Two-dimensional mask, \tilde{M} , is computed from argument **Mask** as follows:
 - 5764 (a) If **Mask** = **GrB_NULL**, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
5765 $j < \mathbf{ncols}(C)\} \rangle$.
 - 5766 (b) If **Mask** \neq **GrB_NULL**,
 - 5767 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
5768 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - 5769 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
5770 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - 5771 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
- 5772 3. Matrix $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$.

5773 The internal matrices and mask are checked for dimension compatibility. The following conditions
5774 must hold:

- 5775 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.
- 5776 2. $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$.
- 5777 3. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$.

5778 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

5779 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5780 error listed above is returned.

5781 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5782 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5783 We are now ready to carry out the apply and any additional associated operations. We describe
5784 this in terms of two intermediate matrices:

- 5785 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the unary operator to the input matrix $\tilde{\mathbf{A}}$.
- 5786 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5787 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$5788 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5789 where $f = \mathbf{f}(\mathbf{op})$.

5790 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 5791 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 5792 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5793 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5794 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5795 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 5796 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 5797 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5798 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5799 \quad & \\ 5800 \end{aligned}$$

5801 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5802 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5803 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5804 mask which acts as a “write mask”.

- 5805 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5806 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5807 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 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,
```



```

5844         const GrB_BinaryOp      accum,
5845         const GrB_BinaryOp      op,
5846         const GrB_Vector        u,
5847         <type>                  val,
5848         const GrB_Descriptor     desc);

5849 // bind-second + GraphBLAS scalar
5850 GrB_Info GrB_apply(GrB_Vector      w,
5851                   const GrB_Vector mask,
5852                   const GrB_BinaryOp accum,
5853                   const GrB_BinaryOp op,
5854                   const GrB_Vector u,
5855                   const GrB_Scalar s,
5856                   const GrB_Descriptor desc);

```

5857 Parameters

5858 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5859 that may be accumulated with the result of the apply operation. On output, this
5860 vector holds the results of the operation.

5861 **mask** (IN) An optional “write” mask that controls which results from this operation are
5862 stored into the output vector **w**. The mask dimensions must match those of the
5863 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5864 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
5865 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5866 dimensions of **w**), **GrB_NULL** should be specified.

5867 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5868 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5869 specified.

5870 **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar
5871 value, **val**.

5872 **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as
5873 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
5874 argument in the *bind-second* variant.

5875 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
5876 argument in the *bind-first* variant, or the right-hand (second) argument in the
5877 *bind-second* variant.

5878 **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand
5879 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
5880 the *bind-second* variant. It must not be empty.

5881 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
5882 should be specified. Non-default field/value pairs are listed as follows:

5883

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 .

5884

5885 Return Values

5886 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
5887 blocking mode, this indicates that the compatibility tests on di-
5888 mensions and domains for the input arguments passed successfully.
5889 Either way, output vector **w** is ready to be used in the next method
5890 of the sequence.

5891 GrB_PANIC Unknown internal error.

5892 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
5893 GraphBLAS objects (input or output) is in an invalid state caused
5894 by a previous execution error. Call GrB_error() to access any error
5895 messages generated by the implementation.

5896 GrB_OUT_OF_MEMORY Not enough memory available for operation.

5897 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
5898 a call to new (or dup for vector parameters).

5899 GrB_DIMENSION_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

5900 GrB_DOMAIN_MISMATCH The domains of the various vectors and scalar are incompatible with
5901 the corresponding domains of the binary operator or accumulation
5902 operator, or the **mask**'s domain is not compatible with **bool** (in the
5903 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

5904 GrB_EMPTY_OBJECT The GrB_Scalar **s** used in the call is empty (**nvals(s) = 0**) and
5905 therefore a value cannot be passed to the binary operator.

5906 Description

5907 This variant of GrB_apply computes the result of applying a binary operator to the elements of a
5908 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5909 bind-first: $w = f(\text{val}, u)$ or $w = f(s, u)$

5910 bind-second: $w = f(u, \text{val})$ or $w = f(u, s)$,

5911 or if an optional binary accumulation operator (\odot) is provided:

5912 bind-first: $w = w \odot f(\text{val}, u)$ or $w = w \odot f(s, u)$

5913 bind-second: $w = w \odot f(u, \text{val})$ or $w = w \odot f(u, s)$.

5914 Logically, this operation occurs in three steps:

5915 **Setup** The internal vectors and mask used in the computation are formed and their domains
5916 and dimensions are tested for compatibility.

5917 **Compute** The indicated computations are carried out.

5918 **Output** The result is written into the output vector, possibly under control of a mask.

5919 Up to three argument vectors are used in this GrB_apply operation:

5920 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

5921 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

5922 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5923 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are
5924 tested for domain compatibility as follows:

5925 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
5926 must be from one of the pre-defined types of Table 3.2.

5927 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.

5928 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5929 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
5930 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5931 4. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

5932 5. If bind-first:

5933 (a) $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

5934 (b) If the non-opaque scalar **val** is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
5935 of the binary operator.

5936 (c) If the GrB_Scalar **s** is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
5937 binary operator.

- 5938 6. If bind-second:
- 5939 (a) $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the binary operator.
- 5940 (b) If the non-opaque scalar \mathbf{val} is provided, then $\mathbf{D}(\mathbf{val})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
- 5941 of the binary operator.
- 5942 (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
- 5943 binary operator.

5944 Two domains are compatible with each other if values from one domain can be cast to values in

5945 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5946 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5947 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5948 error listed above is returned.

5949 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow

5950 denotes copy):

- 5951 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5952 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 5953 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 5954 (b) If `mask \neq GrB_NULL`,
- 5955 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 5956 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 5957 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5958 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 5959 4. Scalar $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$ (GraphBLAS scalar case).

5960 The internal vectors and masks are checked for dimension compatibility. The following conditions

5961 must hold:

- 5962 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5963 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

5964 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5965 error listed above is returned.

5966 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5967 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5968 If an empty `GrB_Scalar` $\tilde{\mathbf{s}}$ is provided ($\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

5969 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

5970 `val` with the same domain as $\tilde{\mathbf{s}}$ and set `val = val($\tilde{\mathbf{s}}$)`.

5971 We are now ready to carry out the apply and any additional associated operations. We describe

5972 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.

6006 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

6007 Computes the transformation of the values of the stored elements of a matrix using a binary
6008 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the
6009 first argument to the binary operator and stored elements of the matrix are passed as the second
6010 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument
6011 and the specified scalar value is passed as the second argument. The scalar can be passed either as
6012 a non-opaque variable or as a GrB_Scalar object.

6013 C Syntax

```
6014 // bind-first + scalar value
6015 GrB_Info GrB_apply(GrB_Matrix      C,
6016                   const GrB_Matrix Mask,
6017                   const GrB_BinaryOp accum,
6018                   const GrB_BinaryOp op,
6019                   <type>           val,
6020                   const GrB_Matrix A,
6021                   const GrB_Descriptor desc);
```

```
6022 // bind-first + GraphBLAS scalar
6023 GrB_Info GrB_apply(GrB_Matrix      C,
6024                   const GrB_Matrix Mask,
6025                   const GrB_BinaryOp accum,
6026                   const GrB_BinaryOp op,
6027                   const GrB_Scalar s,
6028                   const GrB_Matrix A,
6029                   const GrB_Descriptor desc);
```

```
6030 // bind-second + scalar value
6031 GrB_Info GrB_apply(GrB_Matrix      C,
6032                   const GrB_Matrix Mask,
6033                   const GrB_BinaryOp accum,
6034                   const GrB_BinaryOp op,
6035                   const GrB_Matrix A,
6036                   <type>           val,
6037                   const GrB_Descriptor desc);
```

```
6038 // bind-second + GraphBLAS scalar
6039 GrB_Info GrB_apply(GrB_Matrix      C,
6040                   const GrB_Matrix Mask,
6041                   const GrB_BinaryOp accum,
6042                   const GrB_BinaryOp op,
6043                   const GrB_Matrix A,
```

```

6044         const GrB_Scalar      s,
6045         const GrB_Descriptor desc);

```

6046 Parameters

6047 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6048 that may be accumulated with the result of the apply operation. On output, the
6049 matrix holds the results of the operation.

6050 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6051 stored into the output matrix C. The mask dimensions must match those of the
6052 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
6053 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
6054 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
6055 dimensions of C), `GrB_NULL` should be specified.

6056 **accum** (IN) An optional binary operator used for accumulating entries into existing C
6057 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
6058 specified.

6059 **op** (IN) A binary operator applied to each element of input matrix, A, with the element
6060 of the input matrix used as the left-hand argument, and the scalar value, `val`, used
6061 as the right-hand argument.

6062 **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as
6063 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
6064 argument in the *bind-second* variant.

6065 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
6066 argument in the *bind-first* variant, or the right-hand (second) argument in the
6067 *bind-second* variant.

6068 **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand
6069 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
6070 the *bind-second* variant. It must not be empty.

6071 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
6072 should be specified. Non-default field/value pairs are listed as follows:
6073

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation (<i>bind-second</i> variant only).
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation (<i>bind-first</i> variant only).

Return Values

GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call <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>Matrix_dup</code> for matrix parameters).
GrB_INDEX_OUT_OF_BOUNDS	A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</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> \neq <code>nrows(C)</code> , or <code>ncols</code> \neq <code>ncols(C)</code> .
GrB_DOMAIN_MISMATCH	The domains of the various matrices and scalar are incompatible with the corresponding domains of the binary operator or accumulation operator, or the mask's domain is not compatible with <code>bool</code> (in the case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
GrB_EMPTY_OBJECT	The <code>GrB_Scalar s</code> used in the call is empty (<code>nvals(s) = 0</code>) and therefore a value cannot be passed to the binary operator.

6101 Description

6102 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a
 6103 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6104 bind-first: $C = f(\text{val}, A)$ or $C = f(s, A)$

6105 bind-second: $C = f(A, \text{val})$ or $C = f(A, s)$,

6106 or if an optional binary accumulation operator (\odot) is provided:

6107 bind-first: $C = C \odot f(\text{val}, A)$ or $C = C \odot f(s, A)$

6108 bind-second: $C = C \odot f(A, \text{val})$ or $C = C \odot f(A, s)$.

6109 Logically, this operation occurs in three steps:

6110 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6111 and dimensions are tested for compatibility.

6112 **Compute** The indicated computations are carried out.

6113 **Output** The result is written into the output matrix, possibly under control of a mask.

6114 Up to three argument matrices are used in the `GrB_apply` operation:

- 6115 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6116 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)
- 6117 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6118 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are
 6119 tested for domain compatibility as follows:

- 6120 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 6121 must be from one of the pre-defined types of Table 3.2.
- 6122 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.
- 6123 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6124 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
 6125 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6126 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.
- 6127 5. If bind-first:
 6128 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

6129 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
 6130 of the binary operator.

6131 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
 6132 binary operator.

6133 6. If `bind-second`:

6134 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

6135 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$
 6136 of the binary operator.

6137 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the
 6138 binary operator.

6139 Two domains are compatible with each other if values from one domain can be cast to values in
 6140 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6141 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6142 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
 6143 error listed above is returned.

6144 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
 6145 are formed (\leftarrow denotes copy):

6146 1. Matrix $\tilde{C} \leftarrow C$.

6147 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:

6148 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
 6149 $j < \mathbf{ncols}(C)\} \rangle$.

6150 (b) If `Mask` \neq `GrB_NULL`,

6151 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
 6152 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,

6153 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
 6154 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.

6155 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.

6156 3. Matrix \tilde{A} is computed from argument `A` as follows:

6157 `bind-first`: $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? A^T : A$

6158 `bind-second`: $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$

6159 4. Scalar $\tilde{s} \leftarrow s$ (`GraphBLAS` scalar case).

6160 The internal matrices and mask are checked for dimension compatibility. The following conditions
 6161 must hold:

6162 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.

6163 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.

6164 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6165 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6166 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6167 error listed above is returned.

6168 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6169 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6170 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6171 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6172 \mathbf{val} with the same domain as \tilde{s} and set $\mathbf{val} = \mathbf{val}(\tilde{s})$.

6173 We are now ready to carry out the apply and any additional associated operations. We describe
6174 this in terms of two intermediate matrices:

- 6175 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the binary operator to the input matrix $\tilde{\mathbf{A}}$.
- 6176 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6177 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as one of the following:

6178 bind-first: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6179 bind-second: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6180 where $f = \mathbf{f}(\mathbf{op})$.

6181 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6182 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6183 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6184 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6185 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6186 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6187 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6188 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6189 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6190 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6193 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6194 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6195 mask which acts as a “write mask”.

- 6196 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
6197 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6198 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6199 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6200 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6201 mask are unchanged:

$$6202 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6203 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
6204 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
6205 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
6206 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6207 sequence.

6208 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6209 Computes the transformation of the values of the stored elements of a vector using an index unary
6210 operator that is a function of the stored value, its location indices, and an user provided scalar
6211 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6212 C Syntax

```
6213 GrB_Info GrB_apply(GrB_Vector      w,
6214                   const GrB_Vector  mask,
6215                   const GrB_BinaryOp accum,
6216                   const GrB_IndexUnaryOp op,
6217                   const GrB_Vector  u,
6218                   <type>            val,
6219                   const GrB_Descriptor desc);
```

```
6220 GrB_Info GrB_apply(GrB_Vector      w,
6221                   const GrB_Vector  mask,
6222                   const GrB_BinaryOp accum,
6223                   const GrB_IndexUnaryOp op,
6224                   const GrB_Vector  u,
6225                   const GrB_Scalar  s,
6226                   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.

6258 GrB_PANIC Unknown internal error.

6259 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6260 opaque GraphBLAS objects (input or output) is in an invalid
6261 state caused by a previous execution error. Call GrB_error() to
6262 access any error messages generated by the implementation.

6263 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6264 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6265 by a call to new (or another constructor).

6266 GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

6267 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
6268 responding domains of the accumulation operator or index unary
6269 operator, or the mask's domain is not compatible with bool (in
6270 the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

6271 GrB_EMPTY_OBJECT The GrB_Scalar s used in the call is empty ($\mathbf{nvals}(s) = 0$) and
6272 therefore a value cannot be passed to the index unary operator.

6273 Description

6274 This variant of GrB_apply computes the result of applying an index unary operator to the elements
6275 of a GraphBLAS vector each composed with the element's index and a scalar constant, val or s:

$$6276 \quad \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \text{ or } \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}),$$

6277 or if an optional binary accumulation operator (\odot) is provided:

$$6278 \quad \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \text{ or } \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}).$$

6279 Logically, this operation occurs in three steps:

6280 **Setup** The internal vectors and mask used in the computation are formed and their domains
6281 and dimensions are tested for compatibility.

6282 **Compute** The indicated computations are carried out.

6283 **Output** The result is written into the output vector, possibly under control of a mask.

6284 Up to three argument vectors are used in this GrB_apply operation:

- 6285 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6286 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)

6287 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6288 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6289 are tested for domain compatibility as follows:

- 6290 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
6291 must be from one of the pre-defined types of Table 3.2.
- 6292 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the index unary operator.
- 6293 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6294 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be compatible
6295 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6296 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6297 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
6298 the index unary operator.
- 6299 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
6300 unary operator.

6301 Two domains are compatible with each other if values from one domain can be cast to values in
6302 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6303 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6304 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
6305 error listed above is returned.

6306 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6307 denotes copy):

- 6308 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6309 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 6310 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 6311 (b) If `mask \neq GrB_NULL`,
 - 6312 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,
 - 6313 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 6314 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6315 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 6316 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6317 The internal vectors and masks are checked for dimension compatibility. The following conditions
6318 must hold:

6319 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6320 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

6321 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6322 error listed above is returned.

6323 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6324 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6325 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

6326 If a non-empty `GrB_Scalar`, \tilde{s} , is provided ($\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable `val`
6327 with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6328 We are now ready to carry out the apply and any additional associated operations. We describe
6329 this in terms of two intermediate vectors:

- 6330 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6331 $\tilde{\mathbf{u}}$.
- 6332 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6333 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6334 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6335 where $f_i = \mathbf{f}(\text{op})$.

6336 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6337 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 6338 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6339 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6340 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6341 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6342 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6343 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6344 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6345 \quad & \\ 6346 \end{aligned}$$

6347 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6348 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
6349 using what is called a *standard vector mask and replace*. This is carried out under control of the
6350 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.

6363 4.3.8.6 apply: Matrix index unary operator variant[Scott: NEW CONTENT]

6364 Computes the transformation of the values of the stored elements of a matrix using an index unary
6365 operator that is a function of the stored value, its location indices, and an user provided scalar
6366 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6367 C Syntax

```
6368     GrB_Info GrB_apply(GrB_Matrix      C,
6369                       const GrB_Matrix Mask,
6370                       const GrB_BinaryOp accum,
6371                       const GrB_IndexUnaryOp op,
6372                       const GrB_Matrix A,
6373                       <type>          val,
6374                       const GrB_Descriptor desc);
```

```
6375     GrB_Info GrB_apply(GrB_Matrix      C,
6376                       const GrB_Matrix Mask,
6377                       const GrB_BinaryOp accum,
6378                       const GrB_IndexUnaryOp op,
6379                       const GrB_Matrix A,
6380                       const GrB_Scalar s,
6381                       const GrB_Descriptor desc);
```

6382 Parameters

6383 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6384 that may be accumulated with the result of the apply operation. On output, the
6385 matrix holds the results of the operation.

6386 Mask (IN) An optional “write” mask that controls which results from this operation are
6387 stored into the output matrix C. The mask dimensions must match those of the
6388 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
6389 of the Mask matrix must be of type **bool** or any of the predefined “built-in” types
6390 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6391 dimensions of C), GrB_NULL should be specified.

6392 accum (IN) An optional binary operator used for accumulating entries into existing C
6393 entries. If assignment rather than accumulation is desired, GrB_NULL should be
6394 specified.

6395 op (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB_Index}), D_{in_2}, f_i \rangle$, applied
6396 to each element stored in the input matrix, A. It is a function of the stored element’s
6397 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6398 A (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6399 ator.

6400 val (IN) An additional scalar value that is passed to the index unary operator.

6401 s (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6402 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
6403 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.

6406 Return Values

6407 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
6408 blocking mode, this indicates that the compatibility tests on di-
6409 mensions and domains for the input arguments passed successfully.
6410 Either way, output matrix C is ready to be used in the next method
6411 of the sequence.

6412 GrB_PANIC Unknown internal error.

6413 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
6414 GraphBLAS objects (input or output) is in an invalid state caused

6415 by a previous execution error. Call `GrB_error()` to access any error
 6416 messages generated by the implementation.

6417 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

6418 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 6419 a call to `new` (or another constructor).

6420 **GrB_DIMENSION_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6421 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
 6422 corresponding domains of the accumulation operator or index unary
 6423 operator, or the mask's domain is not compatible with `bool` (in the
 6424 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6425 **GrB_EMPTY_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and
 6426 therefore a value cannot be passed to the index unary operator.

6427 Description

6428 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements
 6429 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar
 6430 constant, `val` or `s`:

$$6431 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6432 or if an optional binary accumulation operator (\odot) is provided:

$$6433 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6434 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
 6435 indices, respectively.

6436 Logically, this operation occurs in three steps:

6437 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6438 and dimensions are tested for compatibility.

6439 **Compute** The indicated computations are carried out.

6440 **Output** The result is written into the output matrix, possibly under control of a mask.

6441 Up to three argument matrices are used in the `GrB_apply` operation:

- 6442 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6443 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)

6444 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6445 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6446 are tested for domain compatibility as follows:

- 6447 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\mathbf{Mask})$
6448 must be from one of the pre-defined types of Table 3.2.
- 6449 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator.
- 6450 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6451 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be compatible
6452 with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accumulation operator.
- 6453 4. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
- 6454 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
6455 the index unary operator.
- 6456 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
6457 unary operator.

6458 Two domains are compatible with each other if values from one domain can be cast to values in
6459 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6460 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6461 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
6462 error listed above is returned.

6463 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation
6464 are formed (\leftarrow denotes copy):

- 6465 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 6466 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument **Mask** as follows:
 - 6467 (a) If **Mask** = **GrB_NULL**, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
6468 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 6469 (b) If **Mask** \neq **GrB_NULL**,
 - 6470 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
6471 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 6472 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
6473 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 6474 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 6475 3. Matrix $\tilde{\mathbf{A}}$ is computed from argument **A** as follows:

$$\tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$$
- 6477 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6478 The internal matrices and mask are checked for dimension compatibility. The following conditions
6479 must hold:

- 6480 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 6481 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 6482 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 6483 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6484 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6485 error listed above is returned.

6486 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6487 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6488 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6489 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6490 `val` with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6491 We are now ready to carry out the apply and any additional associated operations. We describe
6492 this in terms of two intermediate matrices:

- 6493 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6494 $\tilde{\mathbf{A}}$.
- 6495 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6496 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6497 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6498 where $f_i = \mathbf{f}(\mathbf{op})$.

6499 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6500 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6501 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6502 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6503 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6504 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 6505 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6506 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6507 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6508 \quad & \\ 6509 \end{aligned}$$

6510 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6511 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 6512 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 6513 mask which acts as a “write mask”.

- 6514 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 6515 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6516 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6517 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 6518 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 6519 mask are unchanged:

$$6520 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6521 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 6522 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 6523 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 6524 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 6525 sequence.

6526 4.3.9 select:

6527 Apply a select operator to the stored elements of an object to determine whether or not to keep
 6528 them.

6529 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6530 Apply a select operator (an index unary operator) to the elements of a vector.

6531 C Syntax

```
6532 // scalar value variant
6533 GrB_Info GrB_select(GrB_Vector          w,
6534                    const GrB_Vector     mask,
6535                    const GrB_BinaryOp   accum,
6536                    const GrB_IndexUnaryOp op,
6537                    const GrB_Vector     u,
6538                    <type>               val,
6539                    const GrB_Descriptor desc);
6540
6541 // GraphBLAS scalar variant
6542 GrB_Info GrB_select(GrB_Vector          w,
6543                    const GrB_Vector     mask,
```

```

6544         const GrB_BinaryOp      accum,
6545         const GrB_IndexUnaryOp  op,
6546         const GrB_Vector        u,
6547         const GrB_Scalar        s,
6548         const GrB_Descriptor    desc);
6549

```

6550 Parameters

6551 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6552 that may be accumulated with the result of the select operation. On output, this
6553 vector holds the results of the operation.

6554 **mask** (IN) An optional “write” mask that controls which results from this operation are
6555 stored into the output vector **w**. The mask dimensions must match those of the
6556 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6557 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6558 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6559 dimensions of **w**), **GrB_NULL** should be specified.

6560 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6561 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6562 specified.

6563 **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
6564 to each element stored in the input vector, **u**. It is a function of the stored element’s
6565 value, its location index, and a user supplied scalar value (either **s** or **val**).

6566 **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-
6567 ator.

6568 **val** (IN) An additional scalar value that is passed to the index unary operator.

6569 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6570 not be empty.

6571 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6572 should be specified. Non-default field/value pairs are listed as follows:

6573

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 .

6574

6575 Return Values

6576 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
6577 blocking mode, this indicates that the compatibility tests on di-
6578 mensions and domains for the input arguments passed success-
6579 fully. Either way, output vector **w** is ready to be used in the next
6580 method of the sequence.

6581 GrB_PANIC Unknown internal error.

6582 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6583 opaque GraphBLAS objects (input or output) is in an invalid
6584 state caused by a previous execution error. Call **GrB_error()** to
6585 access any error messages generated by the implementation.

6586 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6587 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6588 by a call to one of its constructors.

6589 GrB_DIMENSION_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

6590 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
6591 responding domains of the accumulation operator or index unary
6592 operator, or the **mask**'s domain is not compatible with **bool** (in
6593 the case where **desc[GrB_MASK].GrB_STRUCTURE** is not set).

6594 GrB_EMPTY_OBJECT The **GrB_Scalar s** used in the call is empty (**nvals(s) = 0**) and
6595 therefore a value cannot be passed to the index unary operator.

6596 Description

6597 This variant of **GrB_select** computes the result of applying a index unary operator to select the
6598 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored
6599 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding
6600 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.
6601 This acts like a functional mask on the input vector as follows:

$$6602 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6603 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6604 Correspondingly, if a **GrB_Scalar s**, is provided:

$$6605 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6606 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6607 Logically, this operation occurs in three steps:

6608 **Setup** The internal vectors and mask used in the computation are formed and their domains
6609 and dimensions are tested for compatibility.

6610 **Compute** The indicated computations are carried out.

6611 **Output** The result is written into the output vector, possibly under control of a mask.

6612 Up to three argument vectors are used in this `GrB_select` operation:

- 6613 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
6614 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
6615 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6616 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6617 are tested for domain compatibility as follows:

- 6618 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
6619 must be from one of the pre-defined types of Table 3.2.
6620 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
6621 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6622 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
6623 mulation operator.
6624 4. $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
6625 i.e., castable to `bool`.
6626 5. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
6627 6. $\mathbf{D}(\mathbf{val})$ or $\mathbf{D}(\mathbf{s})$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
6628 of the index unary operator.

6629 Two domains are compatible with each other if values from one domain can be cast to values in
6630 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6631 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6632 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch
6633 error listed above is returned.

6634 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6635 denotes copy):

- 6636 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
6637 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

6638 (a) If $\text{mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
6639 (b) If $\text{mask} \neq \text{GrB_NULL}$,
6640 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
6641 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
6642 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
6643 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
6644 4. Scalar $\widetilde{s} \leftarrow s$ (GrB_Scalar version only).

6645 The internal vectors and masks are checked for dimension compatibility. The following conditions
6646 must hold:

- 6647 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6648 2. $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$.

6649 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch
6650 error listed above is returned.

6651 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6652 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6653 If an empty `GrB_Scalar` \widetilde{s} is provided (i.e., $\text{nvals}(\widetilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6654 If a non-empty `GrB_Scalar`, \widetilde{s} , is provided (i.e., $\text{nvals}(\widetilde{s}) = 1$), we then create an internal variable
6655 `val` with the same domain as \widetilde{s} and set $\text{val} = \text{val}(\widetilde{s})$.

6656 We are now ready to carry out the `select` and any additional associated operations. We describe
6657 this in terms of two intermediate vectors:

- 6658 • $\widetilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6659 $\widetilde{\mathbf{u}}$.
- 6660 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6661 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$6662 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6663 where $f_i = \mathbf{f}(\text{op})$.

6664 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6665 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 6666 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$6667 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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);
```

```

6703 // GraphBLAS scalar variant
6704 GrB_Info GrB_select(GrB_Matrix          C,
6705                    const GrB_Matrix     Mask,
6706                    const GrB_BinaryOp   accum,
6707                    const GrB_IndexUnaryOp op,
6708                    const GrB_Matrix     A,
6709                    const GrB_Scalar     s,
6710                    const GrB_Descriptor  desc);

```

6711 Parameters

6712 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6713 that may be accumulated with the result of the select operation. On output, the
6714 matrix holds the results of the operation.

6715 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6716 stored into the output matrix **C**. The mask dimensions must match those of the
6717 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6718 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
6719 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6720 dimensions of **C**), **GrB_NULL** should be specified.

6721 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
6722 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6723 specified.

6724 **op** (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB_Index}), D_{in_2}, f_i \rangle$, applied
6725 to each element stored in the input matrix, **A**. It is a function of the stored element’s
6726 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6727 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6728 ator.

6729 **val** (IN) An additional scalar value that is passed to the index unary operator.

6730 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6731 not be empty.

6732 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6733 should be specified. Non-default field/value pairs are listed as follows:

6734

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:

6764 $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$, or
 6765 $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$.

6766 Correspondingly, if a GrB_Scalar, s , is provided:

6767 $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$, or
 6768 $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$.

6769 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
 6770 indices, respectively.

6771 Logically, this operation occurs in three steps:

6772 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6773 and dimensions are tested for compatibility.

6774 **Compute** The indicated computations are carried out.

6775 **Output** The result is written into the output matrix, possibly under control of a mask.

6776 Up to three argument matrices are used in the GrB_select operation:

- 6777 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6778 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)
- 6779 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6780 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
 6781 are tested for domain compatibility as follows:

- 6782 1. If **Mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
 6783 must be from one of the pre-defined types of Table 3.2.
- 6784 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 6785 3. If **accum** is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6786 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 6787 mulation operator.
- 6788 4. $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
 6789 i.e., castable to **bool**.
- 6790 5. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6791 6. $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\text{op})$
 6792 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:

- 6827 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6828 $\tilde{\mathbf{A}}$.
- 6829 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6830 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6831 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6832 where $f_i = \mathbf{f}(\text{op})$.

6833 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6834 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6835 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6836 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

6837 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6838 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6839 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6840 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6841 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6842 \quad 6843$$

6844 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6845 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6846 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6847 mask which acts as a “write mask”.

- 6848 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
6849 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6850 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6851 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6852 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6853 mask are unchanged:

$$6854 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6855 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6856 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6857 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
6858 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6859 sequence.

6860 4.3.10 reduce: Perform a reduction across the elements of an object

6861 Computes the reduction of the values of the elements of a vector or matrix.

6862 4.3.10.1 reduce: Standard matrix to vector variant

6863 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns
6864 is desired, the input matrix should be transposed using the descriptor.

6865 C Syntax

```
6866         GrB_Info GrB_reduce(GrB_Vector          w,  
6867                             const GrB_Vector    mask,  
6868                             const GrB_BinaryOp   accum,  
6869                             const GrB_Monoid     op,  
6870                             const GrB_Matrix     A,  
6871                             const GrB_Descriptor desc);  
6872  
6873         GrB_Info GrB_reduce(GrB_Vector          w,  
6874                             const GrB_Vector    mask,  
6875                             const GrB_BinaryOp   accum,  
6876                             const GrB_BinaryOp   op,  
6877                             const GrB_Matrix     A,  
6878                             const GrB_Descriptor desc);
```

6879 Parameters

6880 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6881 that may be accumulated with the result of the reduction operation. On output,
6882 this vector holds the results of the operation.

6883 **mask** (IN) An optional “write” mask that controls which results from this operation are
6884 stored into the output vector **w**. The mask dimensions must match those of the
6885 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6886 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6887 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6888 dimensions of **w**), **GrB_NULL** should be specified.

6889 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6890 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6891 specified.

6892 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.
6893 Depending on which type is passed, the following defines the binary operator with
6894 one domain, $F_b = \langle D, D, D, \oplus \rangle$, that is used:

6895 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
6896 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
6897 monoid is ignored.

6898 If op is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in
6899 both cases $\odot(\text{op})$ must be commutative and associative. Otherwise, the outcome
6900 of the operation is undefined.

6901 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

6902 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
6903 should be specified. Non-default field/value pairs are listed as follows:
6904

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.

6906 Return Values

6907 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
6908 blocking mode, this indicates that the compatibility tests on di-
6909 mensions and domains for the input arguments passed successfully.
6910 Either way, output vector w is ready to be used in the next method
6911 of the sequence.

6912 **GrB_PANIC** Unknown internal error.

6913 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
6914 GraphBLAS objects (input or output) is in an invalid state caused
6915 by a previous execution error. Call `GrB_error()` to access any error
6916 messages generated by the implementation.

6917 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

6918 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
6919 a call to `new` (or `dup` for vector parameters).

6920 **GrB_DIMENSION_MISMATCH** mask, w and/or u dimensions are incompatible.

6921 **GrB_DOMAIN_MISMATCH** Either the domains of the various vectors and matrices are incom-
6922 patible with the corresponding domains of the accumulation oper-
6923 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).

6927 Description

6928 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows
 6929 of an input matrix: $w(i) = \bigoplus A(i, :) \forall i$; or, if an optional binary accumulation operator is provided,
 6930 $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$, where $\bigoplus = \odot(F_b)$ and $\odot = \odot(\text{accum})$.

6931 Logically, this operation occurs in three steps:

6932 **Setup** The internal vector, matrix and mask used in the computation are formed and their
 6933 domains and dimensions are tested for compatibility.

6934 **Compute** The indicated computations are carried out.

6935 **Output** The result is written into the output vector, possibly under control of a mask.

6936 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6937 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6938 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 6939 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6940 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested
 6941 for domain compatibility as follows:

- 6942 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 6943 must be from one of the pre-defined types of Table 3.2.
- 6944 2. $\mathbf{D}(w)$ must be compatible with the domain of the reduction binary operator, $\mathbf{D}(F_b)$.
- 6945 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6946 of the accumulation operator and $\mathbf{D}(F_b)$, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 6947 mulation operator.
- 6948 4. $\mathbf{D}(A)$ must be compatible with the domain of the binary reduction operator, $\mathbf{D}(F_b)$.

6949 Two domains are compatible with each other if values from one domain can be cast to values in
 6950 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6951 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6952 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 6953 error listed above is returned.

6954 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 6955 denotes copy):

- 6956 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6957 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 6958 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 6959 (b) If `mask \neq GrB_NULL`,
- 6960 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 6961 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 6962 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6963 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

6964 The internal vectors and masks are checked for dimension compatibility. The following conditions
 6965 must hold:

- 6966 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6967 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6968 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-
 6969 match error listed above is returned.

6970 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 6971 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6972 We carry out the reduce and any additional associated operations. We describe this in terms of
 6973 two intermediate vectors:

- 6974 • $\tilde{\mathbf{t}}$: The vector holding the result from reducing along the rows of input matrix $\tilde{\mathbf{A}}$.
- 6975 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6976 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6977 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6978 The value of each of its elements is computed by

$$6979 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6980 where $\bigoplus = \odot(F_b)$.

6981 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6982 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

6983 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6984 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6985 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 6986 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6987 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 6988 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6989 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

6992 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6993 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 6994 using what is called a *standard vector mask and replace*. This is carried out under control of the
 6995 mask which acts as a “write mask”.

6996 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 6997 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$6998 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6999 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 7000 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 7001 mask are unchanged:

$$7002 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7003 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 7004 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 7005 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
 7006 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 7007 sequence.

7008 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

7009 Reduce all stored values into a single scalar.

7010 C Syntax

```
7011 // scalar value + monoid (only)
7012 GrB_Info GrB_reduce(<type>          *val,
7013                      const GrB_BinaryOp accum,
7014                      const GrB_Monoid  op,
7015                      const GrB_Vector  u,
```

```

7016             const GrB_Descriptor desc);
7017
7018 // GraphBLAS Scalar + monoid
7019 GrB_Info GrB_reduce(GrB_Scalar      s,
7020                   const GrB_BinaryOp accum,
7021                   const GrB_Monoid  op,
7022                   const GrB_Vector  u,
7023                   const GrB_Descriptor desc);
7024
7025 // GraphBLAS Scalar + binary operator
7026 GrB_Info GrB_reduce(GrB_Scalar      s,
7027                   const GrB_BinaryOp accum,
7028                   const GrB_BinaryOp op,
7029                   const GrB_Vector  u,
7030                   const GrB_Descriptor desc);

```

7031 Parameters

7032 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
7033 a value that may be accumulated (optionally) with the result of the reduction
7034 operation. On output, this scalar holds the results of the operation.

7035 **accum** (IN) An optional binary operator used for accumulating entries into an exist-
7036 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,
7037 **GrB_NULL** should be specified.

7038 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
7039 the reduction operation. The \oplus operator must be commutative and associative;
7040 otherwise, the outcome of the operation is undefined.

7041 **u** (IN) The GraphBLAS vector on which reduction will be performed.

7042 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
7043 should be specified. Non-default field/value pairs are listed as follows:

7045 Param	Field	Value	Description
------------	-------	-------	-------------

7046 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
7047 tions. There are currently no non-default field/value pairs that can be set for this
7048 operation.

7049 Return Values

7050 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
7051 cessfully, and the output scalar (**s** or **val**) is ready to be used in the
7052 next method of the sequence.

7053 GrB_PANIC Unknown internal error.

7054 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
7055 GraphBLAS objects (input or output) is in an invalid state caused
7056 by a previous execution error. Call GrB_error() to access any error
7057 messages generated by the implementation.

7058 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7059 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
7060 a call to a respective constructor.

7061 GrB_NULL_POINTER val pointer is NULL.

7062 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
7063 the corresponding domains of the accumulation operator, or reduce
7064 operator.

7065 Description

7066 This variant of GrB_reduce computes the result of performing a reduction across all of the stored
7067 elements of an input vector storing the result into either s or val. This corresponds to (shown here
7068 for the scalar value case only):

$$7069 \quad \text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[\bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7070 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7071 Logically, this operation occurs in three steps:

7072 **Setup** The internal vector used in the computation is formed and its domain is tested for
7073 compatibility.

7074 **Compute** The indicated computations are carried out.

7075 **Output** The result is written into the output scalar.

7076 One vector argument is used in this GrB_reduce operation:

- 7077 1. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7078 The output scalar, argument vector, reduction operator and accumulation operator (if provided)
7079 are tested for domain compatibility as follows:

- 7080 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
7081 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

- 7082 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
7083 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
7084 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 7085 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).

7086 Two domains are compatible with each other if values from one domain can be cast to values in
7087 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
7088 compatible with each other. A domain from a user-defined type is only compatible with itself. If
7089 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
7090 error listed above is returned.

7091 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one
7092 of the following occurs depending on the output variant:

$$7093 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7094 or

$$7095 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7096 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7097 For all other cases, the internal vector and scalar used in the computation is formed (\leftarrow denotes
7098 copy):

- 7099 1. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 7100 2. Scalar $\tilde{s} \leftarrow \text{s}$ (GraphBLAS scalar case).

7101 We are now ready to carry out the reduction and any additional associated operations. An inter-
7102 mediate scalar result t is computed as follows:

$$7103 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

7104 where $\oplus = \odot(\text{op})$.

7105 The final reduction value is computed as follows:

$$7106 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7107 or

$$7108 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7109 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
 7110 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7111 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7112 Reduce all stored values into a single scalar.

7113 C Syntax

```

7114 // scalar value + monoid (only)
7115 GrB_Info GrB_reduce(<type>          *val,
7116                   const GrB_BinaryOp accum,
7117                   const GrB_Monoid   op,
7118                   const GrB_Matrix   A,
7119                   const GrB_Descriptor desc);
7120
7121 // GraphBLAS Scalar + monoid
7122 GrB_Info GrB_reduce(GrB_Scalar      s,
7123                   const GrB_BinaryOp accum,
7124                   const GrB_Monoid   op,
7125                   const GrB_Matrix   A,
7126                   const GrB_Descriptor desc);
7127
7128 // GraphBLAS Scalar + binary operator
7129 GrB_Info GrB_reduce(GrB_Scalar      s,
7130                   const GrB_BinaryOp accum,
7131                   const GrB_BinaryOp op,
7132                   const GrB_Matrix   A,
7133                   const GrB_Descriptor desc);

```

7134 Parameters

7135 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
 7136 a value that may be accumulated (optionally) with the result of the reduction
 7137 operation. On output, this scalar holds the results of the operation.

7138 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or
 7139 **val**) value. If assignment rather than accumulation is desired, GrB_NULL should
 7140 be specified.

7141 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
 7142 the reduction operation. The \oplus operator must be commutative and associative;
 7143 otherwise, the outcome of the operation is undefined.

7144 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

7145 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 7146 should be specified. Non-default field/value pairs are listed as follows:

7147

7148	Param	Field	Value	Description
------	-------	-------	-------	-------------

7149 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
 7150 tions. There are currently no non-default field/value pairs that can be set for this
 7151 operation.

7152 Return Values

7153 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
 7154 cessfully, and the output scalar (s or val) is ready to be used in the
 7155 next method of the sequence.

7156 GrB_PANIC Unknown internal error.

7157 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 7158 GraphBLAS objects (input or output) is in an invalid state caused
 7159 by a previous execution error. Call GrB_error() to access any error
 7160 messages generated by the implementation.

7161 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7162 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 7163 a call to a respective constructor.

7164 GrB_NULL_POINTER val pointer is NULL.

7165 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
 7166 the corresponding domains of the accumulation operator, or reduce
 7167 operator.

7168 Description

7169 This variant of GrB_reduce computes the result of performing a reduction across all of the stored
 7170 elements of an input matrix storing the result into either s or val. This corresponds to (shown here
 7171 for the scalar value case only):

$$7172 \quad \text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[\bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7173 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7174 Logically, this operation occurs in three steps:

7175 **Setup** The internal matrix used in the computation is formed and its domain is tested for
 7176 compatibility.

7177 **Compute** The indicated computations are carried out.

7178 **Output** The result is written into the output scalar.

7179 One matrix argument is used in this GrB_reduce operation:

7180 1. $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7181 The output scalar, argument matrix, reduction operator and accumulation operator (if provided)
 7182 are tested for domain compatibility as follows:

7183 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
 7184 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

7185 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
 7186 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 7187 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

7188 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).

7189 Two domains are compatible with each other if values from one domain can be cast to values in
 7190 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7191 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7192 any compatibility rule above is violated, execution of GrB_reduce ends and the domain mismatch
 7193 error listed above is returned.

7194 The number of values stored in the input, A , is checked. If there are no stored values in A , then
 7195 one of the following occurs depending on the output variant:

$$7196 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7197 or

$$7198 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7199 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with GrB_SUCCESS.

7200 For all other cases, the internal matrix and scalar used in the computation is formed (\leftarrow denotes
 7201 copy):

7202 1. Matrix $\tilde{A} \leftarrow A$.

7203 2. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

7204 We are now ready to carry out the reduce and any additional associated operations. An intermediate
 7205 scalar result t is computed as follows:

$$7206 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7207 where $\oplus = \odot(\text{op})$.

7208 The final reduction value is computed as follows:

$$7209 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7210 or

$$7211 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7212 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
 7213 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7214 4.3.11 transpose: Transpose rows and columns of a matrix

7215 This version computes a new matrix that is the transpose of the source matrix.

7216 C Syntax

```
7217      GrB_Info GrB_transpose(GrB_Matrix      C,
7218                           const GrB_Matrix Mask,
7219                           const GrB_BinaryOp accum,
7220                           const GrB_Matrix A,
7221                           const GrB_Descriptor desc);
```

7222 Parameters

7223 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 7224 that may be accumulated with the result of the transpose operation. On output,
 7225 the matrix holds the results of the operation.

7226 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 7227 stored into the output matrix C. The mask dimensions must match those of the
 7228 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 7229 of the Mask matrix must be of type bool or any of the predefined “built-in” types
 7230 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
 7231 dimensions of C), GrB_NULL should be specified.

7232 **accum** (IN) An optional binary operator used for accumulating entries into existing C
7233 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
7234 specified.

7235 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7236 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
7237 should be specified. Non-default field/value pairs are listed as follows:
7238

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.

7240 Return Values

7241 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
7242 blocking mode, this indicates that the compatibility tests on di-
7243 mensions and domains for the input arguments passed successfully.
7244 Either way, output matrix C is ready to be used in the next method
7245 of the sequence.

7246 **GrB_PANIC** Unknown internal error.

7247 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
7248 GraphBLAS objects (input or output) is in an invalid state caused
7249 by a previous execution error. Call **GrB_error()** to access any error
7250 messages generated by the implementation.

7251 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

7252 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
7253 a call to **new** (or **Matrix_dup** for matrix parameters).

7254 **GrB_DIMENSION_MISMATCH** mask, C and/or A dimensions are incompatible.

7255 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the cor-
7256 responding domains of the accumulation operator, or the mask's do-
7257 main is not compatible with **bool** (in the case where **desc[GrB_MASK].GrB_STRUCTURE**
7258 is not set).

7259 Description

7260 GrB_transpose computes the result of performing a transpose of the input matrix: $C = A^T$; or, if an
 7261 optional binary accumulation operator (\odot) is provided, $C = C \odot A^T$. We note that the input matrix
 7262 A can itself be optionally transposed before the operation, which would cause either an assignment
 7263 from A to C or an accumulation of A into C.

7264 Logically, this operation occurs in three steps:

7265 **Setup** The internal matrix and mask used in the computation are formed and their domains
 7266 and dimensions are tested for compatibility.

7267 **Compute** The indicated computations are carried out.

7268 **Output** The result is written into the output matrix, possibly under control of a mask.

7269 Up to three matrix arguments are used in this GrB_transpose operation:

- 7270 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7271 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)
- 7272 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7273 The argument matrices and accumulation operator (if provided) are tested for domain compatibility
 7274 as follows:

- 7275 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
 7276 must be from one of the pre-defined types of Table 3.2.
- 7277 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$ of the input matrix.
- 7278 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 7279 of the accumulation operator and $\mathbf{D}(A)$ of the input matrix must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 7280 of the accumulation operator.

7281 Two domains are compatible with each other if values from one domain can be cast to values in
 7282 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7283 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7284 any compatibility rule above is violated, execution of GrB_transpose ends and the domain mismatch
 7285 error listed above is returned.

7286 From the argument matrices, the internal matrices and mask used in the computation are formed
 7287 (\leftarrow denotes copy):

- 7288 1. Matrix $\tilde{C} \leftarrow C$.
- 7289 2. Two-dimensional mask, \tilde{M} , is computed from argument Mask as follows:

- 7290 (a) If $\text{Mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
7291 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 7292 (b) If $\text{Mask} \neq \text{GrB_NULL}$,
- 7293 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
7294 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
- 7295 ii. Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
7296 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
- 7297 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$.
- 7298 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

7299 The internal matrices and masks are checked for dimension compatibility. The following conditions
7300 must hold:

- 7301 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
- 7302 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
- 7303 3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.
- 7304 4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

7305 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension
7306 mismatch error listed above is returned.

7307 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
7308 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7309 We are now ready to carry out the matrix transposition and any additional associated operations.
7310 We describe this in terms of two intermediate matrices:

- 7311 • $\widetilde{\mathbf{T}}$: The matrix holding the transpose of $\widetilde{\mathbf{A}}$.
- 7312 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

7313 The intermediate matrix

$$7314 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7315 is created.

7316 The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7317 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- 7318 • If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$7319 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7320 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
7321 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned}
7322 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
7323 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
7324 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
7325 \quad & \\
7326 \quad &
\end{aligned}$$

7327 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7328 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
7329 using what is called a *standard matrix mask and replace*. This is carried out under control of the
7330 mask which acts as a “write mask”.

- 7331 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
7332 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7333 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7334 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
7335 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
7336 mask are unchanged:

$$7337 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7338 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
7339 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
7340 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
7341 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7342 sequence.

7343 4.3.12 kronecker: Kronecker product of two matrices

7344 Computes the Kronecker product of two matrices. The result is a matrix.

7345 C Syntax

```

7346      GrB_Info GrB_kronecker(GrB_Matrix      C,
7347                             const GrB_Matrix  Mask,
7348                             const GrB_BinaryOp accum,
7349                             const GrB_Semiring op,
7350                             const GrB_Matrix  A,
7351                             const GrB_Matrix  B,
7352                             const GrB_Descriptor desc);
7353
```



```

7354     GrB_Info GrB_kronecker(GrB_Matrix      C,
7355                             const GrB_Matrix Mask,
7356                             const GrB_BinaryOp accum,
7357                             const GrB_Monoid op,
7358                             const GrB_Matrix A,
7359                             const GrB_Matrix B,
7360                             const GrB_Descriptor desc);
7361
7362     GrB_Info GrB_kronecker(GrB_Matrix      C,
7363                             const GrB_Matrix Mask,
7364                             const GrB_BinaryOp accum,
7365                             const GrB_BinaryOp op,
7366                             const GrB_Matrix A,
7367                             const GrB_Matrix B,
7368                             const GrB_Descriptor desc);

```

7369 Parameters

7370 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
7371 that may be accumulated with the result of the Kronecker product. On output,
7372 the matrix holds the results of the operation.

7373 **Mask** (IN) An optional “write” mask that controls which results from this operation are
7374 stored into the output matrix C. The mask dimensions must match those of the
7375 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
7376 of the Mask matrix must be of type bool or any of the predefined “built-in” types
7377 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
7378 dimensions of C), GrB_NULL should be specified.

7379 **accum** (IN) An optional binary operator used for accumulating entries into existing C
7380 entries. If assignment rather than accumulation is desired, GrB_NULL should be
7381 specified.

7382 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
7383 operation. Depending on which type is passed, the following defines the binary
7384 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

7385 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

7386 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-
7387 nored.

7388 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
7389 is ignored.

7390 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
7391 product.

7392 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
7393 product.

7394 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
7395 should be specified. Non-default field/value pairs are listed as follows:
7396

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.

7398 Return Values

7399 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
7400 blocking mode, this indicates that the compatibility tests on di-
7401 mensions and domains for the input arguments passed successfully.
7402 Either way, output matrix C is ready to be used in the next method
7403 of the sequence.

7404 GrB_PANIC Unknown internal error.

7405 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
7406 GraphBLAS objects (input or output) is in an invalid state caused
7407 by a previous execution error. Call GrB_error() to access any error
7408 messages generated by the implementation.

7409 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7410 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
7411 a call to new (or Matrix_dup for matrix parameters).

7412 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

7413 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
7414 corresponding domains of the binary operator (op) or accumulation
7415 operator, or the mask's domain is not compatible with bool (in the
7416 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

7417 Description

7418 GrB_kronecker computes the Kronecker product $C = A \otimes B$ or, if an optional binary accumulation
7419 operator (\odot) is provided, $C = C \odot (A \otimes B)$ (where matrices A and B can be optionally transposed).

7420 The Kronecker product is defined as follows:

7421

$$7422 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

7423 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
7424 Kronecker product are defined as

$$7425 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7426 where \otimes is the multiplicative operator specified by the **op** parameter.

7427 Logically, this operation occurs in three steps:

7428 **Setup** The internal matrices and mask used in the computation are formed and their domains
7429 and dimensions are tested for compatibility.

7430 **Compute** The indicated computations are carried out.

7431 **Output** The result is written into the output matrix, possibly under control of a mask.

7432 Up to four argument matrices are used in the **GrB_kronecker** operation:

- 7433 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7434 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)
- 7435 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 7436 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

7437 The argument matrices, the "product" operator (**op**), and the accumulation operator (if provided)
7438 are tested for domain compatibility as follows:

- 7439 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\text{Mask})$
7440 must be from one of the pre-defined types of Table 3.2.
- 7441 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 7442 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 7443 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 7444 5. If **accum** is not **GrB_NULL**, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
7445 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of **op** must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
7446 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.

7480 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$
7481 $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$ is created. The value of
7482 each of its elements is computed by

$$7483 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7484 where \otimes is the multiplicative operator specified by the `op` parameter.

7485 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7486 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 7487 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$7488 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7489 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
7490 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 7491 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 7492 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 7493 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 7494 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 7495 \end{aligned}$$

7496 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7497 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
7498 using what is called a *standard matrix mask and replace*. This is carried out under control of the
7499 mask which acts as a “write mask”.

- 7500 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
7501 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7502 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7503 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
7504 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
7505 mask are unchanged:

$$7506 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7507 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
7508 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
7509 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
7510 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7511 sequence. s

Chapter 5

Nonpolymorphic interface[Scott: NEW CONTENT]

Each polymorphic GraphBLAS method (those with multiple parameter signatures under the same name) has a corresponding set of long-name forms that are specific to each parameter signature. That is show in Tables 5.1 through 5.11.

Table 5.1: Long-name, nonpolymorphic form of GraphBLAS methods.

Polymorphic signature	Nonpolymorphic signature
GrB_Monoid_new(GrB_Monoid*,...,bool)	GrB_Monoid_new_BOOL(GrB_Monoid*,GrB_BinaryOp,bool)
GrB_Monoid_new(GrB_Monoid*,...,int8_t)	GrB_Monoid_new_INT8(GrB_Monoid*,GrB_BinaryOp,int8_t)
GrB_Monoid_new(GrB_Monoid*,...,uint8_t)	GrB_Monoid_new_UINT8(GrB_Monoid*,GrB_BinaryOp,uint8_t)
GrB_Monoid_new(GrB_Monoid*,...,int16_t)	GrB_Monoid_new_INT16(GrB_Monoid*,GrB_BinaryOp,int16_t)
GrB_Monoid_new(GrB_Monoid*,...,uint16_t)	GrB_Monoid_new_UINT16(GrB_Monoid*,GrB_BinaryOp,uint16_t)
GrB_Monoid_new(GrB_Monoid*,...,int32_t)	GrB_Monoid_new_INT32(GrB_Monoid*,GrB_BinaryOp,int32_t)
GrB_Monoid_new(GrB_Monoid*,...,uint32_t)	GrB_Monoid_new_UINT32(GrB_Monoid*,GrB_BinaryOp,uint32_t)
GrB_Monoid_new(GrB_Monoid*,...,int64_t)	GrB_Monoid_new_INT64(GrB_Monoid*,GrB_BinaryOp,int64_t)
GrB_Monoid_new(GrB_Monoid*,...,uint64_t)	GrB_Monoid_new_UINT64(GrB_Monoid*,GrB_BinaryOp,uint64_t)
GrB_Monoid_new(GrB_Monoid*,...,float)	GrB_Monoid_new_FP32(GrB_Monoid*,GrB_BinaryOp,float)
GrB_Monoid_new(GrB_Monoid*,...,double)	GrB_Monoid_new_FP64(GrB_Monoid*,GrB_BinaryOp,double)
GrB_Monoid_new(GrB_Monoid*,...,other)	GrB_Monoid_new_UDT(GrB_Monoid*,GrB_BinaryOp,void*)

Table 5.2: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_setElement(..., bool,...)	GrB_Scalar_setElement_BOOL(..., bool,...)
GrB_Scalar_setElement(..., int8_t,...)	GrB_Scalar_setElement_INT8(..., int8_t,...)
GrB_Scalar_setElement(..., uint8_t,...)	GrB_Scalar_setElement_UINT8(..., uint8_t,...)
GrB_Scalar_setElement(..., int16_t,...)	GrB_Scalar_setElement_INT16(..., int16_t,...)
GrB_Scalar_setElement(..., uint16_t,...)	GrB_Scalar_setElement_UINT16(..., uint16_t,...)
GrB_Scalar_setElement(..., int32_t,...)	GrB_Scalar_setElement_INT32(..., int32_t,...)
GrB_Scalar_setElement(..., uint32_t,...)	GrB_Scalar_setElement_UINT32(..., uint32_t,...)
GrB_Scalar_setElement(..., int64_t,...)	GrB_Scalar_setElement_INT64(..., int64_t,...)
GrB_Scalar_setElement(..., uint64_t,...)	GrB_Scalar_setElement_UINT64(..., uint64_t,...)
GrB_Scalar_setElement(..., float,...)	GrB_Scalar_setElement_FP32(..., float,...)
GrB_Scalar_setElement(..., double,...)	GrB_Scalar_setElement_FP64(..., double,...)
GrB_Scalar_setElement(..., <i>other</i> ,...)	GrB_Scalar_setElement_UDT(..., const void*,...)
GrB_Scalar_extractElement(bool*,...)	GrB_Scalar_extractElement_BOOL(bool*,...)
GrB_Scalar_extractElement(int8_t*,...)	GrB_Scalar_extractElement_INT8(int8_t*,...)
GrB_Scalar_extractElement(uint8_t*,...)	GrB_Scalar_extractElement_UINT8(uint8_t*,...)
GrB_Scalar_extractElement(int16_t*,...)	GrB_Scalar_extractElement_INT16(int16_t*,...)
GrB_Scalar_extractElement(uint16_t*,...)	GrB_Scalar_extractElement_UINT16(uint16_t*,...)
GrB_Scalar_extractElement(int32_t*,...)	GrB_Scalar_extractElement_INT32(int32_t*,...)
GrB_Scalar_extractElement(uint32_t*,...)	GrB_Scalar_extractElement_UINT32(uint32_t*,...)
GrB_Scalar_extractElement(int64_t*,...)	GrB_Scalar_extractElement_INT64(int64_t*,...)
GrB_Scalar_extractElement(uint64_t*,...)	GrB_Scalar_extractElement_UINT64(uint64_t*,...)
GrB_Scalar_extractElement(float*,...)	GrB_Scalar_extractElement_FP32(float*,...)
GrB_Scalar_extractElement(double*,...)	GrB_Scalar_extractElement_FP64(double*,...)
GrB_Scalar_extractElement(<i>other</i> *,...)	GrB_Scalar_extractElement_UDT(void*,...)

Table 5.3: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Vector_build(...,const bool*,...)	GrB_Vector_build_BOOL(...,const bool*,...)
GrB_Vector_build(...,const int8_t*,...)	GrB_Vector_build_INT8(...,const int8_t*,...)
GrB_Vector_build(...,const uint8_t*,...)	GrB_Vector_build_UINT8(...,const uint8_t*,...)
GrB_Vector_build(...,const int16_t*,...)	GrB_Vector_build_INT16(...,const int16_t*,...)
GrB_Vector_build(...,const uint16_t*,...)	GrB_Vector_build_UINT16(...,const uint16_t*,...)
GrB_Vector_build(...,const int32_t*,...)	GrB_Vector_build_INT32(...,const int32_t*,...)
GrB_Vector_build(...,const uint32_t*,...)	GrB_Vector_build_UINT32(...,const uint32_t*,...)
GrB_Vector_build(...,const int64_t*,...)	GrB_Vector_build_INT64(...,const int64_t*,...)
GrB_Vector_build(...,const uint64_t*,...)	GrB_Vector_build_UINT64(...,const uint64_t*,...)
GrB_Vector_build(...,const float*,...)	GrB_Vector_build_FP32(...,const float*,...)
GrB_Vector_build(...,const double*,...)	GrB_Vector_build_FP64(...,const double*,...)
GrB_Vector_build(...,const <i>other</i> *,...)	GrB_Vector_build_UDT(...,const void*,...)
GrB_Vector_setElement(...,GrB_Scalar,...)	GrB_Vector_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Vector_setElement(...,bool,...)	GrB_Vector_setElement_BOOL(..., bool,...)
GrB_Vector_setElement(...,int8_t,...)	GrB_Vector_setElement_INT8(..., int8_t,...)
GrB_Vector_setElement(...,uint8_t,...)	GrB_Vector_setElement_UINT8(..., uint8_t,...)
GrB_Vector_setElement(...,int16_t,...)	GrB_Vector_setElement_INT16(..., int16_t,...)
GrB_Vector_setElement(...,uint16_t,...)	GrB_Vector_setElement_UINT16(..., uint16_t,...)
GrB_Vector_setElement(...,int32_t,...)	GrB_Vector_setElement_INT32(..., int32_t,...)
GrB_Vector_setElement(...,uint32_t,...)	GrB_Vector_setElement_UINT32(..., uint32_t,...)
GrB_Vector_setElement(...,int64_t,...)	GrB_Vector_setElement_INT64(..., int64_t,...)
GrB_Vector_setElement(...,uint64_t,...)	GrB_Vector_setElement_UINT64(..., uint64_t,...)
GrB_Vector_setElement(...,float,...)	GrB_Vector_setElement_FP32(..., float,...)
GrB_Vector_setElement(...,double,...)	GrB_Vector_setElement_FP64(..., double,...)
GrB_Vector_setElement(..., <i>other</i> ,...)	GrB_Vector_setElement_UDT(...,const void*,...)
GrB_Vector_extractElement(GrB_Scalar,...)	GrB_Vector_extractElement_Scalar(GrB_Scalar,...)
GrB_Vector_extractElement(bool*,...)	GrB_Vector_extractElement_BOOL(bool*,...)
GrB_Vector_extractElement(int8_t*,...)	GrB_Vector_extractElement_INT8(int8_t*,...)
GrB_Vector_extractElement(uint8_t*,...)	GrB_Vector_extractElement_UINT8(uint8_t*,...)
GrB_Vector_extractElement(int16_t*,...)	GrB_Vector_extractElement_INT16(int16_t*,...)
GrB_Vector_extractElement(uint16_t*,...)	GrB_Vector_extractElement_UINT16(uint16_t*,...)
GrB_Vector_extractElement(int32_t*,...)	GrB_Vector_extractElement_INT32(int32_t*,...)
GrB_Vector_extractElement(uint32_t*,...)	GrB_Vector_extractElement_UINT32(uint32_t*,...)
GrB_Vector_extractElement(int64_t*,...)	GrB_Vector_extractElement_INT64(int64_t*,...)
GrB_Vector_extractElement(uint64_t*,...)	GrB_Vector_extractElement_UINT64(uint64_t*,...)
GrB_Vector_extractElement(float*,...)	GrB_Vector_extractElement_FP32(float*,...)
GrB_Vector_extractElement(double*,...)	GrB_Vector_extractElement_FP64(double*,...)
GrB_Vector_extractElement(<i>other</i> *,...)	GrB_Vector_extractElement_UDT(void*,...)
GrB_Vector_extractTuples(...,bool*,...)	GrB_Vector_extractTuples_BOOL(..., bool*,...)
GrB_Vector_extractTuples(...,int8_t*,...)	GrB_Vector_extractTuples_INT8(..., int8_t*,...)
GrB_Vector_extractTuples(...,uint8_t*,...)	GrB_Vector_extractTuples_UINT8(..., uint8_t*,...)
GrB_Vector_extractTuples(...,int16_t*,...)	GrB_Vector_extractTuples_INT16(..., int16_t*,...)
GrB_Vector_extractTuples(...,uint16_t*,...)	GrB_Vector_extractTuples_UINT16(..., uint16_t*,...)
GrB_Vector_extractTuples(...,int32_t*,...)	GrB_Vector_extractTuples_INT32(..., int32_t*,...)
GrB_Vector_extractTuples(...,uint32_t*,...)	GrB_Vector_extractTuples_UINT32(..., uint32_t*,...)
GrB_Vector_extractTuples(...,int64_t*,...)	GrB_Vector_extractTuples_INT64(..., int64_t*,...)
GrB_Vector_extractTuples(...,uint64_t*,...)	GrB_Vector_extractTuples_UINT64(..., uint64_t*,...)
GrB_Vector_extractTuples(...,float*,...)	GrB_Vector_extractTuples_FP32(..., float*,...)
GrB_Vector_extractTuples(...,double*,...)	GrB_Vector_extractTuples_FP64(..., double*,...)
GrB_Vector_extractTuples(..., <i>other</i> *,...)	GrB_Vector_extractTuples_UDT(..., void*,...)

Table 5.4: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_build(...,const bool*,...)	GrB_Matrix_build_BOOL(...,const bool*,...)
GrB_Matrix_build(...,const int8_t*,...)	GrB_Matrix_build_INT8(...,const int8_t*,...)
GrB_Matrix_build(...,const uint8_t*,...)	GrB_Matrix_build_UINT8(...,const uint8_t*,...)
GrB_Matrix_build(...,const int16_t*,...)	GrB_Matrix_build_INT16(...,const int16_t*,...)
GrB_Matrix_build(...,const uint16_t*,...)	GrB_Matrix_build_UINT16(...,const uint16_t*,...)
GrB_Matrix_build(...,const int32_t*,...)	GrB_Matrix_build_INT32(...,const int32_t*,...)
GrB_Matrix_build(...,const uint32_t*,...)	GrB_Matrix_build_UINT32(...,const uint32_t*,...)
GrB_Matrix_build(...,const int64_t*,...)	GrB_Matrix_build_INT64(...,const int64_t*,...)
GrB_Matrix_build(...,const uint64_t*,...)	GrB_Matrix_build_UINT64(...,const uint64_t*,...)
GrB_Matrix_build(...,const float*,...)	GrB_Matrix_build_FP32(...,const float*,...)
GrB_Matrix_build(...,const double*,...)	GrB_Matrix_build_FP64(...,const double*,...)
GrB_Matrix_build(...,const <i>other</i> *,...)	GrB_Matrix_build_UDT(...,const void*,...)
GrB_Matrix_setElement(...,GrB_Scalar,...)	GrB_Matrix_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Matrix_setElement(...,bool,...)	GrB_Matrix_setElement_BOOL(..., bool,...)
GrB_Matrix_setElement(...,int8_t,...)	GrB_Matrix_setElement_INT8(..., int8_t,...)
GrB_Matrix_setElement(...,uint8_t,...)	GrB_Matrix_setElement_UINT8(..., uint8_t,...)
GrB_Matrix_setElement(...,int16_t,...)	GrB_Matrix_setElement_INT16(..., int16_t,...)
GrB_Matrix_setElement(...,uint16_t,...)	GrB_Matrix_setElement_UINT16(..., uint16_t,...)
GrB_Matrix_setElement(...,int32_t,...)	GrB_Matrix_setElement_INT32(..., int32_t,...)
GrB_Matrix_setElement(...,uint32_t,...)	GrB_Matrix_setElement_UINT32(..., uint32_t,...)
GrB_Matrix_setElement(...,int64_t,...)	GrB_Matrix_setElement_INT64(..., int64_t,...)
GrB_Matrix_setElement(...,uint64_t,...)	GrB_Matrix_setElement_UINT64(..., uint64_t,...)
GrB_Matrix_setElement(...,float,...)	GrB_Matrix_setElement_FP32(..., float,...)
GrB_Matrix_setElement(...,double,...)	GrB_Matrix_setElement_FP64(..., double,...)
GrB_Matrix_setElement(..., <i>other</i> ,...)	GrB_Matrix_setElement_UDT(...,const void*,...)
GrB_Matrix_extractElement(GrB_Scalar,...)	GrB_Matrix_extractElement_Scalar(GrB_Scalar,...)
GrB_Matrix_extractElement(bool*,...)	GrB_Matrix_extractElement_BOOL(bool*,...)
GrB_Matrix_extractElement(int8_t*,...)	GrB_Matrix_extractElement_INT8(int8_t*,...)
GrB_Matrix_extractElement(uint8_t*,...)	GrB_Matrix_extractElement_UINT8(uint8_t*,...)
GrB_Matrix_extractElement(int16_t*,...)	GrB_Matrix_extractElement_INT16(int16_t*,...)
GrB_Matrix_extractElement(uint16_t*,...)	GrB_Matrix_extractElement_UINT16(uint16_t*,...)
GrB_Matrix_extractElement(int32_t*,...)	GrB_Matrix_extractElement_INT32(int32_t*,...)
GrB_Matrix_extractElement(uint32_t*,...)	GrB_Matrix_extractElement_UINT32(uint32_t*,...)
GrB_Matrix_extractElement(int64_t*,...)	GrB_Matrix_extractElement_INT64(int64_t*,...)
GrB_Matrix_extractElement(uint64_t*,...)	GrB_Matrix_extractElement_UINT64(uint64_t*,...)
GrB_Matrix_extractElement(float*,...)	GrB_Matrix_extractElement_FP32(float*,...)
GrB_Matrix_extractElement(double*,...)	GrB_Matrix_extractElement_FP64(double*,...)
GrB_Matrix_extractElement(<i>other</i> ,...)	GrB_Matrix_extractElement_UDT(void*,...)
GrB_Matrix_extractTuples(..., bool*,...)	GrB_Matrix_extractTuples_BOOL(..., bool*,...)
GrB_Matrix_extractTuples(..., int8_t*,...)	GrB_Matrix_extractTuples_INT8(..., int8_t*,...)
GrB_Matrix_extractTuples(..., uint8_t*,...)	GrB_Matrix_extractTuples_UINT8(..., uint8_t*,...)
GrB_Matrix_extractTuples(..., int16_t*,...)	GrB_Matrix_extractTuples_INT16(..., int16_t*,...)
GrB_Matrix_extractTuples(..., uint16_t*,...)	GrB_Matrix_extractTuples_UINT16(..., uint16_t*,...)
GrB_Matrix_extractTuples(..., int32_t*,...)	GrB_Matrix_extractTuples_INT32(..., int32_t*,...)
GrB_Matrix_extractTuples(..., uint32_t*,...)	GrB_Matrix_extractTuples_UINT32(..., uint32_t*,...)
GrB_Matrix_extractTuples(..., int64_t*,...)	GrB_Matrix_extractTuples_INT64(..., int64_t*,...)
GrB_Matrix_extractTuples(..., uint64_t*,...)	GrB_Matrix_extractTuples_UINT64(..., uint64_t*,...)
GrB_Matrix_extractTuples(..., float*,...)	GrB_Matrix_extractTuples_FP32(..., float*,...)
GrB_Matrix_extractTuples(..., double*,...)	GrB_Matrix_extractTuples_FP64(..., double*,...)
GrB_Matrix_extractTuples(..., <i>other</i> *,...)	GrB_Matrix_extractTuples_UDT(..., void*,...)

Table 5.5: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_import(...,const bool*,...)	GrB_Matrix_import_BOOL(...,const bool*,...)
GrB_Matrix_import(...,const int8_t*,...)	GrB_Matrix_import_INT8(...,const int8_t*,...)
GrB_Matrix_import(...,const uint8_t*,...)	GrB_Matrix_import_UINT8(...,const uint8_t*,...)
GrB_Matrix_import(...,const int16_t*,...)	GrB_Matrix_import_INT16(...,const int16_t*,...)
GrB_Matrix_import(...,const uint16_t*,...)	GrB_Matrix_import_UINT16(...,const uint16_t*,...)
GrB_Matrix_import(...,const int32_t*,...)	GrB_Matrix_import_INT32(...,const int32_t*,...)
GrB_Matrix_import(...,const uint32_t*,...)	GrB_Matrix_import_UINT32(...,const uint32_t*,...)
GrB_Matrix_import(...,const int64_t*,...)	GrB_Matrix_import_INT64(...,const int64_t*,...)
GrB_Matrix_import(...,const uint64_t*,...)	GrB_Matrix_import_UINT64(...,const uint64_t*,...)
GrB_Matrix_import(...,const float*,...)	GrB_Matrix_import_FP32(...,const float*,...)
GrB_Matrix_import(...,const double*,...)	GrB_Matrix_import_FP64(...,const double*,...)
GrB_Matrix_import(...,const other,...)	GrB_Matrix_import_UDT(...,const void*,...)
GrB_Matrix_export(...,bool*,...)	GrB_Matrix_export_BOOL(...,bool*,...)
GrB_Matrix_export(...,int8_t*,...)	GrB_Matrix_export_INT8(...,int8_t*,...)
GrB_Matrix_export(...,uint8_t*,...)	GrB_Matrix_export_UINT8(...,uint8_t*,...)
GrB_Matrix_export(...,int16_t*,...)	GrB_Matrix_export_INT16(...,int16_t*,...)
GrB_Matrix_export(...,uint16_t*,...)	GrB_Matrix_export_UINT16(...,uint16_t*,...)
GrB_Matrix_export(...,int32_t*,...)	GrB_Matrix_export_INT32(...,int32_t*,...)
GrB_Matrix_export(...,uint32_t*,...)	GrB_Matrix_export_UINT32(...,uint32_t*,...)
GrB_Matrix_export(...,int64_t*,...)	GrB_Matrix_export_INT64(...,int64_t*,...)
GrB_Matrix_export(...,uint64_t*,...)	GrB_Matrix_export_UINT64(...,uint64_t*,...)
GrB_Matrix_export(...,float*,...)	GrB_Matrix_export_FP32(...,float*,...)
GrB_Matrix_export(...,double*,...)	GrB_Matrix_export_FP64(...,double*,...)
GrB_Matrix_export(...,other,...)	GrB_Matrix_export_UDT(...,void*,...)
GrB_free(GrB_Type*)	GrB_Type_free(GrB_Type*)
GrB_free(GrB_UnaryOp*)	GrB_UnaryOp_free(GrB_UnaryOp*)
GrB_free(GrB_IndexUnaryOp*)	GrB_IndexUnaryOp_free(GrB_IndexUnaryOp*)
GrB_free(GrB_BinaryOp*)	GrB_BinaryOp_free(GrB_BinaryOp*)
GrB_free(GrB_Monoid*)	GrB_Monoid_free(GrB_Monoid*)
GrB_free(GrB_Semiring*)	GrB_Semiring_free(GrB_Semiring*)
GrB_free(GrB_Scalar*)	GrB_Scalar_free(GrB_Scalar*)
GrB_free(GrB_Vector*)	GrB_Vector_free(GrB_Vector*)
GrB_free(GrB_Matrix*)	GrB_Matrix_free(GrB_Matrix*)
GrB_free(GrB_Descriptor*)	GrB_Descriptor_free(GrB_Descriptor*)
GrB_wait(GrB_Type, GrB_WaitMode)	GrB_Type_wait(GrB_Type, GrB_WaitMode)
GrB_wait(GrB_UnaryOp, GrB_WaitMode)	GrB_UnaryOp_wait(GrB_UnaryOp, GrB_WaitMode)
GrB_wait(GrB_IndexUnaryOp, GrB_WaitMode)	GrB_IndexUnaryOp_wait(GrB_IndexUnaryOp, GrB_WaitMode)
GrB_wait(GrB_BinaryOp, GrB_WaitMode)	GrB_BinaryOp_wait(GrB_BinaryOp, GrB_WaitMode)
GrB_wait(GrB_Monoid, GrB_WaitMode)	GrB_Monoid_wait(GrB_Monoid, GrB_WaitMode)
GrB_wait(GrB_Semiring, GrB_WaitMode)	GrB_Semiring_wait(GrB_Semiring, GrB_WaitMode)
GrB_wait(GrB_Scalar, GrB_WaitMode)	GrB_Scalar_wait(GrB_Scalar, GrB_WaitMode)
GrB_wait(GrB_Vector, GrB_WaitMode)	GrB_Vector_wait(GrB_Vector, GrB_WaitMode)
GrB_wait(GrB_Matrix, GrB_WaitMode)	GrB_Matrix_wait(GrB_Matrix, GrB_WaitMode)
GrB_wait(GrB_Descriptor, GrB_WaitMode)	GrB_Descriptor_wait(GrB_Descriptor, GrB_WaitMode)
GrB_error(const char**, const GrB_Type)	GrB_Type_error(const char**, const GrB_Type)
GrB_error(const char**, const GrB_UnaryOp)	GrB_UnaryOp_error(const char**, const GrB_UnaryOp)
GrB_error(const char**, const GrB_IndexUnaryOp)	GrB_IndexUnaryOp_error(const char**, const GrB_IndexUnaryOp)
GrB_error(const char**, const GrB_BinaryOp)	GrB_BinaryOp_error(const char**, const GrB_BinaryOp)
GrB_error(const char**, const GrB_Monoid)	GrB_Monoid_error(const char**, const GrB_Monoid)
GrB_error(const char**, const GrB_Semiring)	GrB_Semiring_error(const char**, const GrB_Semiring)
GrB_error(const char**, const GrB_Scalar)	GrB_Scalar_error(const char**, const GrB_Scalar)
GrB_error(const char**, const GrB_Vector)	GrB_Vector_error(const char**, const GrB_Vector)
GrB_error(const char**, const GrB_Matrix)	GrB_Matrix_error(const char**, const GrB_Matrix)
GrB_error(const char**, const GrB_Descriptor)	GrB_Descriptor_error(const char**, const GrB_Descriptor)

Table 5.6: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_eWiseMult(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseMult_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseMult_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseMult_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseMult_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseMult_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseMult_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseAdd_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseAdd_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseAdd_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseAdd_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseAdd_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseAdd_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_extract(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_extract(GrB_Vector,...,GrB_Vector,...)
GrB_extract(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_extract(GrB_Matrix,...,GrB_Matrix,...)
GrB_extract(GrB_Vector,...,GrB_Matrix,...)	GrB_Col_extract(GrB_Vector,...,GrB_Matrix,...)
GrB_assign(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_assign(GrB_Vector,...,GrB_Vector,...)
GrB_assign(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_assign(GrB_Matrix,...,GrB_Matrix,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)	GrB_Col_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)	GrB_Row_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)
GrB_assign(GrB_Vector,...,GrB_Scalar,...)	GrB_Vector_assign_Scalar(GrB_Vector,...,const GrB_Scalar,...)
GrB_assign(GrB_Vector,...,bool,...)	GrB_Vector_assign_BOOL(GrB_Vector,..., bool,...)
GrB_assign(GrB_Vector,...,int8_t,...)	GrB_Vector_assign_INT8(GrB_Vector,..., int8_t,...)
GrB_assign(GrB_Vector,...,uint8_t,...)	GrB_Vector_assign_UINT8(GrB_Vector,..., uint8_t,...)
GrB_assign(GrB_Vector,...,int16_t,...)	GrB_Vector_assign_INT16(GrB_Vector,..., int16_t,...)
GrB_assign(GrB_Vector,...,uint16_t,...)	GrB_Vector_assign_UINT16(GrB_Vector,..., uint16_t,...)
GrB_assign(GrB_Vector,...,int32_t,...)	GrB_Vector_assign_INT32(GrB_Vector,..., int32_t,...)
GrB_assign(GrB_Vector,...,uint32_t,...)	GrB_Vector_assign_UINT32(GrB_Vector,..., uint32_t,...)
GrB_assign(GrB_Vector,...,int64_t,...)	GrB_Vector_assign_INT64(GrB_Vector,..., int64_t,...)
GrB_assign(GrB_Vector,...,uint64_t,...)	GrB_Vector_assign_UINT64(GrB_Vector,..., uint64_t,...)
GrB_assign(GrB_Vector,...,float,...)	GrB_Vector_assign_FP32(GrB_Vector,..., float,...)
GrB_assign(GrB_Vector,...,double,...)	GrB_Vector_assign_FP64(GrB_Vector,..., double,...)
GrB_assign(GrB_Vector,...,other,...)	GrB_Vector_assign_UDT(GrB_Vector,...,const void*,...)
GrB_assign(GrB_Matrix,...,GrB_Scalar,...)	GrB_Matrix_assign_Scalar(GrB_Matrix,...,const GrB_Scalar,...)
GrB_assign(GrB_Matrix,...,bool,...)	GrB_Matrix_assign_BOOL(GrB_Matrix,..., bool,...)
GrB_assign(GrB_Matrix,...,int8_t,...)	GrB_Matrix_assign_INT8(GrB_Matrix,..., int8_t,...)
GrB_assign(GrB_Matrix,...,uint8_t,...)	GrB_Matrix_assign_UINT8(GrB_Matrix,..., uint8_t,...)
GrB_assign(GrB_Matrix,...,int16_t,...)	GrB_Matrix_assign_INT16(GrB_Matrix,..., int16_t,...)
GrB_assign(GrB_Matrix,...,uint16_t,...)	GrB_Matrix_assign_UINT16(GrB_Matrix,..., uint16_t,...)
GrB_assign(GrB_Matrix,...,int32_t,...)	GrB_Matrix_assign_INT32(GrB_Matrix,..., int32_t,...)
GrB_assign(GrB_Matrix,...,uint32_t,...)	GrB_Matrix_assign_UINT32(GrB_Matrix,..., uint32_t,...)
GrB_assign(GrB_Matrix,...,int64_t,...)	GrB_Matrix_assign_INT64(GrB_Matrix,..., int64_t,...)
GrB_assign(GrB_Matrix,...,uint64_t,...)	GrB_Matrix_assign_UINT64(GrB_Matrix,..., uint64_t,...)
GrB_assign(GrB_Matrix,...,float,...)	GrB_Matrix_assign_FP32(GrB_Matrix,..., float,...)
GrB_assign(GrB_Matrix,...,double,...)	GrB_Matrix_assign_FP64(GrB_Matrix,..., double,...)
GrB_assign(GrB_Matrix,...,other,...)	GrB_Matrix_assign_UDT(GrB_Matrix,...,const void*,...)

Table 5.7: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)	GrB_Vector_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)
GrB_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)	GrB_Matrix_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_BOOL(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT8(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT8(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT16(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT16(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT32(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT32(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT64(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT64(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP32(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP64(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp, <i>other</i> ,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UDT(GrB_Vector,...,GrB_BinaryOp,const void*,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_BinaryOp2nd_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_BinaryOp2nd_BOOL(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_BinaryOp2nd_INT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_BinaryOp2nd_INT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_BinaryOp2nd_INT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_BinaryOp2nd_INT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)	GrB_Vector_apply_BinaryOp2nd_FP32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)	GrB_Vector_apply_BinaryOp2nd_FP64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_BinaryOp2nd_UDT(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,const void*,...)

Table 5.8: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_BOOL(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT8(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT8(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT16(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT16(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT32(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT32(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT64(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT64(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP32(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP64(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp, <i>other</i> ,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UDT(GrB_Matrix,...,GrB_BinaryOp,const void*,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_BinaryOp2nd_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_BinaryOp2nd_BOOL(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_BinaryOp2nd_FP32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_BinaryOp2nd_FP64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_BinaryOp2nd_UDT(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,const void*,...)

Table 5.9: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_IndexOp_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_IndexOp_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_IndexOp_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_IndexOp_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_IndexOp_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_IndexOp_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_IndexOp_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_IndexOp_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_IndexOp_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_IndexOp_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)	GrB_Vector_apply_IndexOp_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)	GrB_Vector_apply_IndexOp_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_IndexOp_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_IndexOp_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_IndexOp_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_IndexOp_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_IndexOp_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_IndexOp_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_IndexOp_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_IndexOp_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_IndexOp_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_IndexOp_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_IndexOp_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_IndexOp_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_IndexOp_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_IndexOp_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)

Table 5.10: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>	<code>GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>	<code>GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>	<code>GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>	<code>GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>	<code>GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>	<code>GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>	<code>GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>	<code>GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>	<code>GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>	<code>GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>	<code>GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>	<code>GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)</code>	<code>GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>	<code>GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>	<code>GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>	<code>GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>	<code>GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>	<code>GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>	<code>GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>	<code>GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>	<code>GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>	<code>GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>	<code>GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>	<code>GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>	<code>GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)</code>	<code>GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)</code>

Table 5.11: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_reduce(GrB_Vector,...,GrB_Monoid,...)	GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Vector,...)	GrB_Vector_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Vector,...)	GrB_Vector_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(bool*,...,GrB_Vector,...)	GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)
GrB_reduce(int8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)
GrB_reduce(uint8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)
GrB_reduce(int16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)
GrB_reduce(uint16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)
GrB_reduce(int32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)
GrB_reduce(uint32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)
GrB_reduce(int64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)
GrB_reduce(uint64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)
GrB_reduce(float*,...,GrB_Vector,...)	GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)
GrB_reduce(double*,...,GrB_Vector,...)	GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)
GrB_reduce(<i>other</i> *,...,GrB_Vector,...)	GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)	GrB_Matrix_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)	GrB_Matrix_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)
GrB_reduce(bool*,...,GrB_Matrix,...)	GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)
GrB_reduce(int8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)
GrB_reduce(uint8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)
GrB_reduce(int16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)
GrB_reduce(uint16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)
GrB_reduce(int32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)
GrB_reduce(uint32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)
GrB_reduce(int64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)
GrB_reduce(uint64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)
GrB_reduce(float*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)
GrB_reduce(double*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)
GrB_reduce(<i>other</i> *,...,GrB_Matrix,...)	GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)
GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)

Table 5.12: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_get(GrB_Scalar,GrB_Scalar,GrB_Field)	GrB_Scalar_get_Scalar(GrB_Scalar,GrB_Scalar,GrB_Field)
GrB_get(GrB_Scalar,char*,GrB_Field)	GrB_Scalar_get_String(GrB_Scalar,char*,GrB_Field)
GrB_get(GrB_Scalar,int*,GrB_Field)	GrB_Scalar_get_ENUM(GrB_Scalar,int*,GrB_Field)
GrB_get(GrB_Scalar,size_t*,GrB_Field)	GrB_Scalar_get_SIZE(GrB_Scalar,size_t*,GrB_Field)
GrB_get(GrB_Scalar,void*,GrB_Field)	GrB_Scalar_get_VOID(GrB_Scalar,void*,GrB_Field)
GrB_get(GrB_Vector,GrB_Scalar,GrB_Field)	GrB_Vector_get_Scalar(GrB_Vector,GrB_Scalar,GrB_Field)
GrB_get(GrB_Vector,char*,GrB_Field)	GrB_Vector_get_String(GrB_Vector,char*,GrB_Field)
GrB_get(GrB_Vector,int*,GrB_Field)	GrB_Matrix_get_ENUM(GrB_Vector,int*,GrB_Field)
GrB_get(GrB_Vector,size_t*,GrB_Field)	GrB_Matrix_get_SIZE(GrB_Vector,size_t*,GrB_Field)
GrB_get(GrB_Vector,void*,GrB_Field)	GrB_Vector_get_VOID(GrB_Vector,void*,GrB_Field)
GrB_get(GrB_Matrix,GrB_Scalar,GrB_Field)	GrB_Matrix_get_Scalar(GrB_Matrix,GrB_Scalar,GrB_Field)
GrB_get(GrB_Matrix,char*,GrB_Field)	GrB_Matrix_get_String(GrB_Matrix,char*,GrB_Field)
GrB_get(GrB_Matrix,int*,GrB_Field)	GrB_Matrix_get_ENUM(GrB_Matrix,int*,GrB_Field)
GrB_get(GrB_Matrix,size_t*,GrB_Field)	GrB_Matrix_get_SIZE(GrB_Matrix,size_t*,GrB_Field)
GrB_get(GrB_Matrix,void*,GrB_Field)	GrB_Matrix_get_VOID(GrB_Matrix,void*,GrB_Field)
GrB_get(GrB_UnaryOp,GrB_Scalar,GrB_Field)	GrB_UnaryOp_get_Scalar(GrB_UnaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_UnaryOp,char*,GrB_Field)	GrB_UnaryOp_get_String(GrB_UnaryOp,char*,GrB_Field)
GrB_get(GrB_UnaryOp,int*,GrB_Field)	GrB_UnaryOp_get_ENUM(GrB_UnaryOp,int*,GrB_Field)
GrB_get(GrB_UnaryOp,size_t*,GrB_Field)	GrB_UnaryOp_get_SIZE(GrB_UnaryOp,size_t*,GrB_Field)
GrB_get(GrB_UnaryOp,void*,GrB_Field)	GrB_UnaryOp_get_VOID(GrB_UnaryOp,void*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)	GrB_IndexUnaryOp_get_Scalar(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_IndexUnaryOp,char*,GrB_Field)	GrB_IndexUnaryOp_get_String(GrB_IndexUnaryOp,char*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,int*,GrB_Field)	GrB_IndexUnaryOp_get_ENUM(GrB_IndexUnaryOp,int*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,size_t*,GrB_Field)	GrB_IndexUnaryOp_get_SIZE(GrB_IndexUnaryOp,size_t*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,void*,GrB_Field)	GrB_IndexUnaryOp_get_VOID(GrB_IndexUnaryOp,void*,GrB_Field)
GrB_get(GrB_BinaryOp,GrB_Scalar,GrB_Field)	GrB_BinaryOp_get_Scalar(GrB_BinaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_BinaryOp,char*,GrB_Field)	GrB_BinaryOp_get_String(GrB_BinaryOp,char*,GrB_Field)
GrB_get(GrB_BinaryOp,int*,GrB_Field)	GrB_BinaryOp_get_ENUM(GrB_BinaryOp,int*,GrB_Field)
GrB_get(GrB_BinaryOp,size_t*,GrB_Field)	GrB_BinaryOp_get_SIZE(GrB_BinaryOp,size_t*,GrB_Field)
GrB_get(GrB_BinaryOp,void*,GrB_Field)	GrB_BinaryOp_get_VOID(GrB_BinaryOp,void*,GrB_Field)
GrB_get(GrB_Monoid,GrB_Scalar,GrB_Field)	GrB_Monoid_get_Scalar(GrB_Monoid,GrB_Scalar,GrB_Field)
GrB_get(GrB_Monoid,char*,GrB_Field)	GrB_Monoid_get_String(GrB_Monoid,char*,GrB_Field)
GrB_get(GrB_Monoid,int*,GrB_Field)	GrB_Monoid_get_ENUM(GrB_Monoid,int*,GrB_Field)
GrB_get(GrB_Monoid,size_t*,GrB_Field)	GrB_Monoid_get_SIZE(GrB_Monoid,size_t*,GrB_Field)
GrB_get(GrB_Monoid,void*,GrB_Field)	GrB_Monoid_get_VOID(GrB_Monoid,void*,GrB_Field)
GrB_get(GrB_Semiring,GrB_Scalar,GrB_Field)	GrB_Semiring_get_Scalar(GrB_Semiring,GrB_Scalar,GrB_Field)
GrB_get(GrB_Semiring,char*,GrB_Field)	GrB_Semiring_get_String(GrB_Semiring,char*,GrB_Field)
GrB_get(GrB_Semiring,int*,GrB_Field)	GrB_Semiring_get_ENUM(GrB_Semiring,int*,GrB_Field)
GrB_get(GrB_Semiring,size_t*,GrB_Field)	GrB_Semiring_get_SIZE(GrB_Semiring,size_t*,GrB_Field)
GrB_get(GrB_Semiring,void*,GrB_Field)	GrB_Semiring_get_VOID(GrB_Semiring,void*,GrB_Field)
GrB_get(GrB_Descriptor,GrB_Scalar,GrB_Field)	GrB_Descriptor_get_Scalar(GrB_Descriptor,GrB_Scalar,GrB_Field)
GrB_get(GrB_Descriptor,char*,GrB_Field)	GrB_Descriptor_get_String(GrB_Descriptor,char*,GrB_Field)
GrB_get(GrB_Descriptor,int*,GrB_Field)	GrB_Descriptor_get_ENUM(GrB_Descriptor,int*,GrB_Field)
GrB_get(GrB_Descriptor,size_t*,GrB_Field)	GrB_Descriptor_get_SIZE(GrB_Descriptor,size_t*,GrB_Field)
GrB_get(GrB_Descriptor,void*,GrB_Field)	GrB_Descriptor_get_VOID(GrB_Descriptor,void*,GrB_Field)
GrB_get(GrB_Type,GrB_Scalar,GrB_Field)	GrB_Type_get_Scalar(GrB_Type,GrB_Scalar,GrB_Field)
GrB_get(GrB_Type,char*,GrB_Field)	GrB_Type_get_String(GrB_Type,char*,GrB_Field)
GrB_get(GrB_Type,int*,GrB_Field)	GrB_Type_get_ENUM(GrB_Type,int*,GrB_Field)
GrB_get(GrB_Type,size_t*,GrB_Field)	GrB_Type_get_SIZE(GrB_Type,size_t*,GrB_Field)
GrB_get(GrB_Type,void*,GrB_Field)	GrB_Type_get_VOID(GrB_Type,void*,GrB_Field)
GrB_get(GrB_Global,GrB_Scalar,GrB_Field)	GrB_Global_get_Scalar(GrB_Global,GrB_Scalar,GrB_Field)
GrB_get(GrB_Global,char*,GrB_Field)	GrB_Global_get_String(GrB_Global,char*,GrB_Field)
GrB_get(GrB_Global,int*,GrB_Field)	GrB_Global_get_ENUM(GrB_Global,int*,GrB_Field)
GrB_get(GrB_Global,size_t*,GrB_Field)	GrB_Global_get_SIZE(GrB_Global,size_t*,GrB_Field)
GrB_get(GrB_Global,void*,GrB_Field)	GrB_Global_get_VOID(GrB_Global,void*,GrB_Field)

Table 5.13: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_set(GrB_Scalar,GrB_Scalar,GrB_Field)	GrB_Scalar_set_Scalar(GrB_Scalar,GrB_Scalar,GrB_Field)
GrB_set(GrB_Scalar,char*,GrB_Field)	GrB_Scalar_set_String(GrB_Scalar,char*,GrB_Field)
GrB_set(GrB_Scalar,int,GrB_Field)	GrB_Scalar_set_ENUM(GrB_Scalar,int,GrB_Field)
GrB_set(GrB_Scalar,void*,GrB_Field,int)	GrB_Scalar_set_VOID(GrB_Scalar,void*,GrB_Field,int)
GrB_set(GrB_Vector,GrB_Scalar,GrB_Field)	GrB_Vector_set_Scalar(GrB_Vector,GrB_Scalar,GrB_Field)
GrB_set(GrB_Vector,char*,GrB_Field)	GrB_Vector_set_String(GrB_Vector,char*,GrB_Field)
GrB_set(GrB_Vector,int,GrB_Field)	GrB_Vector_set_ENUM(GrB_Vector,int,GrB_Field)
GrB_set(GrB_Vector,void*,GrB_Field,int)	GrB_Vector_set_VOID(GrB_Vector,void*,GrB_Field,int)
GrB_set(GrB_Matrix,GrB_Scalar,GrB_Field)	GrB_Matrix_set_Scalar(GrB_Matrix,GrB_Scalar,GrB_Field)
GrB_set(GrB_Matrix,char*,GrB_Field)	GrB_Matrix_set_String(GrB_Matrix,char*,GrB_Field)
GrB_set(GrB_Matrix,int,GrB_Field)	GrB_Matrix_set_ENUM(GrB_Matrix,int,GrB_Field)
GrB_set(GrB_Matrix,void*,GrB_Field,int)	GrB_Matrix_set_VOID(GrB_Matrix,void*,GrB_Field,int)
GrB_set(GrB_UnaryOp,GrB_Scalar,GrB_Field)	GrB_UnaryOp_set_Scalar(GrB_UnaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_UnaryOp,char*,GrB_Field)	GrB_UnaryOp_set_String(GrB_UnaryOp,char*,GrB_Field)
GrB_set(GrB_UnaryOp,int,GrB_Field)	GrB_UnaryOp_set_ENUM(GrB_UnaryOp,int,GrB_Field)
GrB_set(GrB_UnaryOp,void*,GrB_Field,int)	GrB_UnaryOp_set_VOID(GrB_UnaryOp,void*,GrB_Field,int)
GrB_set(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)	GrB_IndexUnaryOp_set_Scalar(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_IndexUnaryOp,char*,GrB_Field)	GrB_IndexUnaryOp_set_String(GrB_IndexUnaryOp,char*,GrB_Field)
GrB_set(GrB_IndexUnaryOp,int,GrB_Field)	GrB_IndexUnaryOp_set_ENUM(GrB_IndexUnaryOp,int,GrB_Field)
GrB_set(GrB_IndexUnaryOp,void*,GrB_Field,int)	GrB_IndexUnaryOp_set_VOID(GrB_IndexUnaryOp,void*,GrB_Field,int)
GrB_set(GrB_BinaryOp,GrB_Scalar,GrB_Field)	GrB_BinaryOp_set_Scalar(GrB_BinaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_BinaryOp,char*,GrB_Field)	GrB_BinaryOp_set_String(GrB_BinaryOp,char*,GrB_Field)
GrB_set(GrB_BinaryOp,int,GrB_Field)	GrB_BinaryOp_set_ENUM(GrB_BinaryOp,int,GrB_Field)
GrB_set(GrB_BinaryOp,void*,GrB_Field,int)	GrB_BinaryOp_set_VOID(GrB_BinaryOp,void*,GrB_Field,int)
GrB_set(GrB_Monoid,GrB_Scalar,GrB_Field)	GrB_Monoid_set_Scalar(GrB_Monoid,GrB_Scalar,GrB_Field)
GrB_set(GrB_Monoid,char*,GrB_Field)	GrB_Monoid_set_String(GrB_Monoid,char*,GrB_Field)
GrB_set(GrB_Monoid,int,GrB_Field)	GrB_Monoid_set_ENUM(GrB_Monoid,int,GrB_Field)
GrB_set(GrB_Monoid,void*,GrB_Field,int)	GrB_Monoid_set_VOID(GrB_Monoid,void*,GrB_Field,int)
GrB_set(GrB_Semiring,GrB_Scalar,GrB_Field)	GrB_Semiring_set_Scalar(GrB_Semiring,GrB_Scalar,GrB_Field)
GrB_set(GrB_Semiring,char*,GrB_Field)	GrB_Semiring_set_String(GrB_Semiring,char*,GrB_Field)
GrB_set(GrB_Semiring,int,GrB_Field)	GrB_Semiring_set_ENUM(GrB_Semiring,int,GrB_Field)
GrB_set(GrB_Semiring,void*,GrB_Field,int)	GrB_Semiring_set_VOID(GrB_Semiring,void*,GrB_Field,int)
GrB_set(GrB_Descriptor,GrB_Scalar,GrB_Field)	GrB_Descriptor_set_Scalar(GrB_Descriptor,GrB_Scalar,GrB_Field)
GrB_set(GrB_Descriptor,char*,GrB_Field)	GrB_Descriptor_set_String(GrB_Descriptor,char*,GrB_Field)
GrB_set(GrB_Descriptor,int,GrB_Field)	GrB_Descriptor_set_ENUM(GrB_Descriptor,int,GrB_Field)
GrB_set(GrB_Descriptor,void*,GrB_Field,int)	GrB_Descriptor_set_VOID(GrB_Descriptor,void*,GrB_Field,int)
GrB_set(GrB_Type,GrB_Scalar,GrB_Field)	GrB_Type_set_Scalar(GrB_Type,GrB_Scalar,GrB_Field)
GrB_set(GrB_Type,char*,GrB_Field)	GrB_Type_set_String(GrB_Type,char*,GrB_Field)
GrB_set(GrB_Type,int,GrB_Field)	GrB_Type_set_ENUM(GrB_Type,int,GrB_Field)
GrB_set(GrB_Type,void*,GrB_Field,int)	GrB_Type_set_VOID(GrB_Type,void*,GrB_Field,int)
GrB_set(GrB_Global,GrB_Scalar,GrB_Field)	GrB_Global_set_Scalar(GrB_Global,GrB_Scalar,GrB_Field)
GrB_set(GrB_Global,char*,GrB_Field)	GrB_Global_set_String(GrB_Global,char*,GrB_Field)
GrB_set(GrB_Global,int,GrB_Field)	GrB_Global_set_ENUM(GrB_Global,int,GrB_Field)
GrB_set(GrB_Global,void*,GrB_Field,int)	GrB_Global_set_VOID(GrB_Global,void*,GrB_Field,int)

Appendix A

Revision history

Changes in 2.0.1 (Released: ## Xxxxx 2022:

- (Issue GH-69) Fix error in description of contents of matrix constructed from `GrB_Matrix_diag`.

Changes in 2.0.0 (Released: 15 November 2021:

- Reorganized Chapters 2 and 3: Chapter 2 contains prose regarding the basic concepts captured in the API; Chapter 3 presents all of the enumerations, literals, data types, and predefined objects required by the API. Made short captions for the List of Tables.
- (Issue BB-49, BB-50) Updated and corrected language regarding multithreading and completion, and requirements regarding acquire-release memory orders. Methods that used to force complete no longer do.
- (Issue BB-74, BB-9) Assigned integer values to all return codes as well as all enumerations in the API to ensure run-time compatibility between libraries.
- (Issues BB-70, BB-67) Changed semantics and signature of `GrB_wait(obj, mode)`. Added wait modes for 'complete' or 'materialize' and removed `GrB_wait(void)`. **This breaks backward compatibility.**
- (Issue GH-51) Removed deprecated `GrB_SCMP` literal from descriptor values. **This breaks backward compatibility.**
- (Issues BB-8, BB-36) Added sparse `GrB_Scalar` object and its use in additional variants of `extract/setElement` methods, and `reduce`, `apply`, `assign` and `select` operations.
- (Issues BB-34, GH-33, GH-45) Added new `select` operation that uses an index unary operator. Added new variants of `apply` that take an index unary operator (matrix and vector variants).
- (Issues BB-68, BB-51) Added `serialize` and `deserialize` methods for matrices to/from implementation defined formats.

- 7542 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified
7543 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats
7544 have been deferred.
- 7545 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7546 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return
7547 error if duplicate locations encountered.
- 7548 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7549 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7550 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type `T` (not to
7551 be confused with the proposed unary operator).
- 7552 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added
7553 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to
7554 a method that is not supported by the implementation.
- 7555 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque
7556 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7557 • (Issue BB-45) Removed language about annihilators.
- 7558 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7559 • Updated a number algorithms in the appendix to use new operations and methods.
- 7560 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the
7561 specification.
- 7562 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and
7563 `GRB_SUBVERSION`.
- 7564 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7565 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection
7566 fixed to read `eWiseAdd`/set union.
- 7567 • Typographical error (`NEQ` \rightarrow `NE`) in Description of Table 3.8.

7568 Changes in 1.3.0 (Released: 25 September 2019):

- 7569 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque
7570 GraphBLAS object as an argument.
- 7571 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7572 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a
7573 scalar.

7574
7575

7576

7577
7578

7579
7580

7581

7582
7583

7584
7585

7586
7587

7588

7589
7590

7591

7592
7593

7594
7595

7596

7597
7598

7599

7600
7601

7602
7603

7604

7605

7606

- (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.

- 7607 • Fixed miscellaneous typographical errors (such as \otimes , \oplus).
- 7608 Changes in 1.2.0:
- 7609 • Removed "provisional" clause.
- 7610 Changes in 1.1.0:
- 7611 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
 - 7612 `assign` operations.
 - 7613 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
 - 7614 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
 - 7615 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
 - 7616 from the domains of the binary operator provided.
 - 7617 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
 - 7618 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
 - 7619 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
 - 7620 which is returned when the capacities of the output arrays are insufficient to hold all of the
 - 7621 tuples.
 - 7622 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
 - 7623 face.
 - 7624 • Added replace flag (`z`) notation to Table 4.1.
 - 7625 • Updated the “Mathematical Description” of the `assign` operation in Table 4.1.
 - 7626 • Added triangle counting example.
 - 7627 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
 - 7628 sections of GraphBLAS operations when the respective text was the “standard” text (i.e.,
 - 7629 identical in a majority of the operations).
 - 7630 • Fixed typographical errors.
- 7631 Changes in 1.0.2:
- 7632 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
 - 7633 matrices and avoid casting issues in certain implementations.
 - 7634 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
 - 7635 erased outside the assigned area.
 - 7636 • Changes non-polymorphic interface:
 - 7637 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7638 – Renamed GrB_Vector_reduce_BinaryOp to GrB_Matrix_reduce_BinaryOp.
- 7639 – Renamed GrB_Vector_reduce_Monoid to GrB_Matrix_reduce_Monoid.
- 7640 • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7641 • 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.

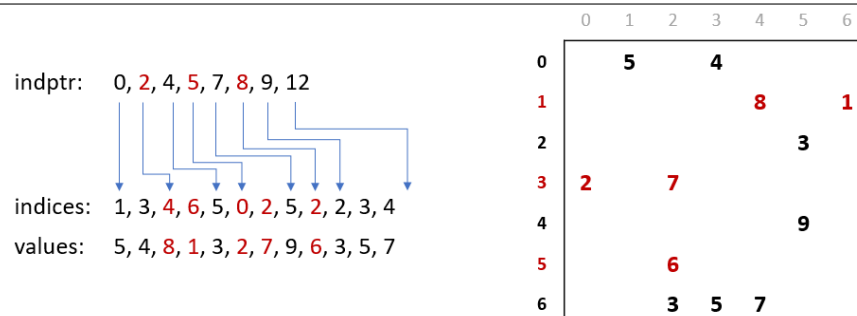


Figure B.1: Data layout for CSR format.

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.

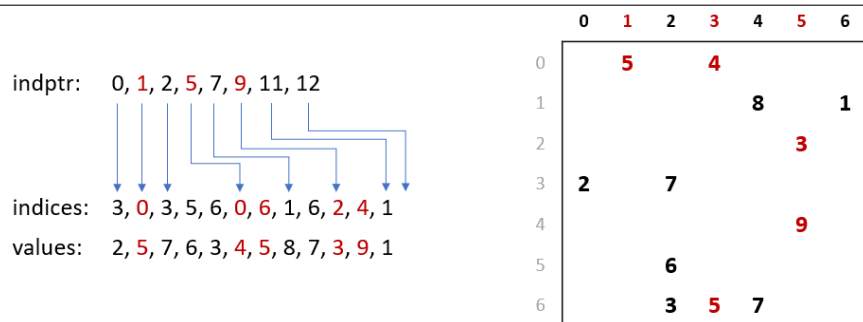


Figure B.2: Data layout for CSC format.

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.

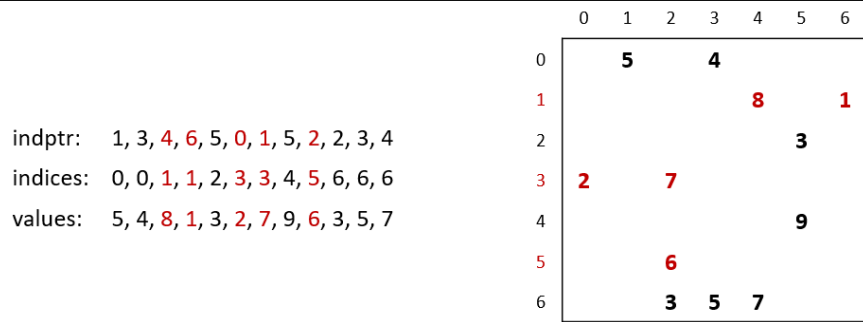


Figure B.3: Data layout for COO format.

7674 **Appendix C**

7675 **Examples**

C.1 Example: Level breadth-first search (BFS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i] = 0$ . (Vector  $v$  should be empty on input.)
11  */
12  GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
13  {
14      GrB_Index n;
15      GrB_Matrix_nrows(&n,A);                //  $n = \#$  of rows of  $A$ 
16
17      GrB_Vector_new(v,GrB_INT32,n);          // Vector<int32_t>  $v(n)$ 
18
19      GrB_Vector q;                          // vertices visited in each level
20      GrB_Vector_new(&q,GrB_BOOL,n);          // Vector<bool>  $q(n)$ 
21      GrB_Vector_setElement(q,(bool)true,s);  //  $q[s] = \text{true}$ , false everywhere else
22
23      /*
24       * BFS traversal and label the vertices.
25       */
26      int32_t d = 0;                          //  $d = \text{level in BFS traversal}$ 
27      bool succ = false;                      //  $\text{succ} == \text{true}$  when some successor found
28      do {
29          ++d;                                // next level (start with 1)
30          GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL); //  $v[q] = d$ 
31          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
32                q,A,GrB_DESC_RC);             //  $q[!v] = q \parallel A$ ; finds all the
33                                              // unvisited successors from current  $q$ 
34          GrB_reduce(&succ,GrB_NULL,GrB_LOR_MONOID_BOOL,
35                q,GrB_NULL);                  //  $\text{succ} = \parallel(q)$ 
36      } while (succ);                         // if there is no successor in  $q$ , we are done.
37
38      GrB_free(&q);                          //  $q$  vector no longer needed
39
40      return GrB_SUCCESS;
41  }

```

C.2 Example: Level BFS in GraphBLAS using apply

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i]$  does not have a stored element.
11  * Vector  $v$  should be uninitialized on input.
12  */
13  GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index n;
16      GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
17
18      GrB_Vector_new(v,GrB_INT32,n);     // Vector<int32_t>  $v(n) = 0$ 
19
20      GrB_Vector q;                     // vertices visited in each level
21      GrB_Vector_new(&q,GrB_BOOL,n);    // Vector<bool>  $q(n) = \text{false}$ 
22      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
23
24      /*
25       * BFS traversal and label the vertices.
26       */
27      int32_t level = 0;                // level = depth in BFS traversal
28      GrB_Index nvals;
29      do {
30          ++level;                      // next level (start with 1)
31          GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,
32                  GrB_SECOND_INT32,q,level,GrB_NULL); //  $v[q] = \text{level}$ 
33          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
34                  q,A,GrB_DESC_RC);    //  $q[!v] = q \ || \ \&\& \ A$ ; finds all the
35                                      // unvisited successors from current  $q$ 
36          GrB_Vector_nvals(&nvals, q);
37      } while (nvals);                 // if there is no successor in  $q$ , we are done.
38
39      GrB_free(&q);                    //  $q$  vector no longer needed
40
41      return GrB_SUCCESS;
42  }

```

C.3 Example: Parent BFS in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a binary  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS
9   * traversal of the graph and sets  $parents[i]$  to the index of vertex  $i$ 's parent.
10  * The parent of the root vertex,  $s$ , will be set to itself ( $parents[s] = s$ ). If
11  * vertex  $i$  is not reachable from  $s$ ,  $parents[i]$  will not contain a stored value.
12  */
13  GrB_Info BFS(GrB_Vector *parents, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index N;
16      GrB_Matrix_nrows(&N, A);                //  $N = \#$  vertices
17
18      GrB_Vector_new(parents, GrB_UINT64, N);
19      GrB_Vector_setElement(*parents, s, s);    //  $parents[s] = s$ 
20
21      GrB_Vector wavefront;
22      GrB_Vector_new(&wavefront, GrB_UINT64, N);
23      GrB_Vector_setElement(wavefront, 1UL, s); //  $wavefront[s] = 1$ 
24
25      /*
26       * BFS traversal and label the vertices.
27       */
28      GrB_Index nvals;
29      GrB_Vector_nvals(&nvals, wavefront);
30
31      while (nvals > 0)
32      {
33          // convert all stored values in wavefront to their 0-based index
34          GrB_apply(wavefront, GrB_NULL, GrB_NULL, GrB_ROWINDEX_INT64,
35                  wavefront, 0UL, GrB_NULL);
36
37          // "FIRST" because left-multiplying wavefront rows. Masking out the parent
38          // list ensures wavefront values do not overwrite parents already stored.
39          GrB_vxm(wavefront, *parents, GrB_NULL, GrB_MIN_FIRST_SEMIRING_UINT64,
40                  wavefront, A, GrB_DESC_RSC);
41
42          // Don't need to mask here since we did it in vxm. Merges new parents in
43          // current wavefront with existing parents:  $parents += wavefront$ 
44          GrB_apply(*parents, GrB_NULL, GrB_PLUS_UINT64,
45                  GrB_IDENTITY_UINT64, wavefront, GrB_NULL);
46
47          GrB_Vector_nvals(&nvals, wavefront);
48      }
49
50      GrB_free(&wavefront);
51
52      return GrB_SUCCESS;
53  }

```


C.4 Example: Betweenness centrality (BC) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ ,
9   * compute the BC-metric vector  $\delta$ , which should be empty on input.
10  */
11  GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
12  {
13      GrB_Index n;
14      GrB_Matrix_nrows(&n, A);                      //  $n = \#$  of vertices in graph
15
16      GrB_Vector_new(delta, GrB_FP32, n);            // Vector<float>  $\delta(n)$ 
17
18      GrB_Matrix sigma;
19      GrB_Matrix_new(&sigma, GrB_INT32, n, n);        // Matrix<int32_t>  $\sigma(n, n)$ 
20                                                         //  $\sigma[d, k] = \#$  shortest paths to node  $k$  at level  $d$ 
21
22      GrB_Vector q;
23      GrB_Vector_new(&q, GrB_INT32, n);              // Vector<int32_t>  $q(n)$  of path counts
24                                                         //  $q[s] = 1$ 
25
26      GrB_Vector p;
27      GrB_Vector_dup(&p, q);                          // Vector<int32_t>  $p(n)$  shortest path counts so far
28                                                         //  $p = q$ 
29
30      GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
31              q, A, GrB_DESC_RC);                    // get the first set of out neighbors
32
33      /*
34       * BFS phase
35       */
36      GrB_Index d = 0;                                // BFS level number
37      int32_t sum = 0;                                // sum == 0 when BFS phase is complete
38
39      do {
40          GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\sigma[d, :] = q$ 
41          GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
42          GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
43                  q, A, GrB_DESC_RC);                //  $q = \#$  paths to nodes reachable
44                                                         // from current level
45          GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
46          ++d;
47      } while (sum);
48
49      /*
50       * BC computation phase
51       * ( $t1, t2, t3, t4$ ) are temporary vectors
52       */
53      GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
54      GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
55      GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
56      GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
57
58      for (int i=d-1; i>0; i--)
59      {
60          GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t1 = 1 + \delta$ 
61          GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, t1, *delta, GrB_NULL);
62          GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t2 = \sigma[i, :]$ 
63          GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t2 = (1 + \delta) / \sigma[i, :]$ 
64          GrB_mvx(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
65                  // add contributions made by

```

```

63         A, t2, GrB_NULL);
64     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, GrB_DESC_T0); // t4 = sigma[i-1,:]
65     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, GrB_TIMES_FP32, t4, t3, GrB_NULL); // t4 = sigma[i-1,:]*t3
66     GrB_eWiseAdd(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, *delta, t4, GrB_NULL); // accumulate into delta
67 }
68
69 GrB_free(&sigma);
70 GrB_free(&q); GrB_free(&p);
71 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
72
73 return GrB_SUCCESS;
74 }

```

C.5 Example: Batched BC in GraphBLAS

```

1  #include <stdlib.h>
2  #include "GraphBLAS.h" // in addition to other required C headers
3
4  // Compute partial BC metric for a subset of source vertices, s, in graph A
5  GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
6  {
7      GrB_Index n;
8      GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
9      GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
10
11     // index and value arrays needed to build numsp
12     GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
13     int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
14     for(int i=0; i<nsver; ++i) {
15         i_nsver[i] = i;
16         ones[i] = 1;
17     }
18
19     // numsp: structure holds the number of shortest paths for each node and starting vertex
20     // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
21     GrB_Matrix numsp;
22     GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
23     GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
24     free(i_nsver); free(ones);
25
26     // frontier: Holds the current frontier where values are path counts.
27     // Initialized to out vertices of each source node in s.
28     GrB_Matrix frontier;
29     GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
30     GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, GrB_DESC_RCT0);
31
32     // sigma: stores frontier information for each level of BFS phase. The memory
33     // for an entry in sigmas is only allocated within the do-while loop if needed.
34     // n is an upper bound on diameter.
35     GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n);
36
37     int32_t d = 0; // BFS level number
38     GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
39
40     // ----- The BFS phase (forward sweep) -----
41     do {
42         // sigmas[d](:,s) = dth level frontier from source vertex s
43         GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
44
45         GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
46                 GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
47         GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL, GrB_PLUS_INT32,
48                 numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
49         GrB_mxm(frontier, numsp, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
50                 A, frontier, GrB_DESC_RCT0); // f<!numsp> = A' +.* f (update frontier)
51         GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level
52         d++;
53     } while (nvals);
54
55     // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
56     GrB_Matrix nspinv;
57     GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
58     GrB_apply(nspinv, GrB_NULL, GrB_NULL,
59             GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
60
61     // bcu: BC updates for each vertex for each starting vertex in s
62     GrB_Matrix bcu;

```

```

63 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
64 GrB_assign(bcu, GrB_NULL, GrB_NULL,
65           1.0f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
66
67 GrB_Matrix w; // temporary workspace matrix
68 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
69
70 // ----- Tally phase (backward sweep) -----
71 for (int i=d-1; i>0; i--) {
72     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
73                 GrB_TIMES_FP32, bcu, nspinv, GrB_DESC_R); // w<sigmas[i]>=(1 ./ nsp).*bcu
74
75     // add contributions by successors and mask with that BFS level's frontier
76     GrB_mxm(w, sigmas[i-1], GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
77            A, w, GrB_DESC_R); // w<sigmas[i-1]> = (A +.* w)
78     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32, GrB_TIMES_FP32,
79                 w, numsp, GrB_NULL); // bcu += w .* numsp
80 }
81
82 // row reduce bcu and subtract "nsver" from every entry to account
83 // for 1 extra value per bcu row element.
84 GrB_reduce(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, bcu, GrB_NULL);
85 GrB_apply(*delta, GrB_NULL, GrB_NULL, GrB_MINUS_FP32, *delta, (float)nsver, GrB_NULL);
86
87 // Release resources
88 for (int i=0; i<d; i++) {
89     GrB_free(&(sigmas[i]));
90 }
91 free(sigmas);
92
93 GrB_free(&frontier); GrB_free(&numsp);
94 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
95
96 return GrB_SUCCESS;
97 }

```

C.6 Example: Maximal independent set (MIS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  // Assign a random number to each element scaled by the inverse of the node's degree.
8  // This will increase the probability that low degree nodes are selected and larger
9  // sets are selected.
10 void setRandom(void *out, const void *in)
11 {
12     uint32_t degree = *(uint32_t*)in;
13     *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
14 }
15
16 /*
17  * A variant of Luby's randomized algorithm [Luby 1985].
18  *
19  * Given a numeric n x n adjacency matrix A of an unweighted and undirected graph (where
20  * the value true represents an edge), compute a maximal set of independent vertices and
21  * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
22  * of the set (the iset vector should be uninitialized on input.)
23  */
24 GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
25 {
26     GrB_Index n;
27     GrB_Matrix_nrows(&n,A); // n = # of rows of A
28
29     GrB_Vector prob; // holds random probabilities for each node
30     GrB_Vector neighbor_max; // holds value of max neighbor probability
31     GrB_Vector new_members; // holds set of new members to iset
32     GrB_Vector new_neighbors; // holds set of new neighbors to new iset mbrs.
33     GrB_Vector candidates; // candidate members to iset
34
35     GrB_Vector_new(&prob,GrB_FP32,n);
36     GrB_Vector_new(&neighbor_max,GrB_FP32,n);
37     GrB_Vector_new(&new_members,GrB_BOOL,n);
38     GrB_Vector_new(&new_neighbors,GrB_BOOL,n);
39     GrB_Vector_new(&candidates,GrB_BOOL,n);
40     GrB_Vector_new(iset,GrB_BOOL,n); // Initialize independent set vector, bool
41
42     GrB_UnaryOp set_random;
43     GrB_UnaryOp_new(&set_random,setRandom,GrB_FP32,GrB_UINT32);
44
45     // compute the degree of each vertex.
46     GrB_Vector degrees;
47     GrB_Vector_new(&degrees,GrB_FP64,n);
48     GrB_reduce(degrees,GrB_NULL,GrB_NULL,GrB_PLUS_FP64,A,GrB_NULL);
49
50     // Isolated vertices are not candidates: candidates[degrees != 0] = true
51     GrB_assign(candidates,degrees,GrB_NULL,true,GrB_ALL,n,GrB_NULL);
52
53     // add all singletons to iset: iset[degree == 0] = 1
54     GrB_assign(*iset,degrees,GrB_NULL,true,GrB_ALL,n,GrB_DESC_RC) ;
55
56     // Iterate while there are candidates to check.
57     GrB_Index nvals;
58     GrB_Vector_nvals(&nvals, candidates);
59     while (nvals > 0) {
60         // compute a random probability scaled by inverse of degree
61         GrB_apply(prob,candidates,GrB_NULL,set_random,degrees,GrB_DESC_R);
62     }

```

```

63 // compute the max probability of all neighbors
64 GrB_mnv(neighbor_max, candidates, GrB_NULL, GrB_MAX_SECOND_SEMIRING_FP32, A, prob, GrB_DESC_R);
65
66 // select vertex if its probability is larger than all its active neighbors,
67 // and apply a "masked no-op" to remove stored falses
68 GrB_eWiseAdd(new_members, GrB_NULL, GrB_NULL, GrB_GT_FP64, prob, neighbor_max, GrB_NULL);
69 GrB_apply(new_members, new_members, GrB_NULL, GrB_IDENTITY_BOOL, new_members, GrB_DESC_R);
70
71 // add new members to independent set.
72 GrB_eWiseAdd(*iset, GrB_NULL, GrB_NULL, GrB_LOR, *iset, new_members, GrB_NULL);
73
74 // remove new members from set of candidates  $c = c \ominus !new$ 
75 GrB_eWiseMult(candidates, new_members, GrB_NULL,
76               GrB_LAND, candidates, candidates, GrB_DESC_RC);
77
78 GrB_Vector_nvals(&nvals, candidates);
79 if (nvals == 0) { break; } // early exit condition
80
81 // Neighbors of new members can also be removed from candidates
82 GrB_mnv(new_neighbors, candidates, GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL,
83         A, new_members, GrB_NULL);
84 GrB_eWiseMult(candidates, new_neighbors, GrB_NULL, GrB_LAND,
85               candidates, candidates, GrB_DESC_RC);
86
87 GrB_Vector_nvals(&nvals, candidates);
88 }
89
90 GrB_free(&neighbor_max); // free all objects "new'ed"
91 GrB_free(&new_members);
92 GrB_free(&new_neighbors);
93 GrB_free(&prob);
94 GrB_free(&candidates);
95 GrB_free(&set_random);
96 GrB_free(&degrees);
97
98 return GrB_SUCCESS;
99 }

```

C.7 Example: Counting triangles in GraphBLAS

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given an  $n \times n$  boolean adjacency matrix,  $A$ , of an undirected graph, computes
9  * the number of triangles in the graph.
10 */
11 uint64_t triangle_count(GrB_Matrix A)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);           //  $n = \#$  of vertices
15
16     //  $L$ :  $N \times N$ , lower-triangular, bool
17     GrB_Matrix L;
18     GrB_Matrix_new(&L, GrB_BOOL, n, n);
19     GrB_select(L, GrB_NULL, GrB_NULL, GrB_TRIL, A, 0UL, GrB_NULL);
20
21     GrB_Matrix C;
22     GrB_Matrix_new(&C, GrB_UINT64, n, n);
23
24     GrB_mxm(C, L, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_UINT64, L, L, GrB_NULL); //  $C \langle L \rangle = L +.* L$ 
25
26     uint64_t count;
27     GrB_reduce(&count, GrB_NULL, GrB_PLUS_MONOID_UINT64, C, GrB_NULL); // 1-norm of  $C$ 
28
29     GrB_free(&C);
30     GrB_free(&L);
31
32     return count;
33 }
```