

The GraphBLAS C API Specification [†]:

Version 2.0.1

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

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

Generated on 2022/11/14 at 08:08:32 EDT

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

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

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

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

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

Contents

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

48	3.4	Algebraic objects, operators and associated functions	33
49	3.4.1	Operators	34
50	3.4.2	Monoids	39
51	3.4.3	Semirings	39
52	3.5	Collections	43
53	3.5.1	Scalars	43
54	3.5.2	Vectors	43
55	3.5.3	Matrices	44
56	3.5.3.1	External matrix formats	44
57	3.5.4	Masks	44
58	3.6	Fields	45
59	3.7	Descriptors	47
60	3.8	GrB_Info return values	47
61	4	Methods	51
62	4.1	Context methods	51
63	4.1.1	init: Initialize a GraphBLAS context	51
64	4.1.2	finalize: Finalize a GraphBLAS context	52
65	4.1.3	getVersion: Get the version number of the standard.	53
66	4.2	Object methods	53
67	4.2.1	Query methods	54
68	4.2.1.1	get: Query the value of an object	54
69	4.2.1.2	Descriptor_set: Set content of descriptor	55
70	4.2.2	Algebra methods	56
71	4.2.2.1	Type_new: Construct a new GraphBLAS (user-defined) type	56
72	4.2.2.2	UnaryOp_new: Construct a new GraphBLAS unary operator	57
73	4.2.2.3	BinaryOp_new: Construct a new GraphBLAS binary operator . . .	59
74	4.2.2.4	Monoid_new: Construct a new GraphBLAS monoid	60
75	4.2.2.5	Semiring_new: Construct a new GraphBLAS semiring	61
76	4.2.2.6	IndexUnaryOp_new: Construct a new GraphBLAS index unary op-	
77		erator [Scott: NEW CONTENT]	62

78	4.2.3	Scalar methods	64
79	4.2.3.1	Scalar_new: Construct a new scalar	64
80	4.2.3.2	Scalar_dup: Construct a copy of a GraphBLAS scalar	65
81	4.2.3.3	Scalar_clear: Clear/remove a stored value from a scalar	66
82	4.2.3.4	Scalar_nvals: Number of stored elements in a scalar	67
83	4.2.3.5	Scalar_setElement: Set the single element in a scalar	68
84	4.2.3.6	Scalar_extractElement: Extract a single element from a scalar. . . .	69
85	4.2.4	Vector methods	71
86	4.2.4.1	Vector_new: Construct new vector	71
87	4.2.4.2	Vector_dup: Construct a copy of a GraphBLAS vector	72
88	4.2.4.3	Vector_resize: Resize a vector	73
89	4.2.4.4	Vector_clear: Clear a vector	74
90	4.2.4.5	Vector_size: Size of a vector	75
91	4.2.4.6	Vector_nvals: Number of stored elements in a vector	75
92	4.2.4.7	Vector_build: Store elements from tuples into a vector	76
93	4.2.4.8	Vector_setElement: Set a single element in a vector	78
94	4.2.4.9	Vector_removeElement: Remove an element from a vector	80
95	4.2.4.10	Vector_extractElement: Extract a single element from a vector. . . .	81
96	4.2.4.11	Vector_extractTuples: Extract tuples from a vector	83
97	4.2.5	Matrix methods	84
98	4.2.5.1	Matrix_new: Construct new matrix	84
99	4.2.5.2	Matrix_dup: Construct a copy of a GraphBLAS matrix	86
100	4.2.5.3	Matrix_diag: Construct a diagonal GraphBLAS matrix	87
101	4.2.5.4	Matrix_resize: Resize a matrix	88
102	4.2.5.5	Matrix_clear: Clear a matrix	89
103	4.2.5.6	Matrix_nrows: Number of rows in a matrix	90
104	4.2.5.7	Matrix_ncols: Number of columns in a matrix	90
105	4.2.5.8	Matrix_nvals: Number of stored elements in a matrix	91
106	4.2.5.9	Matrix_build: Store elements from tuples into a matrix	92
107	4.2.5.10	Matrix_setElement: Set a single element in matrix	94

108	4.2.5.11	Matrix_removeElement: Remove an element from a matrix	96
109	4.2.5.12	Matrix_extractElement: Extract a single element from a matrix . . .	97
110	4.2.5.13	Matrix_extractTuples: Extract tuples from a matrix	99
111	4.2.5.14	Matrix_exportHint: Provide a hint as to which storage format might	
112		be most efficient for exporting a matrix	101
113	4.2.5.15	Matrix_exportSize: Return the array sizes necessary to export a	
114		GraphBLAS matrix object	102
115	4.2.5.16	Matrix_export: Export a GraphBLAS matrix to a pre-defined format	103
116	4.2.5.17	Matrix_import: Import a matrix into a GraphBLAS object	105
117	4.2.5.18	Matrix_serializeSize: Compute the serialize buffer size	107
118	4.2.5.19	Matrix_serialize: Serialize a GraphBLAS matrix.	108
119	4.2.5.20	Matrix_deserialize: Deserialize a GraphBLAS matrix.	109
120	4.2.6	Descriptor methods	110
121	4.2.6.1	Descriptor_new: Create new descriptor	110
122	4.2.6.2	Descriptor_set: Set content of descriptor	111
123	4.2.7	free: Destroy an object and release its resources	112
124	4.2.8	wait: Return once an object is either <i>complete</i> or <i>materialized</i>	114
125	4.2.9	error: Retrieve an error string	115
126	4.3	GraphBLAS operations	116
127	4.3.1	mxm: Matrix-matrix multiply	120
128	4.3.2	vxm: Vector-matrix multiply	125
129	4.3.3	mxv: Matrix-vector multiply	129
130	4.3.4	eWiseMult: Element-wise multiplication	133
131	4.3.4.1	eWiseMult: Vector variant	134
132	4.3.4.2	eWiseMult: Matrix variant	138
133	4.3.5	eWiseAdd: Element-wise addition	143
134	4.3.5.1	eWiseAdd: Vector variant	144
135	4.3.5.2	eWiseAdd: Matrix variant	148
136	4.3.6	extract: Selecting sub-graphs	154
137	4.3.6.1	extract: Standard vector variant	154
138	4.3.6.2	extract: Standard matrix variant	158

139	4.3.6.3	extract: Column (and row) variant	163
140	4.3.7	assign: Modifying sub-graphs	168
141	4.3.7.1	assign: Standard vector variant	168
142	4.3.7.2	assign: Standard matrix variant	173
143	4.3.7.3	assign: Column variant	179
144	4.3.7.4	assign: Row variant	184
145	4.3.7.5	assign: Constant vector variant[Scott: NEW CONTENT]	190
146	4.3.7.6	assign: Constant matrix variant[Scott: NEW CONTENT]	195
147	4.3.8	apply: Apply a function to the elements of an object	201
148	4.3.8.1	apply: Vector variant	201
149	4.3.8.2	apply: Matrix variant	206
150	4.3.8.3	apply: Vector-BinaryOp variants[Scott: NEW CONTENT]	210
151	4.3.8.4	apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]	216
152	4.3.8.5	apply: Vector index unary operator variant[Scott: NEW CONTENT]	222
153	4.3.8.6	apply: Matrix index unary operator variant[Scott: NEW CONTENT]	227
154	4.3.9	select:	232
155	4.3.9.1	select: Vector variant[Scott: NEW CONTENT]	232
156	4.3.9.2	select: Matrix variant[Scott: NEW CONTENT]	237
157	4.3.10	reduce: Perform a reduction across the elements of an object	243
158	4.3.10.1	reduce: Standard matrix to vector variant	243
159	4.3.10.2	reduce: Vector-scalar variant[Scott: NEW CONTENT]	247
160	4.3.10.3	reduce: Matrix-scalar variant[Scott: NEW CONTENT]	251
161	4.3.11	transpose: Transpose rows and columns of a matrix	254
162	4.3.12	kroncker: Kronecker product of two matrices	258
163	5	Nonpolymorphic interface[Scott: NEW CONTENT]	265
164	A	Revision history	277
165	B	Non-opaque data format definitions	283
166	B.1	GrB_Format: Specify the format for input/output of a GraphBLAS matrix.	283
167	B.1.1	GrB_CSR_FORMAT	283

168	B.1.2 GrB_CSC_FORMAT	284
169	B.1.3 GrB_COO_FORMAT	284
170	C Examples	285
171	C.1 Example: Level breadth-first search (BFS) in GraphBLAS	286
172	C.2 Example: Level BFS in GraphBLAS using apply	287
173	C.3 Example: Parent BFS in GraphBLAS	288
174	C.4 Example: Betweenness centrality (BC) in GraphBLAS	289
175	C.5 Example: Batched BC in GraphBLAS	291
176	C.6 Example: Maximal independent set (MIS) in GraphBLAS	293
177	C.7 Example: Counting triangles in GraphBLAS	295

List of Tables

178		
179	2.1	Types of GraphBLAS opaque objects. 23
180	2.2	Methods that forced completion prior to GraphBLAS v2.0. 28
181	3.1	Enumeration literals and corresponding values input to various GraphBLAS methods. 32
182	3.2	Predefined GrB_Type values. 33
183	3.3	Operator input for relevant GraphBLAS operations. 34
184	3.4	Properties and recipes for building GraphBLAS algebraic objects. 35
185	3.5	Predefined unary and binary operators for GraphBLAS in C. 37
186	3.6	Predefined index unary operators for GraphBLAS in C. 38
187	3.7	Predefined monoids for GraphBLAS in C. 40
188	3.8	Predefined “true” semirings for GraphBLAS in C. 41
189	3.9	Other useful predefined semirings for GraphBLAS in C. 42
190	3.10	GrB_Format enumeration literals and corresponding values for matrix import and
191		export methods. 44
192	3.11	Field values of type GrB_Field enumeration, corresponding types, and the objects
193		which must implement that GrB_Field. Collection refers to GrB_Matrix, GrB_Vector,
194		and GrB_Scalar, Algebraic refers to Operators, Monoids, and Semirings, while All refers
195		to all GraphBLAS objects. Global fields are denoted by Global. All fields may be
196		read, some may be written (denoted by W), and some are hints (denoted by H) which
197		may be ignored by the implementation. 46
198	3.12	Descriptor types and literals for fields and values. 48
199	3.13	Predefined GraphBLAS descriptors. 49
200	3.14	Enumeration literals and corresponding values returned by GraphBLAS methods
201		and operations. 50
202	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in
203		this specification. 117

204	5.1	Long-name, nonpolymorphic form of GraphBLAS methods.	265
205	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	266
206	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	267
207	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	268
208	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	269
209	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	270
210	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	271
211	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	272
212	5.9	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
213		CONTENT]	273
214	5.10	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
215		CONTENT]	274
216	5.11	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	275

217 **List of Figures**

218	3.1 Hierarchy of algebraic object classes in GraphBLAS.	43
219	4.1 Flowchart for the GraphBLAS operations.	118
220	B.1 Data layout for CSR format.	283
221	B.2 Data layout for CSC format.	284
222	B.3 Data layout for COO format.	284

Acknowledgments

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

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

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

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

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

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

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

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

Chapter 1

Introduction

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

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

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

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

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

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

is not possible to express the signature of that method.

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

The remainder of this document is organized as follows:

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

Chapter 2

Basic concepts

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

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

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

2.1 Glossary

2.1.1 GraphBLAS API basic definitions

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

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

2.1.2 GraphBLAS objects and their structure

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

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

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

2.1.3 Algebraic structures used in the GraphBLAS

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

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

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

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

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

2.1.4 The execution of an application using the GraphBLAS C API

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

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

2.1.5 GraphBLAS methods: behaviors and error conditions

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

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

- 473 • *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as param-
474 eters to a GraphBLAS method are dimension (or shape) compatible if they have the correct
475 number of dimensions and sizes for each dimension to satisfy the rules of the mathematical def-
476 inition of the operation associated with the method. If any *dimension compatibility* rule above
477 is violated, execution of the GraphBLAS method ends and the GrB_DIMENSION_MISMATCH
478 error is returned.
- 479 • *domain compatible*: Two domains for which values from one domain can be cast to values in
480 the other domain as per the rules of the C language. In particular, domains from Table 3.2
481 are all compatible with each other, and a domain from a user-defined type is only compatible
482 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS
483 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 3.14. 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 3.14(a) are non-negative and include `GrB_SUCCESS` (a value of 0) and `GrB_NO_VALUE`.

An API error (listed in Table 3.14(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 3.14(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

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

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

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

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

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

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

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

788 3.3 Types (domains)

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

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

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

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

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

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

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

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

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

3.4 Algebraic objects, operators and associated functions

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

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

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

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

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

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

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

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

3.4.1 Operators

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

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

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

(a) Properties of algebraic objects.

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

(b) Recipes for algebraic objects.

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

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

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

834 D_{in_2} , and an operation $\odot : D_{in_1} \times D_{in_2} \rightarrow D_{out}$. For a given GraphBLAS binary operator $F_b =$
835 $\langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$, 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 $\odot(F_b) =$
836 \odot . Note that \odot could be used in place of either \oplus or \otimes in other methods and operations.

837 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
838 domains, D_{out} , D_{in_1} , D_{in_2} , the domain of GraphBLAS indices, and an operation $f_i : D_{in_1} \times I_{U64}^2 \times$
839 $D_{in_2} \rightarrow D_{out}$ (where I_{U64} corresponds to the domain of a `GrB_Index`). For a given GraphBLAS
840 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 $\mathbf{f}(F_i) = f_i$.

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

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

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

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

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

3.4.2 Monoids

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

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

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

3.4.3 Semirings

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

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

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

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

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

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

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

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

GraphBLAS identifier	Domains, T ($T \times T \rightarrow T$)	Identity	Description
GrB_PLUS_MONOID_ T	UINT x INT x FP x	0 0 0	addition
GrB_TIMES_MONOID_ T	UINT x INT x FP x	1 1 1	multiplication
GrB_MIN_MONOID_ T	UINT x INT x FP x	UINT x _MAX INT x _MAX INFINITY	minimum
GrB_MAX_MONOID_ T	UINT x INT x FP x	0 INT x _MIN -INFINITY	maximum
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
	$\text{INT}x$	$\text{INT}x_MIN$	
	$\text{FP}x$	$-INFINITY$	

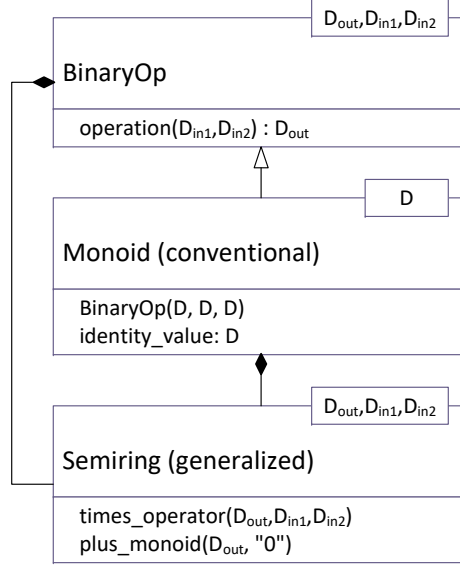


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

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 Fields

GraphBLAS objects and implementations contain internal fields which may provide information to users and allow setting runtime parameters and hints. All GraphBLAS objects are required to implement the **get** and **set** methods required to query and set these fields.

A GraphBLAS object may contain a number of (*field*, *value*) pairs, where the *value* type is determined by the *field*. Objects must implement a set of such pairs as determined by the specification, but may extend that set with implementation specific pairs.

The GraphBLAS implementation itself contains several (*field*, *value*) pairs, which provide defaults to object level fields, and implementation information such as the version number or implementation name.

There are three

Table 3.11: 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.

(a) Types used with GraphBLAS descriptors.

Field Name	W H	Value	Implementing Objects	Type
GrB_OUTP	W	0	GrB_Descriptor	GrB_Desc_Value
GrB_MASK	W	1	GrB_Descriptor	GrB_Desc_Value
GrB_INP0	W	2	GrB_Descriptor	GrB_Desc_Value
GrB_INP1	W	3	GrB_Descriptor	GrB_Desc_Value
GrB_NAMESIZE	—	10	All	GrB_Index
GrB_NAME	*	11	All	Null terminated char* of size GrB_NAMESIZE Minimum supported size of 512-bytes
GrB_LIBRARY_NAME	—	100	Global	256-byte null terminated char*
GrB_LIBRARY_VER	—	101	Global	Length 3 integer array
GrB_API_VER	—	102	Global	Length 3 integer array
GrB_BLOCKING_MODE	—	103	Global	GrB_Mode
GrB_NTHREADS	W	104	Global, GrB_Descriptor	GrB_Index
GrB_STORAGE_ORIENTATION_HINT	WH	200	Global, Collection	GrB_ROWMAJOR, GrB_COLMAJOR
GrB_STORAGE_FORMAT_HINT	WH	201	Collection	GrB_Format
GrB_ELTYPE??	—	202	Collection	GrB_Type
GrB_INPUT1TYPE??	—	300	Algebraic	GrB_Type
GrB_INPUT2TYPE??	—	301	Algebraic	GrB_Type
GrB_OUTPUTTYPE??	—	302	Algebraic	GrB_Type
GrB_BINARYOP??	—	303	GrB_Monoid, GrB_Semiring	GrB_BinaryOp
GrB_MONOID??	—	304	GrB_Semiring	GrB_Monoid

3.7 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 3.12.

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 3.13.

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 3.14.

Table 3.12: Descriptors are GraphBLAS objects passed as arguments to GraphBLAS operations to modify other GraphBLAS objects in the operation’s argument list. A descriptor, `desc`, has one or more (*field*, *value*) pairs indicated as `desc[GrB_Desc_Field].GrB_Desc_Value`. In this table, we define all types and literals used with descriptors.

(a) Types used with GraphBLAS descriptors.

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

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

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

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

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

Table 3.13: Predefined GraphBLAS descriptors. The list includes all possible descriptors, according to the current standard. Columns list the possible fields and entries list the value(s) associated with those fields for a given descriptor.

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

Table 3.14: 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.

(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`.

1017 **Return Values**

1018 `GrB_SUCCESS` operation completed successfully.

1019 `GrB_PANIC` unknown internal error.

1020 `GrB_INVALID_VALUE` invalid mode specified, or method called multiple times.

1021 **Description**

1022 The `init` method creates and initializes a GraphBLAS C API context. The argument to `GrB_init`
1023 defines the mode for the context. The two available modes are:

- 1024 • `GrB_BLOCKING`: In this mode, each method in a sequence returns after its computations have
1025 completed and output arguments are available to subsequent statements in an application.
1026 When executing in `GrB_BLOCKING` mode, the methods execute in program order.
- 1027 • `GrB_NONBLOCKING`: In this mode, methods in a sequence may return after arguments in
1028 the method have been tested for dimension and domain compatibility within the method
1029 but potentially before their computations complete. Output arguments are available to sub-
1030 sequent GraphBLAS methods in an application. When executing in `GrB_NONBLOCKING`
1031 mode, the methods in a sequence may execute in any order that preserves the mathematical
1032 result defined by the sequence.

1033 An application can only create one context per execution instance. An application may only call
1034 `GrB_Init` once. Calling `GrB_Init` more than once results in undefined behavior.

1035 **4.1.2 finalize: Finalize a GraphBLAS context**

1036 Terminates and frees any internal resources created to support the GraphBLAS C API context.

1037 **C Syntax**

1038 `GrB_Info GrB_finalize();`

1039 **Return Values**

1040 `GrB_SUCCESS` operation completed successfully.

1041 `GrB_PANIC` unknown internal error.

1042 **Description**

1043 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS
1044 C API context. `GrB_finalize` may only be called after a context has been initialized by calling
1045 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-
1046 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined
1047 behavior.

1048 **4.1.3 getVersion: Get the version number of the standard.**

1049 Query the library for the version number of the standard that this library implements.

1050 **C Syntax**

```
1051         GrB_Info GrB_getVersion(unsigned int *version,  
1052                                unsigned int *subversion);
```

1053 **Parameters**

1054 version (OUT) On successful return will hold the value of the major version number.

1055 version (OUT) On successful return will hold the value of the subversion number.

1056 **Return Values**

1057 GrB_SUCCESS operation completed successfully.

1058 GrB_PANIC unknown internal error.

1059 **Description**

1060 The `getVersion` method is used to query the major and minor version number of the GraphBLAS
1061 C API specification that the library implements at runtime. To support compile time queries the
1062 following two macros shall also be defined by the library.

```
1063         #define GRB_VERSION      2  
1064         #define GRB_SUBVERSION  0
```

1065 **4.2 Object methods**

1066 This section describes methods that setup and operate on GraphBLAS opaque objects but are not
1067 part of the the GraphBLAS math specification.

1068 4.2.1 Query methods

1069 The methods in this section query and, depending on the field, set internal fields of many Graph-
1070 BLAS objects.

1071 4.2.1.1 get: Query the value of an object

1072 C Syntax

```
1073     GrB_Info GrB_<OBJ>_get(GrB_<OBJ> o, GrB_Field field, ...);
1074
1075     GrB_Info GrB_Scalar_get(GrB_Scalar s, GrB_Field field, ...);
1076     GrB_Info GrB_Vector_get(GrB_Vector v, GrB_Field field, ...);
1077     GrB_Info GrB_Matrix_get(GrB_Matrix A, GrB_Field field, ...);
1078
1079     GrB_Info GrB_UnaryOp_get(GrB_UnaryOp op, GrB_Field field, ...);
1080     GrB_Info GrB_IndexUnaryOp_get(GrB_IndexUnaryOp op, GrB_Field field, ...);
1081     GrB_Info GrB_BinaryOp_get(GrB_BinaryOp op, GrB_Field field, ...);
1082     GrB_Info GrB_Monoid_get(GrB_Monoid op, GrB_Field field, ...);
1083     GrB_Info GrB_Semiring_get(GrB_Semiring op, GrB_Field field, ...);
1084
1085     GrB_Info GrB_Descriptor_get(GrB_Descriptor op, GrB_Field field, ...);
1086     GrB_Info GrB_Type_get(GrB_Type op, GrB_Field field, ...);
1087
1088     GrB_Info GrB_Global_get(GrB_Field field, ...);
```

1089 Parameters

1090 OBJ is replaced in each signature by the object type being queried.

1091 OBJ (IN) An existing GraphBLAS object which is being queried.

1092 field (IN) The internal field being queried.

1093 ... (OUT) A pointer to a variable dependent on field to be filled with the value of the
1094 internal field.

1095 Return Value

1096 GrB_SUCCESS The method completed successfully.

1097 GrB_PANIC unknown internal error.

1098 GrB_OUT_OF_MEMORY not enough memory available for operation.

1099 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to new.

1100 GrB_INVALID_VALUE invalid value set on the field, or invalid field.

1101 Description

1102 Queries a field of an existing GraphBLAS object.

1103 4.2.1.2 Descriptor_set: Set content of descriptor

1104 Sets the content for a field for an existing descriptor.

1105 C Syntax

```
1106      GrB_Info GrB_Descriptor_set(GrB_Descriptor      desc,  
1107                                GrB_Desc_Field      field,  
1108                                GrB_Desc_Value      val);
```

1109 Parameters

1110 desc (IN) An existing GraphBLAS descriptor to be modified.

1111 field (IN) The field being set.

1112 val (IN) New value for the field being set.

1113 Return Values

1114 GrB_SUCCESS operation completed successfully.

1115 GrB_PANIC unknown internal error.

1116 GrB_OUT_OF_MEMORY not enough memory available for operation.

1117 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to new.

1118 GrB_INVALID_VALUE invalid value set on the field, or invalid field.

1119 Description

1120 For a given descriptor, the GrB_Descriptor_set method can be called for each field in the descriptor
1121 to set the value associated with that field. Valid values for the field parameter include the following:

1122 GrB_OUTP refers to the output parameter (result) of the operation.

1123 GrB_MASK refers to the mask parameter of the operation.

1124 GrB_INP0 refers to the first input parameters of the operation (matrices and vectors).

1125 GrB_INP1 refers to the second input parameters of the operation (matrices and vectors).

1126 Valid values for the val parameter are:

1127 GrB_STRUCTURE Use only the structure of the stored values of the corresponding mask
1128 (GrB_MASK) parameter.

1129 GrB_COMP Use the complement of the corresponding mask (GrB_MASK) param-
1130 eter. When combined with GrB_STRUCTURE, the complement of the
1131 structure of the mask is used without evaluating the values stored.

1132 GrB_TRAN Use the transpose of the corresponding matrix parameter (valid for input
1133 matrix parameters only).

1134 GrB_REPLACE When assigning the masked values to the output matrix or vector, clear
1135 the matrix first (or clear the non-masked entries). The default behavior
1136 is to leave non-masked locations unchanged. Valid for the GrB_OUTP
1137 parameter only.

1138 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of GrB_MASK,
1139 multiple values can be set and all will apply (for example, both GrB_COMP and GrB_STRUCTURE).
1140 A value for a given field may be set multiple times but will have no additional effect. Fields that
1141 have no values set result in their default behavior, as defined in Section 3.7.

1142 4.2.2 Algebra methods

1143 4.2.2.1 Type_new: Construct a new GraphBLAS (user-defined) type

1144 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,
1145 monoids, semirings, vectors and matrices.

1146 C Syntax

```
1147       GrB_Info GrB_Type_new(GrB_Type   *utype,
1148                                       size_t       sizeof(ctype));
```

1149 Parameters

1150 utype (INOUT) On successful return, contains a handle to the newly created user-defined
1151 GraphBLAS type object.

1152 ctype (IN) A C type that defines the new GraphBLAS user-defined type.

1153 Return Values

1154 GrB_SUCCESS operation completed successfully.

1155 GrB_PANIC unknown internal error.

1156 GrB_OUT_OF_MEMORY not enough memory available for operation.

1157 GrB_NULL_POINTER utype pointer is NULL.

1158 Description

1159 Given a C type `ctype`, the `Type_new` method returns in `utype` a handle to a new GraphBLAS type
1160 that is equivalent to the C type. Variables of this `ctype` must be a struct, union, or fixed-size array.
1161 In particular, given two variables, `src` and `dst`, of type `ctype`, the following operation must be a
1162 valid way to copy the contents of `src` to `dst`:

1163 `memcpy(&dst, &src, sizeof(ctype))`

1164 A new, user-defined type `utype` should be destroyed with a call to `GrB_free(utype)` when no longer
1165 needed.

1166 It is not an error to call this method more than once on the same variable; however, the handle to
1167 the previously created object will be overwritten.

1168 4.2.2.2 UnaryOp_new: Construct a new GraphBLAS unary operator

1169 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types
1170 (domains).

1171 C Syntax

```
1172           GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1173                                   void           (*unary_func)(void*, const void*),  
1174                                   GrB_Type       d_out,  
1175                                   GrB_Type       d_in);
```

1176 Parameters

1177 `unary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1178 unary operator object.

1179 `unary_func` (IN) a pointer to a user-defined function that takes one input parameter of `d_in`'s
1180 type and returns a value of `d_out`'s type, both passed as `void` pointers. Specifically
1181 the signature of the function is expected to be of the form:

```

1182         void func(void *out, const void *in);
1183

```

1184 **d_out** (IN) The GrB_Type of the return value of the unary operator being created. Should
1185 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1186 BLAS type.

1187 **d_in** (IN) The GrB_Type of the input argument of the unary operator being created.
1188 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1189 GraphBLAS type.

1190 Return Values

1191 GrB_SUCCESS operation completed successfully.

1192 GrB_PANIC unknown internal error.

1193 GrB_OUT_OF_MEMORY not enough memory available for operation.

1194 GrB_UNINITIALIZED_OBJECT any GrB_Type parameter (for user-defined types) has not been ini-
1195 tialized by a call to GrB_Type_new.

1196 GrB_NULL_POINTER unary_op or unary_func pointers are NULL.

1197 Description

1198 The UnaryOp_new method creates a new GraphBLAS unary operator

1199 $f_u = \langle \mathbf{D}(\mathbf{d_out}), \mathbf{D}(\mathbf{d_in}), \text{unary_func} \rangle$

1200 and returns a handle to it in unary_op.

1201 The implementation of unary_func must be such that it works even if the d_out and d_in arguments
1202 are aliased. In other words, for all invocations of the function:

```

1203     unary_func(out, in);

```

1204 the value of out must be the same as if the following code was executed:

```

1205     D(d_in) *tmp = malloc(sizeof(D(d_in)));
1206     memcpy(tmp, in, sizeof(D(d_in)));
1207     unary_func(out, tmp);
1208     free(tmp);

```

1209 It is not an error to call this method more than once on the same variable; however, the handle to
1210 the previously created object will be overwritten.

1211 4.2.2.3 BinaryOp_new: Construct a new GraphBLAS binary operator

1212 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types
1213 (domains).

1214 C Syntax

```
1215         GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
1216                                   void          (*binary_func)(void*,  
1217                                                           const void*,  
1218                                                           const void*),  
1219                                   GrB_Type      d_out,  
1220                                   GrB_Type      d_in1,  
1221                                   GrB_Type      d_in2);
```

1222 Parameters

1223 binary_op (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1224 binary operator object.

1225 binary_func (IN) A pointer to a user-defined function that takes two input parameters of types
1226 d_in1 and d_in2 and returns a value of type d_out, all passed as void pointers.
1227 Specifically the signature of the function is expected to be of the form:

```
1228         void func(void *out, const void *in1, const void *in2);
```

1230 d_out (IN) The GrB_Type of the return value of the binary operator being created. Should
1231 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-
1232 BLAS type.

1233 d_in1 (IN) The GrB_Type of the left hand argument of the binary operator being created.
1234 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1235 GraphBLAS type.

1236 d_in2 (IN) The GrB_Type of the right hand argument of the binary operator being cre-
1237 ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-
1238 defined GraphBLAS type.

1239 Return Values

1240 GrB_SUCCESS operation completed successfully.

1241 GrB_PANIC unknown internal error.

1242 GrB_OUT_OF_MEMORY not enough memory available for operation.

1243 GrB_UNINITIALIZED_OBJECT the GrB_Type (for user-defined types) has not been initialized by a
1244 call to GrB_Type_new.

1245 GrB_NULL_POINTER binary_op or binary_func pointer is NULL.

1246 Description

1247 The BinaryOp_new method creates a new GraphBLAS binary operator

1248 $f_b = \langle \mathbf{D}(\mathbf{d_out}), \mathbf{D}(\mathbf{d_in1}), \mathbf{D}(\mathbf{d_in2}), \text{binary_func} \rangle$

1249 and returns a handle to it in binary_op.

1250 The implementation of binary_func must be such that it works even if any of the d_out, d_in1, and
1251 d_in2 arguments are aliased to each other. In other words, for all invocations of the function:

1252 `binary_func(out, in1, in2);`

1253 the value of out must be the same as if the following code was executed:

```
1254     D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));  
1255     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));  
1256     memcpy(tmp1, in1, sizeof(D(d_in1)));  
1257     memcpy(tmp2, in2, sizeof(D(d_in2)));  
1258     binary_func(out, tmp1, tmp2);  
1259     free(tmp2);  
1260     free(tmp1);
```

1261 It is not an error to call this method more than once on the same variable; however, the handle to
1262 the previously created object will be overwritten.

1263 4.2.2.4 Monoid_new: Construct a new GraphBLAS monoid

1264 Creates a new monoid with specified binary operator and identity value.

1265 C Syntax

```
1266     GrB_Info GrB_Monoid_new(GrB_Monoid *monoid,  
1267                             GrB_BinaryOp binary_op,  
1268                             <type>      identity);
```

1269 Parameters

1270 **monoid** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1271 monoid object.

1272 **binary_op** (IN) An existing GraphBLAS associative binary operator whose input and output
1273 types are the same.

1274 **identity** (IN) The value of the identity element of the monoid. Must be the same type as
1275 the type used by the **binary_op** operator.

1276 Return Values

1277 **GrB_SUCCESS** operation completed successfully.

1278 **GrB_PANIC** unknown internal error.

1279 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1280 **GrB_UNINITIALIZED_OBJECT** the **GrB_BinaryOp** (for user-defined operators) has not been initial-
1281 ized by a call to **GrB_BinaryOp_new**.

1282 **GrB_NULL_POINTER** monoid pointer is NULL.

1283 **GrB_DOMAIN_MISMATCH** all three argument types of the binary operator and the type of the
1284 identity value are not the same.

1285 Description

1286 The **Monoid_new** method creates a new monoid $M = \langle \mathbf{D}(\text{binary_op}), \text{binary_op}, \text{identity} \rangle$ and re-
1287 turns a handle to it in **monoid**.

1288 If **binary_op** is not associative, the results of GraphBLAS operations that require associativity of
1289 this monoid will be undefined.

1290 It is not an error to call this method more than once on the same variable; however, the handle to
1291 the previously created object will be overwritten.

1292 4.2.2.5 Semiring_new: Construct a new GraphBLAS semiring

1293 Creates a new semiring with specified domain, operators, and elements.

1294 C Syntax

```
1295        GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1296                                    GrB_Monoid    add_op,  
1297                                    GrB_BinaryOp   mul_op);
```

1298 Parameters

- 1299 **semiring** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1300 semiring.
- 1301 **add_op** (IN) An existing GraphBLAS commutative monoid that specifies the addition op-
1302 erator and its identity.
- 1303 **mul_op** (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-
1304 plication operator. In addition, **mul_op**'s output domain, $\mathbf{D}_{out}(\text{mul_op})$, must be
1305 the same as the **add_op**'s domain $\mathbf{D}(\text{add_op})$.

1306 Return Values

- 1307 **GrB_SUCCESS** operation completed successfully.
- 1308 **GrB_PANIC** unknown internal error.
- 1309 **GrB_OUT_OF_MEMORY** not enough memory available for this method to complete.
- 1310 **GrB_UNINITIALIZED_OBJECT** the **add_op** (for user-define monoids) object has not been initialized
1311 with a call to **GrB_Monoid_new** or the **mul_op** (for user-defined
1312 operators) object has not been not been initialized by a call to
1313 **GrB_BinaryOp_new**.
- 1314 **GrB_NULL_POINTER** semiring pointer is NULL.
- 1315 **GrB_DOMAIN_MISMATCH** the output domain of **mul_op** does not match the domain of the
1316 **add_op** monoid.

1317 Description

1318 The **Semiring_new** method creates a new semiring:

$$1319 \quad S = \langle \mathbf{D}_{out}(\text{mul_op}), \mathbf{D}_{in_1}(\text{mul_op}), \mathbf{D}_{in_2}(\text{mul_op}), \text{add_op}, \text{mul_op}, \mathbf{0}(\text{add_op}) \rangle$$

1320 and returns a handle to it in **semiring**. Note that $\mathbf{D}_{out}(\text{mul_op})$ must be the same as $\mathbf{D}(\text{add_op})$.

1321 If **add_op** is not commutative, then GraphBLAS operations using this semiring will be undefined.

1322 It is not an error to call this method more than once on the same variable; however, the handle to
1323 the previously created object will be overwritten.

1324 4.2.2.6 IndexUnaryOp_new: Construct a new GraphBLAS index unary operator [Scott: 1325 NEW CONTENT]

1326 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its
1327 types (domains).

1328 C Syntax

```
1329     GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp  *index_unary_op,  
1330                                   void (*index_unary_func)(void*,  
1331                                                             const void*,  
1332                                                             GrB_Index,  
1333                                                             GrB_Index,  
1334                                                             const void*),  
1335                                   GrB_Type          d_out,  
1336                                   GrB_Type          d_in1,  
1337                                   GrB_Type          d_in2);
```

1338 Parameters

1339 **index_unary_op** (INOUT) On successful return, contains a handle to the newly created Graph-
1340 BLAS index unary operator object.

1341 **index_unary_func** (IN) A pointer to a user-defined function that takes input parameters of types
1342 **d_in1**, **GrB_Index**, **GrB_Index** and **d_in2** and returns a value of type **d_out**. Ex-
1343 cept for the **GrB_Index** parameters, all are passed as **void** pointers. Specifically
1344 the signature of the function is expected to be of the form:

```
1345         void func(void      *out,  
1346                   const void *in1,  
1347                   GrB_Index  row_index,  
1348                   GrB_Index  col_index,  
1349                   const void *in2);  
1350
```

1351 **d_out** (IN) The **GrB_Type** of the return value of the index unary operator being created.
1352 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined
1353 GraphBLAS type.

1354 **d_in1** (IN) The **GrB_Type** of the first input argument of the index unary operator being
1355 created and corresponds to the stored values of the **GrB_Vector** or **GrB_Matrix**
1356 being operated on. Should be one of the predefined GraphBLAS types in Ta-
1357 ble 3.2, or a user-defined GraphBLAS type.

1358 **d_in2** (IN) The **GrB_Type** of the last input argument of the index unary operator be-
1359 ing created and corresponds to a scalar provided by the GraphBLAS operation
1360 that uses this operator. Should be one of the predefined GraphBLAS types in
1361 Table 3.2, or a user-defined GraphBLAS type.

1362 Return Values

1363 **GrB_SUCCESS** operation completed successfully.

1392 C Syntax

```
1393         GrB_Info GrB_Scalar_new(GrB_Scalar *s,  
1394                                 GrB_Type    d);
```

1395 Parameters

1396 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1397 scalar.

1398 **d** (IN) The type corresponding to the domain of the scalar being created. Can be
1399 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1400 GraphBLAS type.

1401 Return Values

1402 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1403 blocking mode, this indicates that the API checks for the input
1404 arguments passed successfully. Either way, output scalar **s** is ready
1405 to be used in the next method of the sequence.

1406 **GrB_PANIC** Unknown internal error.

1407 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1408 GraphBLAS objects (input or output) is in an invalid state caused
1409 by a previous execution error. Call **GrB_error()** to access any error
1410 messages generated by the implementation.

1411 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1412 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
1413 (needed for user-defined types).

1414 **GrB_NULL_POINTER** The **s** pointer is NULL.

1415 Description

1416 Creates a new GraphBLAS scalar **s** of domain **D(d)** and empty **L(s)**. The method returns a handle
1417 to the new scalar in **s**.

1418 It is not an error to call this method more than once on the same variable; however, the handle to
1419 the previously created object will be overwritten.

1420 4.2.3.2 Scalar_dup: Construct a copy of a GraphBLAS scalar

1421 Creates a new scalar with the same domain and contents as another scalar.

1422 C Syntax

```
1423         GrB_Info GrB_Scalar_dup(GrB_Scalar      *t,  
1424                                 const GrB_Scalar  s);
```

1425 Parameters

1426 **t** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1427 scalar.

1428 **s** (IN) The GraphBLAS scalar to be duplicated.

1429 Return Values

1430 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1431 blocking mode, this indicates that the API checks for the input
1432 arguments passed successfully. Either way, output scalar **t** is ready
1433 to be used in the next method of the sequence.

1434 **GrB_PANIC** Unknown internal error.

1435 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1436 GraphBLAS objects (input or output) is in an invalid state caused
1437 by a previous execution error. Call **GrB_error()** to access any error
1438 messages generated by the implementation.

1439 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1440 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to
1441 **Scalar_new** or **Scalar_dup**.

1442 **GrB_NULL_POINTER** The **t** pointer is NULL.

1443 Description

1444 Creates a new scalar *t* of domain **D(s)** and contents **L(s)**. The method returns a handle to the new
1445 scalar in **t**.

1446 It is not an error to call this method more than once with the same output variable; however, the
1447 handle to the previously created object will be overwritten.

1448 4.2.3.3 **Scalar_clear**: Clear/remove a stored value from a scalar

1449 Removes the stored value from a scalar.

1450 C Syntax

```
1451      GrB_Info GrB_Scalar_clear(GrB_Scalar s);
```

1452 Parameters

1453 `s` (INOUT) An existing GraphBLAS scalar to clear.

1454 Return Values

1455 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1456 blocking mode, this indicates that the API checks for the input
1457 arguments passed successfully. Either way, output scalar `s` is ready
1458 to be used in the next method of the sequence.

1459 `GrB_PANIC` Unknown internal error.

1460 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1461 GraphBLAS objects (input or output) is in an invalid state caused
1462 by a previous execution error. Call `GrB_error()` to access any error
1463 messages generated by the implementation.

1464 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1465 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to
1466 `Scalar_new` or `Scalar_dup`.

1467 Description

1468 Removes the stored value from an existing scalar. After the call, `L(s)` is empty. The size of the
1469 scalar does not change.

1470 4.2.3.4 `Scalar_nvals`: Number of stored elements in a scalar

1471 Retrieve the number of stored elements in a scalar (either zero or one).

1472 C Syntax

```
1473      GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,  
1474                                const GrB_Scalar s);
```

1475

1476

1477

1478

1479

1480

1481

1483

1484

1485

1496

1490

1491

1492

1493

1494

1495

1496

1497

1498

1499

1500

1501

1502 Return Values

1503 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1504 blocking mode, this indicates that the compatibility tests on in-
1505 dex/dimensions and domains for the input arguments passed suc-
1506 cessfully. Either way, the output scalar `s` is ready to be used in the
1507 next method of the sequence.

1508 GrB_PANIC Unknown internal error.

1509 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1510 GraphBLAS objects (input or output) is in an invalid state caused
1511 by a previous execution error. Call `GrB_error()` to access any error
1512 messages generated by the implementation.

1513 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1514 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, `s`, has not been initialized by a call to
1515 `Scalar_new` or `Scalar_dup`.

1516 GrB_DOMAIN_MISMATCH The domains of `s` and `val` are incompatible.

1517 Description

1518 First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: `D(val)` must
1519 be compatible with `D(s)`. Two domains are compatible with each other if values from one domain
1520 can be cast to values in the other domain as per the rules of the C language. In particular, domains
1521 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-
1522 patible with itself. If any compatibility rule above is violated, execution of `GrB_Scalar_setElement`
1523 ends and the domain mismatch error listed above is returned.

1524 We are now ready to carry out the assignment `val`; that is:

$$1525 \qquad s(0) = \text{val}$$

1526 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1527 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1528 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1529 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be
1530 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1531 4.2.3.6 `Scalar_extractElement`: Extract a single element from a scalar.

1532 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

1533 C Syntax

```
1534      GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1535                                         const GrB_Scalar s);
```

1536 Parameters

1537 **val** (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain
1538 of scalar **s**. On successful return, **val** holds the result of the operation, and any
1539 previous value in **val** is overwritten.

1540 **s** (IN) The GraphBLAS scalar from which an element is extracted.

1541 Return Values

1542 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1543 cessfully. This indicates that the compatibility tests on dimensions
1544 and domains for the input arguments passed successfully, and the
1545 output scalar, **val**, has been computed and is ready to be used in
1546 the next method of the sequence.

1547 **GrB_PANIC** Unknown internal error.

1548 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1549 GraphBLAS objects (input or output) is in an invalid state caused
1550 by a previous execution error. Call **GrB_error()** to access any error
1551 messages generated by the implementation.

1552 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1553 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to
1554 **Scalar_new** or **Scalar_dup**.

1555 **GrB_NULL_POINTER** **val** pointer is NULL.

1556 **GrB_DOMAIN_MISMATCH** The domains of the scalar or scalar are incompatible.

1557 **GrB_NO_VALUE** There is no stored value in the scalar.

1558 Description

1559 First, **val** and input GraphBLAS scalar are tested for domain compatibility as follows: **D(val)**
1560 must be compatible with **D(s)**. Two domains are compatible with each other if values from
1561 one domain can be cast to values in the other domain as per the rules of the C language. In
1562 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1563 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1564 **GrB_Scalar_extractElement** ends and the domain mismatch error listed above is returned.

1565 Then, if no value is currently stored in the GraphBLAS scalar, the method returns `GrB_NO_VALUE`
1566 and `val` remains unchanged.

1567 Finally the extract into the output argument, `val` can be performed; that is:

1568
$$\text{val} = \text{s}(0)$$

1569 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1570 `GrB_SUCCESS`, the new contents of `val` are as defined above.

1571 4.2.4 Vector methods

1572 4.2.4.1 `Vector_new`: Construct new vector

1573 Creates a new vector with specified domain and size.

1574 C Syntax

```
1575     GrB_Info GrB_Vector_new(GrB_Vector *v,  
1576                             GrB_Type    d,  
1577                             GrB_Index   nsize);
```

1578 Parameters

1579 `v` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1580 vector.

1581 `d` (IN) The type corresponding to the domain of the vector being created. Can be
1582 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1583 GraphBLAS type.

1584 `nsize` (IN) The size of the vector being created.

1585 Return Values

1586 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1587 blocking mode, this indicates that the API checks for the input
1588 arguments passed successfully. Either way, output vector `v` is ready
1589 to be used in the next method of the sequence.

1590 `GrB_PANIC` Unknown internal error.

1591 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1592 GraphBLAS objects (input or output) is in an invalid state caused
1593 by a previous execution error. Call `GrB_error()` to access any error
1594 messages generated by the implementation.

1595 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1596 GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to GrB_Type_new
1597 (needed for user-defined types).

1598 GrB_NULL_POINTER The v pointer is NULL.

1599 GrB_INVALID_VALUE nsize is zero or outside the range of the type GrB_Index.

1600 Description

1601 Creates a new vector \mathbf{v} of domain $\mathbf{D}(\mathbf{d})$, size nsize, and empty $\mathbf{L}(\mathbf{v})$. The method returns a handle
1602 to the new vector in v.

1603 It is not an error to call this method more than once on the same variable; however, the handle to
1604 the previously created object will be overwritten.

1605 4.2.4.2 Vector_dup: Construct a copy of a GraphBLAS vector

1606 Creates a new vector with the same domain, size, and contents as another vector.

1607 C Syntax

```
1608            GrB_Info GrB_Vector_dup(GrB_Vector        *w,  
1609                                    const GrB_Vector   u);
```

1610 Parameters

1611 w (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1612 vector.

1613 u (IN) The GraphBLAS vector to be duplicated.

1614 Return Values

1615 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1616 blocking mode, this indicates that the API checks for the input
1617 arguments passed successfully. Either way, output vector w is ready
1618 to be used in the next method of the sequence.

1619 GrB_PANIC Unknown internal error.

1620 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1621 GraphBLAS objects (input or output) is in an invalid state caused
1622 by a previous execution error. Call GrB_error() to access any error
1623 messages generated by the implementation.

1624 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1625 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, u , has not been initialized by a call to
1626 Vector_new or Vector_dup.

1627 GrB_NULL_POINTER The w pointer is NULL.

1628 Description

1629 Creates a new vector w of domain $D(u)$, size $size(u)$, and contents $L(u)$. The method returns a
1630 handle to the new vector in w .

1631 It is not an error to call this method more than once on the same variable; however, the handle to
1632 the previously created object will be overwritten.

1633 4.2.4.3 Vector_resize: Resize a vector

1634 Changes the size of an existing vector.

1635 C Syntax

```
1636      GrB_Info GrB_Vector_resize(GrB_Vector  w,
1637                                GrB_Index   nsize);
```

1638 Parameters

1639 w (INOUT) An existing Vector object that is being resized.

1640 $nsize$ (IN) The new size of the vector. It can be smaller or larger than the current size.

1641 Return Values

1642 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1643 blocking mode, this indicates that the API checks for the input
1644 arguments passed successfully. Either way, output vector w is ready
1645 to be used in the next method of the sequence.

1646 GrB_PANIC Unknown internal error.

1647 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1648 GraphBLAS objects (input or output) is in an invalid state caused
1649 by a previous execution error. Call GrB_error() to access any error
1650 messages generated by the implementation.

1651 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1652 GrB_NULL_POINTER The w pointer is NULL.

1653 GrB_INVALID_VALUE nsz is zero or outside the range of the type GrB_Index.

1654 Description

1655 Changes the size of w to nsz . The domain $\mathbf{D}(w)$ of vector w remains the same. The contents $\mathbf{L}(w)$
1656 are modified as described below.

1657 Let $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$ when the method is called. When the method returns, $w = \langle \mathbf{D}(w), nsz, \mathbf{L}'(w) \rangle$
1658 where $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < nsz)\}$. That is, all elements of w with index greater
1659 than or equal to the new vector size (nsz) are dropped.

1660 4.2.4.4 Vector_clear: Clear a vector

1661 Removes all the elements (tuples) from a vector.

1662 C Syntax

1663 GrB_Info GrB_Vector_clear(GrB_Vector v);

1664 Parameters

1665 v (INOUT) An existing GraphBLAS vector to clear.

1666 Return Values

1667 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1668 blocking mode, this indicates that the API checks for the input
1669 arguments passed successfully. Either way, output vector v is ready
1670 to be used in the next method of the sequence.

1671 GrB_PANIC Unknown internal error.

1672 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1673 GraphBLAS objects (input or output) is in an invalid state caused
1674 by a previous execution error. Call GrB_error() to access any error
1675 messages generated by the implementation.

1676 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1677 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v , has not been initialized by a call to
1678 Vector_new or Vector_dup.

1679 **Description**

1680 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,
1681 $L(v) = \emptyset$. The size of the vector does not change.

1682 **4.2.4.5 Vector_size: Size of a vector**

1683 Retrieve the size of a vector.

1684 **C Syntax**

```
1685         GrB_Info GrB_Vector_size(GrB_Index      *nsize,  
1686                                const GrB_Vector v);
```

1687 **Parameters**

1688 `nsize` (OUT) On successful return, is set to the size of the vector.

1689 `v` (IN) An existing GraphBLAS vector being queried.

1690 **Return Values**

1691 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
1692 cessfully and the value of `nsize` has been set.

1693 `GrB_PANIC` Unknown internal error.

1694 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1695 GraphBLAS objects (input or output) is in an invalid state caused
1696 by a previous execution error. Call `GrB_error()` to access any error
1697 messages generated by the implementation.

1698 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS vector, `v`, has not been initialized by a call to
1699 `Vector_new` or `Vector_dup`.

1700 `GrB_NULL_POINTER` `nsize` pointer is NULL.

1701 **Description**

1702 Return `size(v)` in `nsize`.

1703 **4.2.4.6 Vector_nvals: Number of stored elements in a vector**

1704 Retrieve the number of stored elements (tuples) in a vector.

1705 C Syntax

```
1706      GrB_Info GrB_Vector_nvals(GrB_Index      *nvals,  
1707                                const GrB_Vector v);
```

1708 Parameters

1709 **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)
1710 in the vector.

1711 **v** (IN) An existing GraphBLAS vector being queried.

1712 Return Values

1713 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1714 cessfully and the value of **nvals** has been set.

1715 **GrB_PANIC** Unknown internal error.

1716 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1717 GraphBLAS objects (input or output) is in an invalid state caused
1718 by a previous execution error. Call **GrB_error()** to access any error
1719 messages generated by the implementation.

1720 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

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

1723 **GrB_NULL_POINTER** The **nvals** pointer is **NULL**.

1724 Description

1725 Return **nvals(v)** in **nvals**. This is the number of stored elements in vector **v**, which is the size of
1726 **L(v)** (see Section 3.5.2).

1727 4.2.4.7 Vector_build: Store elements from tuples into a vector

1728 C Syntax

```
1729      GrB_Info GrB_Vector_build(GrB_Vector      w,  
1730                                const GrB_Index *indices,  
1731                                const <type>    *values,  
1732                                GrB_Index      n,  
1733                                const GrB_BinaryOp dup);
```

1734 Parameters

- 1735 **w** (INOUT) An existing Vector object to store the result.
- 1736 **indices** (IN) Pointer to an array of indices.
- 1737 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of
1738 vector **w**.
- 1739 **n** (IN) The number of entries contained in each array (the same for **indices** and **values**).
- 1740 **dup** (IN) An associative and commutative binary operator to apply when duplicate
1741 values for the same location are present in the input arrays. All three domains of
1742 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
1743 then duplicate locations will result in an error.

1744 Return Values

- 1745 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1746 blocking mode, this indicates that the API checks for the input
1747 arguments passed successfully. Either way, output vector **w** is
1748 ready to be used in the next method of the sequence.
- 1749 **GrB_PANIC** Unknown internal error.
- 1750 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
1751 opaque GraphBLAS objects (input or output) is in an invalid
1752 state caused by a previous execution error. Call **GrB_error()** to
1753 access any error messages generated by the implementation.
- 1754 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
- 1755 **GrB_UNINITIALIZED_OBJECT** Either **w** has not been initialized by a call to **GrB_Vector_new**
1756 or by **GrB_Vector_dup**, or **dup** has not been initialized by a call
1757 to **GrB_BinaryOp_new**.
- 1758 **GrB_NULL_POINTER** **indices** or **values** pointer is **NULL**.
- 1759 **GrB_INDEX_OUT_OF_BOUNDS** A value in **indices** is outside the allowed range for **w**.
- 1760 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
1761 not all the same, or the domains of **values** and **w** are incompatible
1762 with each other or D_{dup} .
- 1763 **GrB_OUTPUT_NOT_EMPTY** Output vector **w** already contains valid tuples (elements). In
1764 other words, **GrB_Vector_nvals(C)** returns a positive value.
- 1765 **GrB_INVALID_VALUE** **indices** contains a duplicate location and **dup** is **GrB_NULL**.

1766 Description

1767 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
1768 from \mathbf{w} in its domain; otherwise, $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$.

1769 Each tuple $\{\text{indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the output in the form of

$$1770 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1771 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,
1772 `dup` is used to reduce the values before assignment into $\tilde{\mathbf{w}}$ as follows:

$$1773 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1774 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{w}}$ is copied into \mathbf{w} via typecasting its
1775 values to $\mathbf{D}(\mathbf{w})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

1776 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1777 It is an error to call this function on an output object with existing elements. In other words,
1778 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1779 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`
1780 or `values`.

1781 4.2.4.8 Vector_setElement: Set a single element in a vector

1782 Set one element of a vector to a given value.

1783 C Syntax

```
1784 // scalar value
1785 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1786                               <type>         val,
1787                               GrB_Index       index);
1788
1789 // GraphBLAS scalar
1790 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1791                               const GrB_Scalar s,
1792                               GrB_Index       index);
```

1793 Parameters

1794 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1795 **val** or **s** (IN) Scalar assign. Its domain (type) must be compatible with the domain of **w**.
1796 **index** (IN) The location of the element to be assigned.

1797 **Return Values**

1798 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1799 blocking mode, this indicates that the compatibility tests on in-
1800 dex/dimensions and domains for the input arguments passed suc-
1801 cessfully. Either way, the output vector **w** is ready to be used in
1802 the next method of the sequence.

1803 **GrB_PANIC** Unknown internal error.

1804 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1805 GraphBLAS objects (input or output) is in an invalid state caused
1806 by a previous execution error. Call **GrB_error()** to access any error
1807 messages generated by the implementation.

1808 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1809 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been
1810 initialized by a call to a respective constructor.

1811 **GrB_INVALID_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1812 **GrB_DOMAIN_MISMATCH** The domains of the vector and the scalar are incompatible.

1813 **Description**

1814 First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)**
1815 must be compatible with **D(w)**. Two domains are compatible with each other if values from
1816 one domain can be cast to values in the other domain as per the rules of the C language. In
1817 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1818 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1819 **GrB_Vector_setElement** ends and the domain mismatch error listed above is returned.

1820 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1821 \qquad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1822 If this condition is violated, execution of **GrB_Vector_setElement** ends and the invalid index error
1823 listed above is returned.

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

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

1824 In the case of a transparent scalar or if $\mathbf{L}(\mathbf{s})$ is not empty, then a value will be stored at the
 1825 specified location in \mathbf{w} , overwriting any value that may have been stored there before. In the case
 1826 of a GraphBLAS scalar, if $\mathbf{L}(\mathbf{s})$ is empty, then any value stored at the specified location in \mathbf{w} will
 1827 be removed.

1828 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
 1829 of \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
 1830 return value GrB_SUCCESS and the new contents of vector \mathbf{w} is as defined above but may not be
 1831 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1832 4.2.4.9 Vector_removeElement: Remove an element from a vector

1833 Remove (annihilate) one stored element from a vector.

1834 C Syntax

```
1835         GrB_Info GrB_Vector_removeElement(GrB_Vector  w,
1836                                         GrB_Index   index);
```

1837 Parameters

1838 \mathbf{w} (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1839 \mathbf{index} (IN) The location of the element to be removed.

1840 Return Values

1841 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 1842 blocking mode, this indicates that the compatibility tests on in-
 1843 dex/dimensions and domains for the input arguments passed suc-
 1844 cessfully. Either way, the output vector \mathbf{w} is ready to be used in
 1845 the next method of the sequence.

1846 GrB_PANIC Unknown internal error.

1847 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 1848 GraphBLAS objects (input or output) is in an invalid state caused
 1849 by a previous execution error. Call GrB_error() to access any error
 1850 messages generated by the implementation.

1851 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1852 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, \mathbf{w} , has not been initialized by a call to
 1853 Vector_new or Vector_dup.

1854 GrB_INVALID_INDEX \mathbf{index} specifies a location that is outside the dimensions of \mathbf{w} .

1855 Description

1856 First, the `index` parameter is checked for a valid value where the following condition must hold:

$$1857 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1858 If this condition is violated, execution of `GrB_Vector_removeElement` ends and the invalid index
1859 error listed above is returned.

1860 We are now ready to carry out the removal of a value that may be stored at the location specified
1861 by `index`. If a value does not exist at the specified location in \mathbf{w} , no error is reported and the
1862 operation has no effect on the state of \mathbf{w} . In either case, the following will be true on return from
1863 the method: `index` \notin `ind(w)`.

1864 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1865 of \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1866 return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not be
1867 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1868 4.2.4.10 Vector_extractElement: Extract a single element from a vector.

1869 Extract one element of a vector into a scalar.

1870 C Syntax

```
1871 // scalar value
1872 GrB_Info GrB_Vector_extractElement(<type>          *val,
1873                                   const GrB_Vector u,
1874                                   GrB_Index         index);
1875
1876 // GraphBLAS scalar
1877 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1878                                   const GrB_Vector u,
1879                                   GrB_Index         index);
```

1880 Parameters

1881 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector
1882 `u`. On successful return, this scalar holds the result of the extract. Any previous
1883 value stored in `val` or `s` is overwritten.

1884 `u` (IN) The GraphBLAS vector from which an element is extracted.

1885 `index` (IN) The location in `u` to extract.

1886 Return Values

- 1887 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1888 cessfully. This indicates that the compatibility tests on dimensions
1889 and domains for the input arguments passed successfully, and the
1890 output scalar, **val** or **s**, has been computed and is ready to be used
1891 in the next method of the sequence.
- 1892 **GrB_NO_VALUE** When using the transparent scalar, **val**, this is returned when there
1893 is no stored value at specified location.
- 1894 **GrB_PANIC** Unknown internal error.
- 1895 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1896 GraphBLAS objects (input or output) is in an invalid state caused
1897 by a previous execution error. Call **GrB_error()** to access any error
1898 messages generated by the implementation.
- 1899 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
- 1900 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **u**, or scalar, **s**, has not been initialized by
1901 a call to a corresponding constructor.
- 1902 **GrB_NULL_POINTER** **val** pointer is NULL.
- 1903 **GrB_INVALID_INDEX** **index** specifies a location that is outside the dimensions of **w**.
- 1904 **GrB_DOMAIN_MISMATCH** The domains of the vector and scalar are incompatible.

1905 Description

1906 First, the scalar and input vector are tested for domain compatibility as follows: **D(val)** or **D(s)**
1907 must be compatible with **D(u)**. Two domains are compatible with each other if values from
1908 one domain can be cast to values in the other domain as per the rules of the C language. In
1909 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
1910 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
1911 **GrB_Vector_extractElement** ends and the domain mismatch error listed above is returned.

1912 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1913 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1914 If this condition is violated, execution of **GrB_Vector_extractElement** ends and the invalid index
1915 error listed above is returned.

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

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

1916 If $\text{index} \in \mathbf{ind}(u)$, then the corresponding value from u is copied into s or val with casting as
 1917 necessary. If $\text{index} \notin \mathbf{ind}(u)$, then one of the follow occurs depending on output scalar type:

- 1918 • The GraphBLAS scalar, s , is cleared and `GrB_SUCCESS` is returned.
- 1919 • The non-opaque scalar, val , is unchanged, and `GrB_NO_VALUE` is returned.

1920 When using the non-opaque scalar variant (val) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
 1921 mode, the new contents of val are as defined above if the method exits with return value `GrB_SUCCESS`
 1922 or `GrB_NO_VALUE`.

1923 When using the GraphBLAS scalar variant (s) with a `GrB_SUCCESS` return value, the method
 1924 exits and the new contents of s is as defined above and fully computed in `GrB_BLOCKING` mode.
 1925 In `GrB_NONBLOCKING` mode, the new contents of s is as defined above but may not be fully
 1926 computed; however, it can be used in the next GraphBLAS method call in a sequence.

1927 4.2.4.11 Vector_extractTuples: Extract tuples from a vector

1928 Extract the contents of a GraphBLAS vector into non-opaque data structures.

1929 C Syntax

```
1930      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,
1931                                     <type>          *values,
1932                                     GrB_Index        *n,
1933                                     const GrB_Vector  v);
1934
```

1935 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored
 1936 values' indices.

1937 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
 1938 the stored values whose type is compatible with $\mathbf{D}(v)$.

1939 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the
 1940 values and indices arrays can hold. Upon return, it will contain the number of
 1941 values written to the arrays.

1942 **v** (IN) An existing GraphBLAS vector.

1943 Return Values

1944 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
 1945 cessfully. This indicates that the compatibility tests on the input
 1946 argument passed successfully, and the output arrays, **indices** and
 1947 **values**, have been computed.

1948 GrB_PANIC Unknown internal error.

1949 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1950 GraphBLAS objects (input or output) is in an invalid state caused
1951 by a previous execution error. Call GrB_error() to access any error
1952 messages generated by the implementation.

1953 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1954 GrB_INSUFFICIENT_SPACE Not enough space in `indices` and `values` (as indicated by the `n` pa-
1955 rameter) to hold all of the tuples that will be extracted.

1956 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to
1957 Vector_new or Vector_dup.

1958 GrB_NULL_POINTER `indices`, `values`, or `n` pointer is NULL.

1959 GrB_DOMAIN_MISMATCH The domains of the `v` vector or `values` array are incompatible with
1960 one another.

1961 Description

1962 This method will extract all the tuples from the GraphBLAS vector `v`. The values associated
1963 with those tuples are placed in the `values` array and the indices are placed in the `indices` array.
1964 Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least
1965 GrB_Vector_nvals(`v`) elements before calling this function.

1966 Upon return of this function, `n` will be set to the number of values (and indices) copied. Also, the
1967 entries of `indices` are unique, but not necessarily sorted. Each tuple (i, v_i) in `v` is unzipped and
1968 copied into a distinct k th location in output vectors:

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

1969 where $0 \leq k < \text{GrB_Vector_nvals}(v)$. No gaps in output vectors are allowed; that is, if `indices`[k]
1970 and `values`[k] exist upon return, so does `indices`[j] and `values`[j] for all j such that $0 \leq j < k$.

1971 Note that if the value in `n` on input is less than the number of values contained in the vector `v`,
1972 then a GrB_INSUFFICIENT_SPACE error is returned because it is undefined which subset of values
1973 would be extracted otherwise.

1974 In both GrB_BLOCKING mode GrB_NONBLOCKING mode if the method exits with return value
1975 GrB_SUCCESS, the new contents of the arrays `indices` and `values` are as defined above.

1976 4.2.5 Matrix methods

1977 4.2.5.1 Matrix_new: Construct new matrix

1978 Creates a new matrix with specified domain and dimensions.

1979 C Syntax

```
1980         GrB_Info GrB_Matrix_new(GrB_Matrix *A,  
1981                                 GrB_Type      d,  
1982                                 GrB_Index     nrows,  
1983                                 GrB_Index     ncols);
```

1984 Parameters

1985 **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1986 matrix.

1987 **d** (IN) The type corresponding to the domain of the matrix being created. Can be
1988 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
1989 GraphBLAS type.

1990 **nrows** (IN) The number of rows of the matrix being created.

1991 **ncols** (IN) The number of columns of the matrix being created.

1992 Return Values

1993 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1994 blocking mode, this indicates that the API checks for the input ar-
1995 guments passed successfully. Either way, output matrix **A** is ready
1996 to be used in the next method of the sequence.

1997 **GrB_PANIC** Unknown internal error.

1998 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1999 GraphBLAS objects (input or output) is in an invalid state caused
2000 by a previous execution error. Call **GrB_error()** to access any error
2001 messages generated by the implementation.

2002 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2003 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
2004 (needed for user-defined types).

2005 **GrB_NULL_POINTER** The **A** pointer is **NULL**.

2006 **GrB_INVALID_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB_Index**.

2007 Description

2008 Creates a new matrix **A** of domain **D**(**d**), size **nrows** \times **ncols**, and empty **L**(**A**). The method returns
2009 a handle to the new matrix in **A**.

2010 It is not an error to call this method more than once on the same variable; however, the handle to
2011 the previously created object will be overwritten.

2012 4.2.5.2 Matrix_dup: Construct a copy of a GraphBLAS matrix

2013 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

2014 C Syntax

```
2015         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
2016                                const GrB_Matrix  A);
```

2017 Parameters

2018 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2019 matrix.

2020 A (IN) The GraphBLAS matrix to be duplicated.

2021 Return Values

2022 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2023 blocking mode, this indicates that the API checks for the input
2024 arguments passed successfully. Either way, output matrix C is ready
2025 to be used in the next method of the sequence.

2026 GrB_PANIC Unknown internal error.

2027 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2028 GraphBLAS objects (input or output) is in an invalid state caused
2029 by a previous execution error. Call GrB_error() to access any error
2030 messages generated by the implementation.

2031 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2032 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2033 any matrix constructor.

2034 GrB_NULL_POINTER The C pointer is NULL.

2035 Description

2036 Creates a new matrix C of domain D(A), size nrows(A) × ncols(A), and contents L(A). It returns
2037 a handle to it in C.

2038 It is not an error to call this method more than once on the same variable; however, the handle to
2039 the previously created object will be overwritten.

2040 4.2.5.3 Matrix_diag: Construct a diagonal GraphBLAS matrix

2041 Creates a new matrix with the same domain and contents as a GrB_Vector, and square dimensions
2042 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

2043 C Syntax

```
2044         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
2045                                 const GrB_Vector  v,  
2046                                 int64_t           k);
```

2047 Parameters

2048 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2049 matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

2050 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the
2051 matrix.

2052 k (IN) The diagonal to which the vector is assigned. k = 0 represents the main
2053 diagonal, k > 0 is above the main diagonal, and k < 0 is below.

2054 Return Values

2055 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2056 blocking mode, this indicates that the API checks for the input
2057 arguments passed successfully. Either way, output matrix C is ready
2058 to be used in the next method of the sequence.

2059 GrB_PANIC Unknown internal error.

2060 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2061 GraphBLAS objects (input or output) is in an invalid state caused
2062 by a previous execution error. Call GrB_error() to access any error
2063 messages generated by the implementation.

2064 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2065 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to
2066 Vector_new or Vector_dup.

2067 GrB_NULL_POINTER The C pointer is NULL.

2068 Description

2069 Creates a new matrix **C** of domain **D(v)**, size $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$, and contents

$$2070 \quad \mathbf{L}(\mathbf{C}) = \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or}$$

$$2071 \quad \mathbf{L}(\mathbf{C}) = \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0.$$

2072 It returns a handle to it in **C**. It is not an error to call this method more than once on the same
2073 variable; however, the handle to the previously created object will be overwritten.

2074 4.2.5.4 Matrix_resize: Resize a matrix

2075 Changes the dimensions of an existing matrix.

2076 C Syntax

```
2077 GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
2078                             GrB_Index nrows,  
2079                             GrB_Index ncols);
```

2080 Parameters

2081 **C** (INOUT) An existing Matrix object that is being resized.

2082 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the
2083 current number of rows.

2084 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than
2085 the current number of columns.

2086 Return Values

2087 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2088 blocking mode, this indicates that the API checks for the input
2089 arguments passed successfully. Either way, output matrix **C** is ready
2090 to be used in the next method of the sequence.

2091 **GrB_PANIC** Unknown internal error.

2092 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2093 GraphBLAS objects (input or output) is in an invalid state caused
2094 by a previous execution error. Call **GrB_error()** to access any error
2095 messages generated by the implementation.

2096 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2097 GrB_NULL_POINTER The C pointer is NULL.

2098 GrB_INVALID_VALUE nrows or ncols is zero or outside the range of the type GrB_Index.

2099 **Description**

2100 Changes the number of rows and columns of C to nrows and ncols, respectively. The domain $\mathbf{D}(\mathbf{C})$
2101 of matrix C remains the same. The contents $\mathbf{L}(\mathbf{C})$ are modified as described below.

2102 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
2103 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
2104 greater than or equal to ncols are dropped.
2105

2106 **4.2.5.5 Matrix_clear: Clear a matrix**

2107 Removes all elements (tuples) from a matrix.

2108 **C Syntax**

2109 GrB_Info GrB_Matrix_clear(GrB_Matrix A);

2110 **Parameters**

2111 A (IN) An existing GraphBLAS matrix to clear.

2112 **Return Values**

2113 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2114 blocking mode, this indicates that the API checks for the input ar-
2115 guments passed successfully. Either way, output matrix A is ready
2116 to be used in the next method of the sequence.

2117 GrB_PANIC Unknown internal error.

2118 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2119 GraphBLAS objects (input or output) is in an invalid state caused
2120 by a previous execution error. Call GrB_error() to access any error
2121 messages generated by the implementation.

2122 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2123 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2124 any matrix constructor.

2125 **Description**

2126 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,
2127 $\mathbf{L}(\mathbf{A}) = \emptyset$. The dimensions of the matrix do not change.

2128 **4.2.5.6 Matrix_nrows: Number of rows in a matrix**

2129 Retrieve the number of rows in a matrix.

2130 **C Syntax**

```
2131         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2132                                   const GrB_Matrix A);
```

2133 **Parameters**

2134 nrows (OUT) On successful return, contains the number of rows in the matrix.

2135 A (IN) An existing GraphBLAS matrix being queried.

2136 **Return Values**

2137 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2138 cessfully and the value of `nrows` has been set.

2139 GrB_PANIC Unknown internal error.

2140 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2141 GraphBLAS objects (input or output) is in an invalid state caused
2142 by a previous execution error. Call `GrB_error()` to access any error
2143 messages generated by the implementation.

2144 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
2145 any matrix constructor.

2146 GrB_NULL_POINTER `nrows` pointer is NULL.

2147 **Description**

2148 Return `nrows(A)` in `nrows` (the number of rows).

2149 **4.2.5.7 Matrix_ncols: Number of columns in a matrix**

2150 Retrieve the number of columns in a matrix.

2151 C Syntax

```
2152         GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2153                                   const GrB_Matrix A);
```

2154 Parameters

2155 ncols (OUT) On successful return, contains the number of columns in the matrix.

2156 A (IN) An existing GraphBLAS matrix being queried.

2157 Return Values

2158 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
2159 cessfully and the value of ncols has been set.

2160 GrB_PANIC Unknown internal error.

2161 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2162 GraphBLAS objects (input or output) is in an invalid state caused
2163 by a previous execution error. Call GrB_error() to access any error
2164 messages generated by the implementation.

2165 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
2166 any matrix constructor.

2167 GrB_NULL_POINTER ncols pointer is NULL.

2168 Description

2169 Return **ncols(A)** in ncols (the number of columns).

2170 4.2.5.8 Matrix_nvals: Number of stored elements in a matrix

2171 Retrieve the number of stored elements (tuples) in a matrix.

2172 C Syntax

```
2173         GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2174                                   const GrB_Matrix A);
```

2175 **Parameters**

2176 **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in
2177 the matrix.

2178 **A** (IN) An existing GraphBLAS matrix being queried.

2179 **Return Values**

2180 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2181 cessfully and the value of **nvals** has been set.

2182 **GrB_PANIC** Unknown internal error.

2183 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2184 GraphBLAS objects (input or output) is in an invalid state caused
2185 by a previous execution error. Call **GrB_error()** to access any error
2186 messages generated by the implementation.

2187 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

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

2190 **GrB_NULL_POINTER** The **nvals** pointer is **NULL**.

2191 **Description**

2192 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of
2193 **L(A)** (see Section 3.5.3).

2194 **4.2.5.9 Matrix_build: Store elements from tuples into a matrix**

2195 **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);
```

2196 **Parameters**

2197 **C** (INOUT) An existing Matrix object to store the result.

2198 **row_indices** (IN) Pointer to an array of row indices.

2199 **col_indices** (IN) Pointer to an array of column indices.

2200 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of
2201 matrix, **C**.

2202 **n** (IN) The number of entries contained in each array (the same for **row_indices**,
2203 **col_indices**, and **values**).

2204 **dup** (IN) An associative and commutative binary operator to apply when duplicate
2205 values for the same location are present in the input arrays. All three domains of
2206 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
2207 then duplicate locations will result in an error.

2208 Return Values

2209 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2210 blocking mode, this indicates that the API checks for the input
2211 arguments passed successfully. Either way, output matrix **C** is
2212 ready to be used in the next method of the sequence.

2213 **GrB_PANIC** Unknown internal error.

2214 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2215 opaque GraphBLAS objects (input or output) is in an invalid
2216 state caused by a previous execution error. Call **GrB_error()** to
2217 access any error messages generated by the implementation.

2218 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2219 **GrB_UNINITIALIZED_OBJECT** Either **C** has not been initialized by a call to any matrix construc-
2220 tor, or **dup** has not been initialized by a call to **GrB_BinaryOp_new**.

2221 **GrB_NULL_POINTER** **row_indices**, **col_indices** or **values** pointer is **NULL**.

2222 **GrB_INDEX_OUT_OF_BOUNDS** A value in **row_indices** or **col_indices** is outside the allowed range
2223 for **C**.

2224 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
2225 not all the same, or the domains of **values** and **C** are incompatible
2226 with each other or D_{dup} .

2227 **GrB_OUTPUT_NOT_EMPTY** Output matrix **C** already contains valid tuples (elements). In
2228 other words, **GrB_Matrix_nvals(C)** returns a positive value.

2229 **GrB_INVALID_VALUE** indices contains a duplicate location and **dup** is **GrB_NULL**.

2230 Description

2231 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
 2232 only differs from \mathbf{C} in its domain; otherwise, $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$.

2233 Each tuple $\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the
 2234 output in the form of

$$2235 \quad \tilde{\mathbf{C}}(\text{row_indices}[k], \text{col_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2236 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,
 2237 `dup` is used to reduce the values before assignment into $\tilde{\mathbf{C}}$ as follows:

$$2238 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row_indices}[k]=i \wedge \text{col_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2239 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{C}}$ is copied into \mathbf{C} via typecasting its
 2240 values to $\mathbf{D}(\mathbf{C})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

2241 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2242 It is an error to call this function on an output object with existing elements. In other words,
 2243 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2244 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,
 2245 `col_indices`, or `values`.

2246 4.2.5.10 Matrix_setElement: Set a single element in matrix

2247 Set one element of a matrix to a given value.

2248 C Syntax

```
2249 // scalar value
2250 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2251                                <type>         val,
2252                                GrB_Index        row_index,
2253                                GrB_Index        col_index);
2254
2255 // GraphBLAS scalar
2256 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2257                                const GrB_Scalar s,
2258                                GrB_Index        row_index,
2259                                GrB_Index        col_index);
```

2260 Parameters

2261 **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.
2262 **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of
2263 **C**.
2264 **row_index** (IN) Row index of element to be assigned
2265 **col_index** (IN) Column index of element to be assigned

2266 Return Values

2267 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2268 blocking mode, this indicates that the compatibility tests on in-
2269 dex/dimensions and domains for the input arguments passed suc-
2270 cessfully. Either way, the output matrix **C** is ready to be used in
2271 the next method of the sequence.
2272 **GrB_PANIC** Unknown internal error.
2273 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2274 GraphBLAS objects (input or output) is in an invalid state caused
2275 by a previous execution error. Call **GrB_error()** to access any error
2276 messages generated by the implementation.
2277 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
2278 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been
2279 initialized by a call to a respective constructor.
2280 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e., not less
2281 than **nrows(C)** or **ncols(C)**, respectively).
2282 **GrB_DOMAIN_MISMATCH** The domains of the matrix and the scalar are incompatible.

2283 Description

2284 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or
2285 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from
2286 one domain can be cast to values in the other domain as per the rules of the C language. In
2287 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
2288 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
2289 **GrB_Matrix_setElement** ends and the domain mismatch error listed above is returned.

2290 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2291 \quad & 0 \leq \text{row_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2292 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid
 2293 index error listed above is returned.

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

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

2294 In the case of a transparent scalar or if $\mathbf{L}(s)$ is not empty, then a value will be stored at the
 2295 specified location in C , overwriting any value that may have been stored there before. In the case
 2296 of a GraphBLAS scalar and if $\mathbf{L}(s)$ is empty, then any value stored at the specified location in C
 2297 will be removed.

2298 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
 2299 of C is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
 2300 return value `GrB_SUCCESS` and the new content of vector C is as defined above but may not be
 2301 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2302 **4.2.5.11 Matrix_removeElement: Remove an element from a matrix**

2303 Remove (annihilate) one stored element from a matrix.

2304 **C Syntax**

```
2305      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2306                                       GrB_Index   row_index,
2307                                       GrB_Index   col_index);
```

2308 **Parameters**

2309 C (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2310 row_index (IN) Row index of element to be removed

2311 col_index (IN) Column index of element to be removed

2312 **Return Values**

2313 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 2314 blocking mode, this indicates that the compatibility tests on in-
 2315 dex/dimensions and domains for the input arguments passed suc-
 2316 cessfully. Either way, the output matrix C is ready to be used in
 2317 the next method of the sequence.

2318 `GrB_PANIC` Unknown internal error.

2319 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 2320 GraphBLAS objects (input or output) is in an invalid state caused
 2321 by a previous execution error. Call GrB_error() to access any error
 2322 messages generated by the implementation.

2323 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2324 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to
 2325 any matrix constructor.

2326 GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e., not less
 2327 than nrows(C) or ncols(C), respectively).

2328 Description

2329 First, both index parameters are checked for valid values where following conditions must hold:

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

2331 If either of these conditions is violated, execution of GrB_Matrix_removeElement ends and the
 2332 invalid index error listed above is returned.

2333 We are now ready to carry out the removal of a value that may be stored at the location specified by
 2334 (row_index, col_index). If a value does not exist at the specified location in C, no error is reported
 2335 and the operation has no effect on the state of C. In either case, the following will be true on return
 2336 from this method: (row_index, col_index) \notin ind(C)

2337 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
 2338 of C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
 2339 return value GrB_SUCCESS and the new content of vector C is as defined above but may not be
 2340 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2341 4.2.5.12 Matrix_extractElement: Extract a single element from a matrix

2342 Extract one element of a matrix into a scalar.

2343 C Syntax

```

2344     // scalar value
2345     GrB_Info GrB_Matrix_extractElement(<type>          *val,
2346                                     const GrB_Matrix  A,
2347                                     GrB_Index          row_index,
2348                                     GrB_Index          col_index);
2349
2350     // GraphBLAS scalar
  
```

```

2351         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2352                                           const GrB_Matrix A,
2353                                           GrB_Index      row_index,
2354                                           GrB_Index      col_index);
2355

```

2356 Parameters

2357 **val or s (INOUT)** An existing scalar whose domain is compatible with the domain of matrix
2358 A. On successful return, this scalar holds the result of the extract. Any previous
2359 value stored in **val** or **s** is overwritten.

2360 **A (IN)** The GraphBLAS matrix from which an element is extracted.

2361 **row_index (IN)** The row index of location in **A** to extract.

2362 **col_index (IN)** The column index of location in **A** to extract.

2363 Return Values

2364 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2365 cessfully. This indicates that the compatibility tests on dimensions
2366 and domains for the input arguments passed successfully, and the
2367 output scalar, **val** or **s**, has been computed and is ready to be used
2368 in the next method of the sequence.

2369 **GrB_NO_VALUE** When using the transparent scalar, **val**, this is returned when there
2370 is no stored value at specified location.

2371 **GrB_PANIC** Unknown internal error.

2372 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2373 GraphBLAS objects (input or output) is in an invalid state caused
2374 by a previous execution error. Call **GrB_error()** to access any error
2375 messages generated by the implementation.

2376 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2377 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by
2378 a call to a corresponding constructor.

2379 **GrB_NULL_POINTER** **val** pointer is **NULL**.

2380 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e. less than
2381 zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-
2382 tively).

2383 **GrB_DOMAIN_MISMATCH** The domains of the matrix and scalar are incompatible.

2384 Description

2385 First, the scalar and input matrix are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ or $\mathbf{D}(\mathbf{s})$
 2386 must be compatible with $\mathbf{D}(\mathbf{A})$. Two domains are compatible with each other if values from
 2387 one domain can be cast to values in the other domain as per the rules of the C language. In
 2388 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-
 2389 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
 2390 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2391 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2392 \quad & 0 \leq \text{row_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2393 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index
 2394 error listed above is returned.

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

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

2395 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
 2396 with casting as necessary. If $(\text{row_index}, \text{col_index}) \notin \mathbf{ind}(\mathbf{A})$, then one of the follow occurs
 2397 depending on output scalar type:

- 2398 • The GraphBLAS scalar, \mathbf{s} , is cleared and `GrB_SUCCESS` is returned.
- 2399 • The non-opaque scalar, val , is unchanged, and `GrB_NO_VALUE` is returned.

2400 When using the non-opaque scalar variant (val) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
 2401 mode, the new contents of val are as defined above if the method exits with return value `GrB_SUCCESS`
 2402 or `GrB_NO_VALUE`.

2403 When using the GraphBLAS scalar variant (\mathbf{s}) with a `GrB_SUCCESS` return value, the method
 2404 exits and the new contents of \mathbf{s} is as defined above and fully computed in `GrB_BLOCKING` mode.
 2405 In `GrB_NONBLOCKING` mode, the new contents of \mathbf{s} is as defined above but may not be fully
 2406 computed; however, it can be used in the next GraphBLAS method call in a sequence.

2407 4.2.5.13 Matrix_extractTuples: Extract tuples from a matrix

2408 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

2409 C Syntax

```
2410      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2411                                       GrB_Index      *col_indices,
```

```

2412                                     <type>          *values,
2413                                     GrB_Index         *n,
2414                                     const GrB_Matrix   A);

```

2415 Parameters

2416 **row_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the
2417 row indices.

2418 **col_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the
2419 column indices.

2420 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
2421 the stored values whose type is compatible with $\mathbf{D}(\mathbf{A})$.

2422 **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,
2423 **row_indices**, and **col_indices** arrays can hold. Upon return, it will contain the
2424 number of values written to the arrays.

2425 **A** (IN) An existing GraphBLAS matrix.

2426 Return Values

2427 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2428 cessfully. This indicates that the compatibility tests on the input
2429 argument passed successfully, and the output arrays, **indices** and
2430 **values**, have been computed.

2431 **GrB_PANIC** Unknown internal error.

2432 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2433 GraphBLAS objects (input or output) is in an invalid state caused
2434 by a previous execution error. Call **GrB_error()** to access any error
2435 messages generated by the implementation.

2436 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2437 **GrB_INSUFFICIENT_SPACE** Not enough space in **row_indices**, **col_indices**, and **values** (as indi-
2438 cated by the **n** parameter) to hold all of the tuples that will be
2439 extracted.

2440 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2441 any matrix constructor.

2442 **GrB_NULL_POINTER** **row_indices**, **col_indices**, **values** or **n** pointer is NULL.

2443 **GrB_DOMAIN_MISMATCH** The domains of the **A** matrix and **values** array are incompatible
2444 with one another.

2445 Description

2446 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with
2447 those tuples are placed in the **values** array, the column indices are placed in the **col_indices** array,
2448 and the row indices are placed in the **row_indices** array. These output arrays are pre-allocated by
2449 the user before calling this function such that each output array has enough space to hold at least
2450 **GrB_Matrix_nvals(A)** elements.

2451 Upon return of this function, a pair of $\{\text{row_indices}[k], \text{col_indices}[k]\}$ are unique for every valid
2452 k , but they are not required to be sorted in any particular order. Each tuple (i, j, A_{ij}) in **A** is
2453 unzipped and copied into a distinct k th location in output vectors:

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

2454 where $0 \leq k < \text{GrB_Matrix_nvals}(v)$. No gaps in output vectors are allowed; that is, if **row_indices**[k],
2455 **col_indices**[k] and **values**[k] exist upon return, so does **row_indices**[j], **col_indices**[j] and **values**[j] for
2456 all j such that $0 \leq j < k$.

2457 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,
2458 then a **GrB_INSUFFICIENT_SPACE** error is returned since it is undefined which subset of values
2459 would be extracted.

2460 In both **GrB_BLOCKING** mode **GrB_NONBLOCKING** mode if the method exits with return value
2461 **GrB_SUCCESS**, the new contents of the arrays **row_indices**, **col_indices** and **values** are as defined
2462 above.

2463 4.2.5.14 Matrix_exportHint: Provide a hint as to which storage format might be most 2464 efficient for exporting a matrix

2465 C Syntax

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

2466 Parameters

2467 hint (OUT) Pointer to a value of type **GrB_Format**.

2468 A (IN) A GraphBLAS matrix object.

2469 Return Values

2470 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2471 cessfully and the value of **hint** has been set.

2472 **GrB_PANIC** Unknown internal error.

2473 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 2474 opaque GraphBLAS objects (input or output) is in an invalid
 2475 state caused by a previous execution error. Call GrB_error() to
 2476 access any error messages generated by the implementation.

2477 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2478 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
 2479 any matrix constructor.

2480 GrB_NULL_POINTER hint is NULL.

2481 GrB_NO_VALUE If the implementation does not have a preferred format, it may
 2482 return the value GrB_NO_VALUE.

2483 Description

2484 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for
 2485 exporting the matrix A. GraphBLAS implementations might return the current storage format of
 2486 the matrix, or the format to which it could most efficiently be exported. However, implementations
 2487 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is
 2488 free to refuse to provide a format hint, returning GrB_NO_VALUE.

2489 4.2.5.15 Matrix_exportSize: Return the array sizes necessary to export a GraphBLAS 2490 matrix object

2491 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);

```

2492 Parameters

2493 n_indptr (OUT) Pointer to a value of type GrB_Index.

2494 n_indices (OUT) Pointer to a value of type GrB_Index.

2495 n_values (OUT) Pointer to a value of type GrB_Index.

2496 format (IN) a value indicating the format in which the matrix will be exported, as defined
 2497 in Section 3.5.3.1.

2498 A (IN) A GraphBLAS matrix object.

2499 Return Values

2500 GrB_SUCCESS In blocking mode or non-blocking mode, the operation com-
2501 pleted successfully. This indicates that the API checks for the
2502 input arguments passed successfully, and the number of elements
2503 necessary for the export buffers have been written to `n_indptr`,
2504 `n_indices`, and `n_values`, respectively.

2505 GrB_PANIC Unknown internal error.

2506 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
2507 opaque GraphBLAS objects (input or output) is in an invalid
2508 state caused by a previous execution error. Call `GrB_error()` to
2509 access any error messages generated by the implementation.

2510 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2511 GrB_UNINITIALIZED_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to
2512 any matrix constructor.

2513 GrB_NULL_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

2514 Description

2515 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to
2516 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and
2517 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be
2518 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user
2519 as one of the values defined in Section 3.5.3.1.

2520 4.2.5.16 Matrix_export: Export a GraphBLAS matrix to a pre-defined format

2521 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);
```

2522 Parameters

- 2523 **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-
2524 dices, depending on the value of **format**. It must be large enough to hold at
2525 least **n_indptr** elements of type **GrB_Index**, where **n_indices** was returned from
2526 **GrB_Matrix_exportSize()** method.
- 2527 **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements
2528 in **values**, depending on the value of **format**. It must be large enough to hold at
2529 least **n_indices** elements of type **GrB_Index**, where **n_indices** was returned from
2530 **GrB_Matrix_exportSize()** method.
- 2531 **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-
2532 ment must match the type of the values stored in **A**. It must be large enough
2533 to hold at least **n_values** elements of that type, where **n_values** was returned from
2534 **GrB_Matrix_exportSize**.
- 2535 **n_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**
2536 array can hold. Upon return, it will contain the number of elements written to the
2537 array.
- 2538 **n_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**
2539 array can hold. Upon return, it will contain the number of elements written to the
2540 array.
- 2541 **n_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**
2542 array can hold. Upon return, it will contain the number of elements written to the
2543 array.
- 2544 **format** (IN) a value indicating the format in which the matrix will be exported, as defined
2545 in Section 3.5.3.1.
- 2546 **A** (IN) A GraphBLAS matrix object.

2547 Return Values

- 2548 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2549 cessfully. This indicates that the compatibility tests on the input
2550 argument passed successfully, and the output arrays, **indptr**, **in-**
2551 **dices** and **values**, have been computed.
- 2552 **GrB_PANIC** Unknown internal error.
- 2553 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2554 opaque GraphBLAS objects (input or output) is in an invalid
2555 state caused by a previous execution error. Call **GrB_error()** to
2556 access any error messages generated by the implementation.
- 2557 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2558 GrB_INSUFFICIENT_SPACE Not enough space in `indptr`, `indices`, and/or `values` (as indicated
2559 by the corresponding `n_*` parameter) to hold all of the corre-
2560 sponding elements that will be extracted.

2561 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
2562 any matrix constructor.

2563 GrB_NULL_POINTER `indptr`, `indices`, `values` `n_indptr`, `n_indices`, `n_values` pointer is
2564 NULL.

2565 GrB_DOMAIN_MISMATCH The domain of `A` does not match with the type of `values`.

2566 Description

2567 Given a matrix `A`, this method exports the contents of the matrix into one of the pre-defined
2568 `GrB_Format` formats from Section 3.5.3.1. The user-allocated arrays pointed to by `indptr`, `indices`,
2569 and `values` must be at least large enough to hold the corresponding number of elements returned
2570 by calling `GrB_Matrix_exportSize`. The value of `format` can be chosen arbitrarily, but a call to
2571 `GrB_Matrix_exportHint` may suggest a format that results in the most efficient export. Details
2572 of the contents of `indptr`, `indices`, and `values` corresponding to each supported format is given in
2573 Appendix B.

2574 4.2.5.17 Matrix_import: Import a matrix into a GraphBLAS object

2575 C Syntax

```

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

```

2576 Parameters

2577 `A` (INOUT) On a successful return, contains a handle to the newly created Graph-
2578 BLAS matrix.

2579 `d` (IN) The type corresponding to the domain of the matrix being created. Can be
2580 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined
2581 GraphBLAS type.

2582 **nrows** (IN) Integer value holding the number of rows in the matrix.

2583 **ncols** (IN) Integer value holding the number of columns in the matrix.

2584 **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the
2585 value of **format**.

2586 **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending
2587 on the value of **format**.

2588 **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2589 **n_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2590 **n_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2591 **n_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2592 **format** (IN) a value indicating the format of the matrix being imported, as defined in
2593 Section 3.5.3.1.

2594 **Return Values**

2595 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2596 blocking mode, this indicates that the API checks for the input
2597 arguments passed successfully and the input arrays have been
2598 consumed. Either way, output matrix **A** is ready to be used in
2599 the next method of the sequence.

2600 **GrB_PANIC** Unknown internal error.

2601 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2602 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
2603 (needed for user-defined types).

2604 **GrB_NULL_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2605 **GrB_INDEX_OUT_OF_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices
2606 in **A** and or the size of **values**, **n_values**, depending on the value
2607 of **format**.

2608 **GrB_INVALID_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB_Index**.

2609 **GrB_DOMAIN_MISMATCH** The domain given in parameter **d** does not match the element
2610 type of **values**.

2611 Description

2612 Creates a new matrix **A** of domain **D**(d) and dimension **nrows** \times **ncols**. The new GraphBLAS
2613 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.
2614 The method returns a handle to the new matrix in **A**. The structure of the data being imported is
2615 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of
2616 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2617 It is not an error to call this method more than once on the same output matrix; however, the
2618 handle to the previously created object will be overwritten.

2619 4.2.5.18 Matrix_serializeSize: Compute the serialize buffer size

2620 Compute the buffer size (in bytes) necessary to serialize a GrB_Matrix using GrB_Matrix_serialize.

2621 C Syntax

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

2622 Parameters

2623 size (OUT) Pointer to GrB_Index value where size in bytes of serialized object will be
2624 written.

2625 A (IN) A GraphBLAS matrix object.

2626 Return Values

2627 GrB_SUCCESS The operation completed successfully and the value pointed to
2628 by *size has been computed and is ready to use.

2629 GrB_PANIC Unknown internal error.

2630 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2631 GrB_NULL_POINTER size is NULL.

2632 Description

2633 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object A.
2634 Users may then allocate a buffer of size bytes to pass as a parameter to GrB_Matrix_serialize.

2635 **4.2.5.19 Matrix_serialize: Serialize a GraphBLAS matrix.**

2636 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2637 **C Syntax**

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

2638 **Parameters**

2639 **serialized_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be
2640 written.

2641 **serialized_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized_data**.
2642 On output, the number of bytes written to **serialized_data**.

2643 **A** (IN) A GraphBLAS matrix object.

2644 **Return Values**

2645 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2646 cessfully. This indicates that the compatibility tests on the in-
2647 put argument passed successfully, and the output buffer **serial-
2648 ized_data** and **serialized_size**, have been computed and are ready
2649 to use.

2650 **GrB_PANIC** Unknown internal error.

2651 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2652 opaque GraphBLAS objects (input or output) is in an invalid
2653 state caused by a previous execution error. Call **GrB_error()** to
2654 access any error messages generated by the implementation.

2655 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2656 **GrB_NULL_POINTER** **serialized_data** or **serialize_size** is NULL.

2657 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2658 any matrix constructor.

2659 **GrB_INSUFFICIENT_SPACE** The size of the buffer **serialized_data** (provided as an input **seri-
2660 alized_size**) was not large enough.

2661 Description

2662 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,
2663 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must
2664 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the
2665 serialized matrix written to `serialized_data` is provided upon completion as an output written to
2666 `serialized_size`.

2667 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created
2668 with one library implementation is not necessarily valid for deserialization with another implemen-
2669 tation.

2670 4.2.5.20 Matrix_deserialize: Deserialize a GraphBLAS matrix.

2671 Construct a new GraphBLAS matrix from a serialized object.

2672 C Syntax

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

2673 Parameters

2674 A (INOUT) On a successful return, contains a handle to the newly created Graph-
2675 BLAS matrix.

2676 d (IN) the type of the matrix that was serialized in `serialized_data`.

2677 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2678 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

2679 Return Values

2680 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2681 blocking mode, this indicates that the API checks for the input
2682 arguments passed successfully. Either way, output matrix A is
2683 ready to be used in the next method of the sequence.

2684 GrB_PANIC Unknown internal error.

2685 GrB_INVALID_OBJECT This is returned if `serialized_data` is invalid or corrupted.

2686 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2687 GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to GrB_Type_new
2688 (needed for user-defined types).

2689 GrB_NULL_POINTER serialized_data or A is NULL.

2690 GrB_DOMAIN_MISMATCH The type given in d does not match the type of the matrix
2691 serialized in serialized_data.

2692 Description

2693 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object
2694 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The
2695 domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized
2696 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2697 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix
2698 serialized in one library implementation can be deserialized by another.

2699 It is not an error to call this method more than once on the same output matrix; however, the
2700 handle to the previously created object will be overwritten.

2701 4.2.6 Descriptor methods

2702 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-
2703 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

2704 4.2.6.1 Descriptor_new: Create new descriptor

2705 Creates a new (empty or default) descriptor.

2706 C Syntax

2707 GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);

2708 Parameters

2709 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2710 descriptor.

2711 Return Value

2712 GrB_SUCCESS The method completed successfully.

2713 GrB_PANIC unknown internal error.

2714 GrB_OUT_OF_MEMORY not enough memory available for operation.

2715 GrB_NULL_POINTER desc pointer is NULL.

2716 **Description**

2717 Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can
2718 be populated by calls to Descriptor_set.

2719 It is not an error to call this method more than once on the same variable; however, the handle to
2720 the previously created object will be overwritten.

2721 **4.2.6.2 Descriptor_set: Set content of descriptor**

2722 Sets the content for a field for an existing descriptor.

2723 **C Syntax**

```
2724        GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2725                                    GrB_Desc_Field        field,  
2726                                    GrB_Desc_Value        val);
```

2727 **Parameters**

2728 desc (IN) An existing GraphBLAS descriptor to be modified.

2729 field (IN) The field being set.

2730 val (IN) New value for the field being set.

2731 **Return Values**

2732 GrB_SUCCESS operation completed successfully.

2733 GrB_PANIC unknown internal error.

2734 GrB_OUT_OF_MEMORY not enough memory available for operation.

2735 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to new.

2736 GrB_INVALID_VALUE invalid value set on the field, or invalid field.

2737 Description

2738 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor
2739 to set the value associated with that field. Valid values for the `field` parameter include the following:

2740 `GrB_OUTP` refers to the output parameter (result) of the operation.

2741 `GrB_MASK` refers to the mask parameter of the operation.

2742 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2743 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2744 Valid values for the `val` parameter are:

2745 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask
2746 (`GrB_MASK`) parameter.

2747 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-
2748 eter. When combined with `GrB_STRUCTURE`, the complement of the
2749 structure of the mask is used without evaluating the values stored.

2750 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input
2751 matrix parameters only).

2752 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear
2753 the matrix first (or clear the non-masked entries). The default behavior
2754 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`
2755 parameter only.

2756 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,
2757 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).
2758 A value for a given field may be set multiple times but will have no additional effect. Fields that
2759 have no values set result in their default behavior, as defined in Section 3.7.

2760 4.2.7 free: Destroy an object and release its resources

2761 Destroys a previously created GraphBLAS object and releases any resources associated with the
2762 object.

2763 C Syntax

2764 `GrB_Info GrB_free(<GrB_Object> *obj);`

2765 Parameters

2766 **obj** (INOUT) An existing GraphBLAS object to be destroyed. The object must have
2767 been created by an explicit call to a GraphBLAS constructor. It can be any of the
2768 opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,
2769 binary op, unary op, or type. On successful completion of **GrB_free**, **obj** behaves
2770 as an uninitialized object.

2771 Return Values

2772 **GrB_SUCCESS** operation completed successfully

2773 **GrB_PANIC** unknown internal error. If this return value is encountered when
2774 in nonblocking mode, the error responsible for the panic condition
2775 could be from any method involved in the computation of the input
2776 object. The **GrB_error()** method should be called for additional
2777 information.

2778 Description

2779 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime
2780 system. A call to **GrB_free** frees those resources so they are available for use by other GraphBLAS
2781 objects.

2782 The parameter passed into **GrB_free** is a handle referencing a GraphBLAS opaque object of a data
2783 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-
2784 structor. The behavior of a program that calls **GrB_free** on a pre-defined object is implementation
2785 defined.

2786 After the **GrB_free** method returns, the object referenced by the input handle is destroyed and the
2787 handle has the value **GrB_INVALID_HANDLE**. The handle can be used in subsequent GraphBLAS
2788 methods but only after the handle has been reinitialized with a call the the appropriate **_new** or
2789 **_dup** method.

2790 Note that unlike other GraphBLAS methods, calling **GrB_free** with an object with an invalid handle
2791 is legal. The system may attempt to free resources that might be associated with that object, if
2792 possible, and return normally.

2793 When using **GrB_free** it is possible to create a dangling reference to an object. This would occur
2794 when a handle is assigned to a second variable of the same opaque type. This creates two handles
2795 that reference the same object. If **GrB_free** is called with one of the variables, the object is destroyed
2796 and the handle associated with the other variable no longer references a valid object. This is not an
2797 error condition that the implementation of the GraphBLAS API can be expected to catch, hence
2798 programmers must take care to prevent this situation from occurring.

2799 **4.2.8 wait: Return once an object is either *complete* or *materialized***

2800 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2801 **C Syntax**

2802 `GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);`

2803 **Parameters**

2804 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an
2805 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2806 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2807 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another
2808 thread (completion) or all computing to produce `obj` by all GraphBLAS operations
2809 in its sequence have finished (materialization).

2810 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the
2811 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or
2812 `GrB_MATERIALIZE`.

2813 **Return values**

2814 `GrB_SUCCESS` operation completed successfully.

2815 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-
2816 pletion of pending operations.

2817 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion
2818 of pending operations.

2819 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,
2820 or other constructor, method.

2821 `GrB_PANIC` unknown internal error.

2822 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`
2823 `GrB_MATERIALIZE`.

2824 **Description**

2825 On successful return from `GrB_wait()`, the input object, `obj` is in one of two states depending on
2826 the mode of `GrB_wait`:

- 2827 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program
2828 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another
2829 thread. This result occurs when the mode parameter is set to `GrB_COMPLETE`.
- 2830 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on
2831 behalf of `obj` and error information is available. This result occurs when the mode parameter
2832 is set to `GrB_MATERIALIZE`.

2833 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,
2834 `GrB_wait(obj,mode)` has no effect when called in blocking mode.

2835 In non-blocking mode, the status of any pending method calls, other than those associated with pro-
2836 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to `GrB_wait(obj,mode)`.
2837 Methods in the sequence for `obj`, however, most likely would be impacted by a call to `GrB_wait(obj,mode)`;
2838 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be
2839 finished prior to the return from `GrB_wait(obj,mode)`.

2840 4.2.9 error: Retrieve an error string

2841 Retrieve an error-message about any errors encountered during the processing associated with an
2842 object.

2843 C Syntax

```
2844      GrB_Info GrB_error(const char      **error,
2845                        const GrB_Object  obj);
```

2846 Parameters

2847 `error` (OUT) A pointer to a null-terminated string. The contents of the string are im-
2848 plementation defined.

2849 `obj` (IN) An existing GraphBLAS object. The object must have been created by an
2850 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2851 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2852 or type.

2853 Return value

2854 `GrB_SUCCESS` operation completed successfully.

2855 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,
2856 or other constructor, method.

2857 `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	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

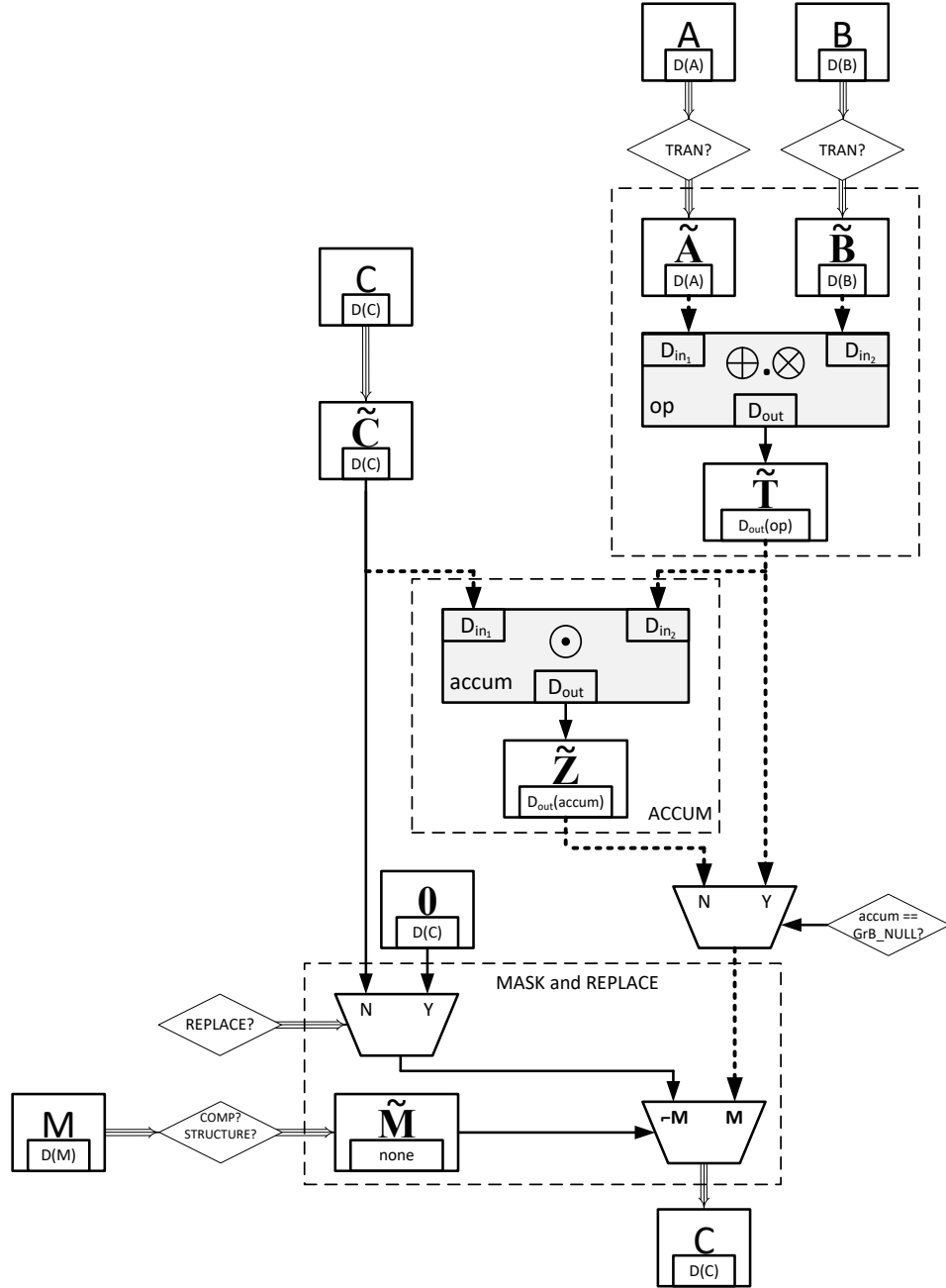


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows (\Rightarrow) denote where “as if copy” takes place (including both collections and descriptor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

2927 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-
 2928 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at
 2929 the point of their declaration, as shown in the following examples:

```
2930         GrB_Type          type = GrB_INVALID_HANDLE;
2931         GrB_Semiring       semiring = GrB_INVALID_HANDLE;
2932         GrB_Matrix         matrix = GrB_INVALID_HANDLE;
```

2933 Compliance

2934 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.
 2935 That is, for each operation we give a recipe for producing its outcome. Any implementation that
 2936 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error
 2937 model (Section 2.6) is a conforming implementation.

2938 4.3.1 mxm: Matrix-matrix multiply

2939 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

2940 C Syntax

```
2941         GrB_Info GrB_mxm(GrB_Matrix          C,
2942                         const GrB_Matrix     Mask,
2943                         const GrB_BinaryOp    accum,
2944                         const GrB_Semiring    op,
2945                         const GrB_Matrix      A,
2946                         const GrB_Matrix      B,
2947                         const GrB_Descriptor   desc);
```

2948 Parameters

2949 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 2950 that may be accumulated with the result of the matrix product. On output, the
 2951 matrix holds the results of the operation.

2952 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 2953 stored into the output matrix C. The mask dimensions must match those of the
 2954 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
 2955 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
 2956 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
 2957 dimensions of C), `GrB_NULL` should be specified.

2958 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
 2959 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 2960 specified.

2961 **op** (IN) The semiring used in the matrix-matrix multiply.

2962 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 2963 multiplication.

2964 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 2965 multiplication.

2966 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 2967 should be specified. Non-default field/value pairs are listed as follows:
 2968

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.

2970 Return Values

2971 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 2972 blocking mode, this indicates that the compatibility tests on di-
 2973 mensions and domains for the input arguments passed successfully.
 2974 Either way, output matrix **C** is ready to be used in the next method
 2975 of the sequence.

2976 **GrB_PANIC** Unknown internal error.

2977 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 2978 GraphBLAS objects (input or output) is in an invalid state caused
 2979 by a previous execution error. Call **GrB_error()** to access any error
 2980 messages generated by the implementation.

2981 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

2982 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 2983 a call to **new** (or **Matrix_dup** for matrix parameters).

2984 **GrB_DIMENSION_MISMATCH** Mask and/or matrix dimensions are incompatible.

2985 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 2986 corresponding domains of the semiring or accumulation operator,
 2987 or the mask's domain is not compatible with `bool` (in the case where
 2988 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

2989 Description

2990 GrB_mxm computes the matrix product $C = A \oplus . \otimes B$ or, if an optional binary accumulation operator
 2991 (\odot) is provided, $C = C \odot (A \oplus . \otimes B)$ (where matrices A and B can be optionally transposed).
 2992 Logically, this operation occurs in three steps:

2993 **Setup** The internal matrices and mask used in the computation are formed and their domains
 2994 and dimensions are tested for compatibility.

2995 **Compute** The indicated computations are carried out.

2996 **Output** The result is written into the output matrix, possibly under control of a mask.

2997 Up to four argument matrices are used in the GrB_mxm operation:

- 2998 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 2999 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)
- 3000 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3001 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3002 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for
 3003 domain compatibility as follows:

- 3004 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3005 must be from one of the pre-defined types of Table 3.2.
- 3006 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 3007 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 3008 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 3009 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3010 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 3011 of the accumulation operator.

3012 Two domains are compatible with each other if values from one domain can be cast to values in
 3013 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3014 all compatible with each other. A domain from a user-defined type is only compatible with itself.

3015 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch
 3016 error listed above is returned.

3017 From the argument matrices, the internal matrices and mask used in the computation are formed
 3018 (\leftarrow denotes copy):

- 3019 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3020 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3021 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3022 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3023 (b) If `Mask \neq GrB_NULL`,
 - 3024 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3025 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3026 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3027 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 3028 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3029 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3030 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3031 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3032 must hold:

- 3033 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 3034 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 3035 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 3036 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.
- 3037 5. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.

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

3040 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3041 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3042 We are now ready to carry out the matrix multiplication and any additional associated operations.
 3043 We describe this in terms of two intermediate matrices:

- 3044 • $\tilde{\mathbf{T}}$: The matrix holding the product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3045 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3046 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

$$3048 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3049 where \oplus and \otimes are the additive and multiplicative operators of semiring \mathbf{op} , respectively.

3050 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3051 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3052 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3053 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3054 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
3055 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 3056 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3057 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3058 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3059 \quad & \\ 3060 \end{aligned}$$

3061 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3062 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
3063 using what is called a *standard matrix mask and replace*. This is carried out under control of the
3064 mask which acts as a “write mask”.

- 3065 • If $\mathbf{desc}[\mathbf{GrB_OUTP}].\mathbf{GrB_REPLACE}$ is set, then any values in \mathbf{C} on input to this operation are
3066 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3067 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3068 • If $\mathbf{desc}[\mathbf{GrB_OUTP}].\mathbf{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
3069 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
3070 mask are unchanged:

$$3071 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3072 In $\mathbf{GrB_BLOCKING}$ mode, the method exits with return value $\mathbf{GrB_SUCCESS}$ and the new content
3073 of matrix \mathbf{C} is as defined above and fully computed. In $\mathbf{GrB_NONBLOCKING}$ mode, the method
3074 exits with return value $\mathbf{GrB_SUCCESS}$ and the new content of matrix \mathbf{C} is as defined above but
3075 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3076 sequence.

3077 4.3.2 vxm: Vector-matrix multiply

3078 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

3079 C Syntax

```
3080         GrB_Info GrB_vxm(GrB_Vector          w,  
3081                           const GrB_Vector    mask,  
3082                           const GrB_BinaryOp    accum,  
3083                           const GrB_Semiring    op,  
3084                           const GrB_Vector    u,  
3085                           const GrB_Matrix     A,  
3086                           const GrB_Descriptor  desc);
```

3087 Parameters

3088 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3089 that may be accumulated with the result of the vector-matrix product. On output,
3090 this vector holds the results of the operation.

3091 **mask** (IN) An optional “write” mask that controls which results from this operation are
3092 stored into the output vector **w**. The mask dimensions must match those of the
3093 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3094 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3095 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3096 dimensions of **w**), **GrB_NULL** should be specified.

3097 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3098 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
3099 specified.

3100 **op** (IN) Semiring used in the vector-matrix multiply.

3101 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3102 multiplication.

3103 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
3104 multiplication.

3105 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
3106 should be specified. Non-default field/value pairs are listed as follows:

3107

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.

3108

3109 Return Values

3110 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3111 blocking mode, this indicates that the compatibility tests on di-
3112 mensions and domains for the input arguments passed successfully.
3113 Either way, output vector w is ready to be used in the next method
3114 of the sequence.

3115 GrB_PANIC Unknown internal error.

3116 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3117 GraphBLAS objects (input or output) is in an invalid state caused
3118 by a previous execution error. Call GrB_error() to access any error
3119 messages generated by the implementation.

3120 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3121 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3122 a call to new (or dup for matrix or vector parameters).

3123 GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3124 GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with
3125 the corresponding domains of the semiring or accumulation opera-
3126 tor, or the mask's domain is not compatible with bool (in the case
3127 where desc[GrB_MASK].GrB_STRUCTURE is not set).

3128 Description

3129 GrB_vxm computes the vector-matrix product $w^T = u^T \oplus . \otimes A$, or, if an optional binary accu-
3130 mulation operator (\odot) is provided, $w^T = w^T \odot (u^T \oplus . \otimes A)$ (where matrix A can be optionally
3131 transposed). Logically, this operation occurs in three steps:

3132 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
3133 domains/dimensions are tested for compatibility.

3134 **Compute** The indicated computations are carried out.

3135 **Output** The result is written into the output vector, possibly under control of a mask.

3136 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3137 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3138 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3139 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3140 4. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3141 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
 3142 tested for domain compatibility as follows:

- 3143 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 3144 must be from one of the pre-defined types of Table 3.2.
- 3145 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 3146 3. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 3147 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.
- 3148 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3149 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$
 3150 of the accumulation operator.

3151 Two domains are compatible with each other if values from one domain can be cast to values in
 3152 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are
 3153 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 3154 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch
 3155 error listed above is returned.

3156 From the argument vectors and matrices, the internal matrices and mask used in the computation
 3157 are formed (\leftarrow denotes copy):

- 3158 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3159 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3160 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3161 (b) If `mask` \neq `GrB_NULL`,
 - 3162 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3163 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$.
 - 3164 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3165 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3166 4. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

3167 The internal matrices and masks are checked for shape compatibility. The following conditions
3168 must hold:

3169 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.

3170 2. $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$.

3171 3. $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$.

3172 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch
3173 error listed above is returned.

3174 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3175 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3176 We are now ready to carry out the vector-matrix multiplication and any additional associated
3177 operations. We describe this in terms of two intermediate vectors:

- 3178 • $\tilde{\mathbf{t}}$: The vector holding the product of vector $\tilde{\mathbf{u}}^T$ and matrix $\tilde{\mathbf{A}}$.
- 3179 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3180 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.
3181 The value of each of its elements is computed by

$$3182 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3183 where \oplus and \otimes are the additive and multiplicative operators of semiring `op`, respectively.

3184 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3185 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3186 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3187 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3188 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3189 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3190 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3191 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3192 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3193 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3194 \end{aligned}$$

3195 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3196 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3197 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3198 mask which acts as a “write mask”.

- 3199 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 3200 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3201 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3202 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3203 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3204 mask are unchanged:

$$3205 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3206 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 3207 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 3208 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 3209 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3210 sequence.

3211 4.3.3 mxv: Matrix-vector multiply

3212 Multiplies a matrix by a vector on a semiring. The result is a vector.

3213 C Syntax

```
3214      GrB_Info GrB_mxv(GrB_Vector      w,
3215                      const GrB_Vector mask,
3216                      const GrB_BinaryOp accum,
3217                      const GrB_Semiring op,
3218                      const GrB_Matrix A,
3219                      const GrB_Vector u,
3220                      const GrB_Descriptor desc);
```

3221 Parameters

3222 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 3223 that may be accumulated with the result of the matrix-vector product. On output,
 3224 this vector holds the results of the operation.

3225 **mask** (IN) An optional “write” mask that controls which results from this operation are
 3226 stored into the output vector \mathbf{w} . The mask dimensions must match those of the
 3227 vector \mathbf{w} . If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain

3228 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types
 3229 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
 3230 dimensions of `w`), `GrB_NULL` should be specified.

3231 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`
 3232 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 3233 specified.

3234 `op` (IN) Semiring used in the vector-matrix multiply.

3235 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3236 multiplication.

3237 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3238 multiplication.

3239 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 3240 should be specified. Non-default field/value pairs are listed as follows:
 3241

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.

3243 Return Values

3244 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 3245 blocking mode, this indicates that the compatibility tests on di-
 3246 mensions and domains for the input arguments passed successfully.
 3247 Either way, output vector `w` is ready to be used in the next method
 3248 of the sequence.

3249 `GrB_PANIC` Unknown internal error.

3250 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 3251 GraphBLAS objects (input or output) is in an invalid state caused
 3252 by a previous execution error. Call `GrB_error()` to access any error
 3253 messages generated by the implementation.

3254 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3255 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by
 3256 a call to `new` (or `dup` for matrix or vector parameters).

3257 GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3258 GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with
3259 the corresponding domains of the semiring or accumulation opera-
3260 tor, or the mask's domain is not compatible with **bool** (in the case
3261 where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3262 Description

3263 GrB_mvx computes the matrix-vector product $w = A \oplus . \otimes u$, or, if an optional binary accumulation
3264 operator (\odot) is provided, $w = w \odot (A \oplus . \otimes u)$ (where matrix A can be optionally transposed).
3265 Logically, this operation occurs in three steps:

3266 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
3267 domains/dimensions are tested for compatibility.

3268 **Compute** The indicated computations are carried out.

3269 **Output** The result is written into the output vector, possibly under control of a mask.

3270 Up to four argument vectors or matrices are used in the GrB_mvx operation:

- 3271 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3272 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3273 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3274 4. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3275 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
3276 tested for domain compatibility as follows:

- 3277 1. If **mask** is not GrB_NULL, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3278 must be from one of the pre-defined types of Table 3.2.
- 3279 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 3280 3. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 3281 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 3282 5. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3283 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
3284 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_m xv` ends and the domain mismatch error listed above is returned.

From the argument vectors and matrices, the internal matrices 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. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
4. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

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

1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.
2. $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}(\tilde{\mathbf{A}})$.
3. $\text{size}(\tilde{\mathbf{u}}) = \text{ncols}(\tilde{\mathbf{A}})$.

If any compatibility rule above is violated, execution of `GrB_m xv` 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 matrix-vector multiplication and any additional associated operations. We describe this in terms of two intermediate vectors:

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

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

$$t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3317 where \oplus and \otimes are the additive and multiplicative operators of semiring **op**, respectively.

3318 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3319 • If **accum** = **GrB_NULL**, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3320 • If **accum** is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3321 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3322 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3323 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3324 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3325 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3326 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3327 \end{aligned}$$

3328 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3330 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector **w**,
 3331 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3332 mask which acts as a “write mask”.

- 3333 • If **desc[GrB_OUTP].GrB_REPLACE** is set, then any values in **w** on input to this operation are
 3334 deleted and the content of the new output vector, **w**, is defined as,

$$3335 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3336 • If **desc[GrB_OUTP].GrB_REPLACE** is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3337 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the
 3338 mask are unchanged:

$$3339 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3340 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new content
 3341 of vector **w** is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method
 3342 exits with return value **GrB_SUCCESS** and the new content of vector **w** is as defined above but
 3343 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3344 sequence.

3345 4.3.4 eWiseMult: Element-wise multiplication

3346 **Note:** The difference between **eWiseAdd** and **eWiseMult** is not about the element-wise operation
 3347 but how the index sets are treated. **eWiseAdd** returns an object whose indices are the “union” of
 3348 the indices of the inputs whereas **eWiseMult** returns an object whose indices are the “intersection”
 3349 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
 3350 the set of values from the resulting index set.

4.3.4.1 eWiseMult: Vector variant

Perform element-wise (general) multiplication on the intersection of elements of two vectors, producing a third vector as result.

C Syntax

```
GrB_Info GrB_eWiseMult(GrB_Vector      w,
                       const GrB_Vector mask,
                       const GrB_BinaryOp accum,
                       const GrB_Semiring op,
                       const GrB_Vector u,
                       const GrB_Vector v,
                       const GrB_Descriptor desc);

GrB_Info GrB_eWiseMult(GrB_Vector      w,
                       const GrB_Vector mask,
                       const GrB_BinaryOp accum,
                       const GrB_Monoid op,
                       const GrB_Vector u,
                       const GrB_Vector v,
                       const GrB_Descriptor desc);

GrB_Info GrB_eWiseMult(GrB_Vector      w,
                       const GrB_Vector mask,
                       const GrB_BinaryOp accum,
                       const GrB_BinaryOp op,
                       const GrB_Vector u,
                       const GrB_Vector v,
                       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 element-wise 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**

3389 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3390 specified.

3391 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3392 operation. Depending on which type is passed, the following defines the binary
3393 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3394 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3395 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-
3396 nored.

3397 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
3398 is ignored.

3399 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3400 operation.

3401 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
3402 operation.

3403 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
3404 should be specified. Non-default field/value pairs are listed as follows:
3405

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.

3407 Return Values

3408 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3409 blocking mode, this indicates that the compatibility tests on di-
3410 mensions and domains for the input arguments passed successfully.
3411 Either way, output vector w is ready to be used in the next method
3412 of the sequence.

3413 GrB_PANIC Unknown internal error.

3414 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3415 GraphBLAS objects (input or output) is in an invalid state caused
3416 by a previous execution error. Call GrB_error() to access any error
3417 messages generated by the implementation.

3418 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3419 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3420 a call to `new` (or `dup` for vector parameters).

3421 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3422 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 3423 responding domains of the binary operator (`op`) or accumulation
 3424 operator, or the mask's domain is not compatible with `bool` (in the
 3425 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3426 Description

3427 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:
 3428 $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$.
 3429 Logically, this operation occurs in three steps:

3430 **Setup** The internal vectors and mask used in the computation are formed and their domains
 3431 and dimensions are tested for compatibility.

3432 **Compute** The indicated computations are carried out.

3433 **Output** The result is written into the output vector, possibly under control of a mask.

3434 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3435 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3436 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3437 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3438 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3439 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)
 3440 are tested for domain compatibility as follows:

- 3441 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 3442 must be from one of the pre-defined types of Table 3.2.
- 3443 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 3444 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 3445 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3446 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3447 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
 3448 the accumulation operator.

3449 Two domains are compatible with each other if values from one domain can be cast to values in
 3450 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3451 compatible with each other. A domain from a user-defined type is only compatible with itself. If any
 3452 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3453 error listed above is returned.

3454 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3455 denotes copy):

- 3456 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3457 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3458 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3459 (b) If `mask \neq GrB_NULL`,
 - 3460 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3461 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3462 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3463 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3464 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3465 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3466 must hold:

- 3467 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$.

3468 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3469 mismatch error listed above is returned.

3470 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3471 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3472 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3473 We describe this in terms of two intermediate vectors:

- 3474 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “product” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3475 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3476 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
 3477 value of each of its elements is computed by:

$$3478 \quad t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))$$

3479 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

3480 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3481 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3482 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3483 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3484 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3485 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3486

$$3487 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3488

$$3489 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3490 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3491 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3492 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3493 mask which acts as a “write mask”.

3494 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in \mathbf{w} on input to this operation are
 3495 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3496 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3497 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3498 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3499 mask are unchanged:

$$3500 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3501 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new content
 3502 of vector \mathbf{w} is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method
 3503 exits with return value **GrB_SUCCESS** and the new content of vector \mathbf{w} is as defined above but
 3504 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3505 sequence.

3506 4.3.4.2 eWiseMult: Matrix variant

3507 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-
 3508 ducing a third matrix as result.

3509 C Syntax

```

3510     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3511                           const GrB_Matrix Mask,
3512                           const GrB_BinaryOp accum,
3513                           const GrB_Semiring op,
3514                           const GrB_Matrix A,
3515                           const GrB_Matrix B,
3516                           const GrB_Descriptor desc);
3517
3518     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3519                           const GrB_Matrix Mask,
3520                           const GrB_BinaryOp accum,
3521                           const GrB_Monoid op,
3522                           const GrB_Matrix A,
3523                           const GrB_Matrix B,
3524                           const GrB_Descriptor desc);
3525
3526     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3527                           const GrB_Matrix Mask,
3528                           const GrB_BinaryOp accum,
3529                           const GrB_BinaryOp op,
3530                           const GrB_Matrix A,
3531                           const GrB_Matrix B,
3532                           const GrB_Descriptor desc);

```

3533 Parameters

3534 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3535 that may be accumulated with the result of the element-wise operation. On output,
3536 the matrix holds the results of the operation.

3537 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3538 stored into the output matrix C. The mask dimensions must match those of the
3539 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3540 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types
3541 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
3542 dimensions of C), `GrB_NULL` should be specified.

3543 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3544 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3545 specified.

3546 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3547 operation. Depending on which type is passed, the following defines the binary
3548 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3549 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 3550 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-
 3551 nored.
 3552 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
 3553 is ignored.

3554 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3555 operation.

3556 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3557 operation.

3558 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3559 should be specified. Non-default field/value pairs are listed as follows:
 3560

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.

3562 Return Values

3563 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3564 blocking mode, this indicates that the compatibility tests on di-
 3565 mensions and domains for the input arguments passed successfully.
 3566 Either way, output matrix C is ready to be used in the next method
 3567 of the sequence.

3568 GrB_PANIC Unknown internal error.

3569 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3570 GraphBLAS objects (input or output) is in an invalid state caused
 3571 by a previous execution error. Call GrB_error() to access any error
 3572 messages generated by the implementation.

3573 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3574 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3575 a call to new (or Matrix_dup for matrix parameters).

3576 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3577 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3578 corresponding domains of the binary operator (`op`) or accumulation
 3579 operator, or the mask's domain is not compatible with `bool` (in the
 3580 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3581 Description

3582 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:
 3583 $C = A \otimes B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \otimes B)$.
 3584 Logically, this operation occurs in three steps:

3585 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3586 and dimensions are tested for compatibility.

3587 **Compute** The indicated computations are carried out.

3588 **Output** The result is written into the output matrix, possibly under control of a mask.

3589 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3590 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3591 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)
- 3592 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3593 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3594 The argument matrices, the “product” operator (`op`), and the accumulation operator (if provided)
 3595 are tested for domain compatibility as follows:

- 3596 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3597 must be from one of the pre-defined types of Table 3.2.
- 3598 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3599 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3600 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3601 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3602 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3603 the accumulation operator.

3604 Two domains are compatible with each other if values from one domain can be cast to values in
 3605 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3606 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3607 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3608 error listed above is returned.

3609 From the argument matrices, the internal matrices and mask used in the computation are formed
 3610 (\leftarrow denotes copy):

- 3611 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3612 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3613 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3614 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3615 (b) If `Mask \neq GrB_NULL`,
 - 3616 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3617 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3618 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3619 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$.
 - 3620 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3621 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3622 4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3623 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3624 must hold:

- 3625 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3626 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3627 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3628 mismatch error listed above is returned.

3629 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3630 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3631 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3632 We describe this in terms of two intermediate matrices:

- 3633 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise product of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3634 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3635 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$
 3636 is created. The value of each of its elements is computed by

$$3637 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3638 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

3639 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

3640 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3641 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3642 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3643 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3644 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3645 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3646 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3649 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3650 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 3651 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 3652 mask which acts as a “write mask”.

3653 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 3654 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3655 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3656 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 3657 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 3658 mask are unchanged:

$$3659 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3660 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3661 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3662 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 3663 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3664 sequence.

3665 4.3.5 eWiseAdd: Element-wise addition

3666 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation
 3667 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of
 3668 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”
 3669 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
 3670 the set of values from the resulting index set.

3671 4.3.5.1 eWiseAdd: Vector variant

3672 Perform element-wise (general) addition on the elements of two vectors, producing a third vector
3673 as result.

3674 C Syntax

```
3675     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3676                          const GrB_Vector mask,  
3677                          const GrB_BinaryOp accum,  
3678                          const GrB_Semiring op,  
3679                          const GrB_Vector u,  
3680                          const GrB_Vector v,  
3681                          const GrB_Descriptor desc);  
3682  
3683     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3684                          const GrB_Vector mask,  
3685                          const GrB_BinaryOp accum,  
3686                          const GrB_Monoid op,  
3687                          const GrB_Vector u,  
3688                          const GrB_Vector v,  
3689                          const GrB_Descriptor desc);  
3690  
3691     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3692                          const GrB_Vector mask,  
3693                          const GrB_BinaryOp accum,  
3694                          const GrB_BinaryOp op,  
3695                          const GrB_Vector u,  
3696                          const GrB_Vector v,  
3697                          const GrB_Descriptor desc);
```

3698 Parameters

3699 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3700 that may be accumulated with the result of the element-wise operation. On output,
3701 this vector holds the results of the operation.

3702 **mask** (IN) An optional “write” mask that controls which results from this operation are
3703 stored into the output vector **w**. The mask dimensions must match those of the
3704 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3705 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3706 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
3707 dimensions of **w**), **GrB_NULL** should be specified.

3708 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3709 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3710 specified.

3711 op (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
3712 operation. Depending on which type is passed, the following defines the binary
3713 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3714 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3715 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-
3716 nored.

3717 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-
3718 nary op and additive identity are ignored.

3719 u (IN) The GraphBLAS vector holding the values for the left-hand vector in the
3720 operation.

3721 v (IN) The GraphBLAS vector holding the values for the right-hand vector in the
3722 operation.

3723 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
3724 should be specified. Non-default field/value pairs are listed as follows:

3725

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.

3726

3727 Return Values

3728 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3729 blocking mode, this indicates that the compatibility tests on di-
3730 mensions and domains for the input arguments passed successfully.
3731 Either way, output vector w is ready to be used in the next method
3732 of the sequence.

3733 GrB_PANIC Unknown internal error.

3734 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3735 GraphBLAS objects (input or output) is in an invalid state caused
3736 by a previous execution error. Call GrB_error() to access any error
3737 messages generated by the implementation.

3738 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3739 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3740 a call to `new` (or `dup` for vector parameters).

3741 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3742 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
3743 responding domains of the binary operator (`op`) or accumulation
3744 operator, or the mask's domain is not compatible with `bool` (in the
3745 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3746 Description

3747 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors: $w =$
3748 $u \oplus v$, or, if an optional binary accumulation operator (\odot) is provided, $w = w \odot (u \oplus v)$. Logically,
3749 this operation occurs in three steps:

3750 **Setup** The internal vectors and mask used in the computation are formed and their domains
3751 and dimensions are tested for compatibility.

3752 **Compute** The indicated computations are carried out.

3753 **Output** The result is written into the output vector, possibly under control of a mask.

3754 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3755 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3756 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3757 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3758 4. $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3759 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are
3760 tested for domain compatibility as follows:

- 3761 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3762 must be from one of the pre-defined types of Table 3.2.
- 3763 2. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3764 3. $\mathbf{D}(v)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3765 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3766 5. $\mathbf{D}(u)$ and $\mathbf{D}(v)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3767 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3768 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
3769 the accumulation operator.

3770 Two domains are compatible with each other if values from one domain can be cast to values in
 3771 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3772 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3773 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch
 3774 error listed above is returned.

3775 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3776 denotes copy):

- 3777 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3778 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3779 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3780 (b) If `mask \neq GrB_NULL`,
 - 3781 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3782 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3783 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3784 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3785 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3786 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3787 must hold:

- 3788 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$.

3789 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 3790 mismatch error listed above is returned.

3791 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3792 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3793 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3794 We describe this in terms of two intermediate vectors:

- 3795 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “sum” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3796 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3797 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
 3798 value of each of its elements is computed by:

$$\begin{aligned}
 3799 \quad t_i &= (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})) \\
 3800 \\
 3801 \quad t_i &= \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})))
 \end{aligned}$$

3802
3803

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3804

where the difference operator in the previous expressions refers to set difference.

3805

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

3806

- If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3807

- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

3808

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

3809

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 $\tilde{\mathbf{t}}$.

3810

3811

$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

3812

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3813

3814

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3815

3816

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

3817

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”.

3818

3819

3820

- 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,

3821

3822

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

3823

- 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:

3824

3825

3826

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

3827

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.

3828

3829

3830

3831

3832

4.3.5.2 eWiseAdd: Matrix variant

3833

Perform element-wise (general) addition on the elements of two matrices, producing a third matrix as result.

3834

3835 C Syntax

```

3836     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3837                           const GrB_Matrix Mask,
3838                           const GrB_BinaryOp accum,
3839                           const GrB_Semiring op,
3840                           const GrB_Matrix A,
3841                           const GrB_Matrix B,
3842                           const GrB_Descriptor desc);
3843
3844     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3845                           const GrB_Matrix Mask,
3846                           const GrB_BinaryOp accum,
3847                           const GrB_Monoid op,
3848                           const GrB_Matrix A,
3849                           const GrB_Matrix B,
3850                           const GrB_Descriptor desc);
3851
3852     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3853                           const GrB_Matrix Mask,
3854                           const GrB_BinaryOp accum,
3855                           const GrB_BinaryOp op,
3856                           const GrB_Matrix A,
3857                           const GrB_Matrix B,
3858                           const GrB_Descriptor desc);

```

3859 Parameters

3860 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3861 that may be accumulated with the result of the element-wise operation. On output,
3862 the matrix holds the results of the operation.

3863 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3864 stored into the output matrix C. The mask dimensions must match those of the
3865 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3866 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types
3867 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
3868 dimensions of C), `GrB_NULL` should be specified.

3869 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3870 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3871 specified.

3872 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
3873 operation. Depending on which type is passed, the following defines the binary
3874 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3875 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 3876 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-
 3877 nored.
 3878 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-
 3879 nary op and additive identity are ignored.

3880 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3881 operation.

3882 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3883 operation.

3884 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3885 should be specified. Non-default field/value pairs are listed as follows:
 3886

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.

3888 Return Values

3889 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3890 blocking mode, this indicates that the compatibility tests on di-
 3891 mensions and domains for the input arguments passed successfully.
 3892 Either way, output matrix C is ready to be used in the next method
 3893 of the sequence.

3894 GrB_PANIC Unknown internal error.

3895 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3896 GraphBLAS objects (input or output) is in an invalid state caused
 3897 by a previous execution error. Call GrB_error() to access any error
 3898 messages generated by the implementation.

3899 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3900 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3901 a call to new (or Matrix_dup for matrix parameters).

3902 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3903 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3904 corresponding domains of the binary operator (op) or accumulation
 3905 operator, or the mask's domain is not compatible with `bool` (in the
 3906 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3907 Description

3908 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:
 3909 $C = A \oplus B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \oplus B)$.
 3910 Logically, this operation occurs in three steps:

3911 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3912 and dimensions are tested for compatibility.

3913 **Compute** The indicated computations are carried out.

3914 **Output** The result is written into the output matrix, possibly under control of a mask.

3915 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3916 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3917 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)
- 3918 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3919 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3920 The argument matrices, the “sum” operator (op), and the accumulation operator (if provided) are
 3921 tested for domain compatibility as follows:

- 3922 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3923 must be from one of the pre-defined types of Table 3.2.
- 3924 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3925 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3926 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3927 5. $\mathbf{D}(A)$ and $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3928 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3929 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3930 the accumulation operator.

3931 Two domains are compatible with each other if values from one domain can be cast to values in
 3932 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 3933 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3934 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch
 3935 error listed above is returned.

3936 From the argument matrices, the internal matrices and mask used in the computation are formed
 3937 (\leftarrow denotes copy):

- 3938 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3939 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3940 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3941 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3942 (b) If `Mask \neq GrB_NULL`,
 - 3943 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3944 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3945 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3946 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 3947 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3948 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3949 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3950 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3951 must hold:

- 3952 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3953 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3954 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 3955 mismatch error listed above is returned.

3956 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3957 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3958 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3959 We describe this in terms of two intermediate matrices:

- 3960 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise sum of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3961 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3962 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$
 3963 is created. The value of each of its elements is computed by

$$\begin{aligned}
 3964 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\
 3965 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\
 3966 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})))
 \end{aligned}$$

3969 where the difference operator in the previous expressions refers to set difference.

3970 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3971 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3972 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3973 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3974 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3975 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned}
 3976 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 3977 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 3978 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 3979 \quad & \\
 3980 \quad &
 \end{aligned}$$

3981 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3982 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 3983 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 3984 mask which acts as a “write mask”.

- 3985 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 3986 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3987 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3988 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 3989 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 3990 mask are unchanged:

$$3991 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3992 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3993 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3994 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 3995 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3996 sequence.

3997 4.3.6 extract: Selecting sub-graphs

3998 Extract a subset of a matrix or vector.

3999 4.3.6.1 extract: Standard vector variant

4000 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector
4001 whose size is equal to the number of indices.

4002 C Syntax

```
4003      GrB_Info GrB_extract(GrB_Vector      w,  
4004                          const GrB_Vector mask,  
4005                          const GrB_BinaryOp accum,  
4006                          const GrB_Vector u,  
4007                          const GrB_Index *indices,  
4008                          GrB_Index      nindices,  
4009                          const GrB_Descriptor desc);
```

4010 Parameters

4011 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4012 that may be accumulated with the result of the extract operation. On output, this
4013 vector holds the results of the operation.

4014 **mask** (IN) An optional “write” mask that controls which results from this operation are
4015 stored into the output vector **w**. The mask dimensions must match those of the
4016 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4017 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4018 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4019 dimensions of **w**), **GrB_NULL** should be specified.

4020 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4021 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4022 specified.

4023 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4024 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of
4025 elements from **u** that are extracted. If all elements of **u** are to be extracted in order
4026 from 0 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution
4027 mode and return value, this array may be manipulated by the caller after this
4028 operation returns without affecting any deferred computations for this operation.

4029 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4030 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4031 should be specified. Non-default field/value pairs are listed as follows:

4032

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 .

4033

4034 Return Values

4035 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 4036 blocking mode, this indicates that the compatibility tests on
 4037 dimensions and domains for the input arguments passed suc-
 4038 cessfully. Either way, output vector **w** is ready to be used in the
 4039 next method of the sequence.

4040 GrB_PANIC Unknown internal error.

4041 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 4042 opaque GraphBLAS objects (input or output) is in an invalid
 4043 state caused by a previous execution error. Call GrB_error() to
 4044 access any error messages generated by the implementation.

4045 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4046 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
 4047 by a call to **new** (or **dup** for vector parameters).

4048 GrB_INDEX_OUT_OF_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-
 4049 blocking mode, this error can be deferred.

4050 GrB_DIMENSION_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices** \neq **size(w)**.

4051 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each
 4052 other or the corresponding domains of the accumulation oper-
 4053 ator, or the mask's domain is not compatible with **bool** (in the
 4054 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4055 GrB_NULL_POINTER Argument **row_indices** is a NULL pointer.

4056 Description

4057 This variant of GrB_extract computes the result of extracting a subset of locations from a Graph-
 4058 BLAS vector in a specific order: **w** = **u(indices)**; or, if an optional binary accumulation operator

4059 (\odot) is provided, $w = w \odot u(\text{indices})$. More explicitly:

$$4060 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4061 Logically, this operation occurs in three steps:

4062 **Setup** The internal vectors and mask used in the computation are formed and their domains
4063 and dimensions are tested for compatibility.

4064 **Compute** The indicated computations are carried out.

4065 **Output** The result is written into the output vector, possibly under control of a mask.

4066 Up to three argument vectors are used in this GrB_extract operation:

- 4067 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4068 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4069 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4070 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4071 bility as follows:

- 4072 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4073 must be from one of the pre-defined types of Table 3.2.
- 4074 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(u)$.
- 4075 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4076 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4077 mulation operator.

4078 Two domains are compatible with each other if values from one domain can be cast to values in
4079 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4080 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4081 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
4082 error listed above is returned.

4083 From the arguments, the internal vectors, mask, and index array used in the computation are
4084 formed (\leftarrow denotes copy):

- 4085 1. Vector $\tilde{w} \leftarrow w$.
- 4086 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
4087 (a) If `mask = GrB_NULL`, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.

- 4088 (b) If $\text{mask} \neq \text{GrB_NULL}$,
 4089 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 4090 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 4091 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
- 4092 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 4093 4. The internal index array, $\widetilde{\mathbf{I}}$, is computed from argument indices as follows:
- 4094 (a) If $\text{indices} = \text{GrB_ALL}$, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.
 4095 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

4096 The internal vectors and mask are checked for dimension compatibility. The following conditions
 4097 must hold:

- 4098 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
 4099 2. $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$.

4100 If any compatibility rule above is violated, execution of GrB_extract ends and the dimension mis-
 4101 match error listed above is returned.

4102 From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with
 4103 GrB_SUCCESS return code and defer any computation and/or execution error codes.

4104 We are now ready to carry out the extract and any additional associated operations. We describe
 4105 this in terms of two intermediate vectors:

- 4106 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{w}}$.
- 4107 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4108 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4109 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4110 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
 4111 GrB_extract ends and the index-out-of-bounds error listed above is generated. In GrB_NONBLOCKING
 4112 mode, the error can be deferred until a sequence-terminating $\text{GrB_wait}()$ is called. Regardless, the
 4113 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4114 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 4115 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 4116 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$4117 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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.

4184 Return Values

4185	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
4186		blocking mode, this indicates that the compatibility tests on
4187		dimensions and domains for the input arguments passed suc-
4188		cessfully. Either way, output matrix C is ready to be used in the
4189		next method of the sequence.
4190	GrB_PANIC	Unknown internal error.
4191	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
4192		opaque GraphBLAS objects (input or output) is in an invalid
4193		state caused by a previous execution error. Call <code>GrB_error()</code> to
4194		access any error messages generated by the implementation.
4195	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
4196	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
4197		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4198	GrB_INDEX_OUT_OF_BOUNDS	A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or
4199		a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In
4200		non-blocking mode, this error can be deferred.
4201	GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, <code>nrows</code> \neq <code>nrows(C)</code> , or
4202		<code>ncols</code> \neq <code>ncols(C)</code> .
4203	GrB_DOMAIN_MISMATCH	The domains of the various matrices are incompatible with each
4204		other or the corresponding domains of the accumulation oper-
4205		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4206		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4207	GrB_NULL_POINTER	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4208		is a NULL pointer, or both.

4209 Description

4210 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified
 4211 rows and columns of a GraphBLAS matrix in a specific order: $C = A(\text{row_indices}, \text{col_indices})$; or,
 4212 if an optional binary accumulation operator (\odot) is provided, $C = C \odot A(\text{row_indices}, \text{col_indices})$.
 4213 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4215 Logically, this operation occurs in three steps:

4216 **Setup** The internal matrices and mask used in the computation are formed and their domains
 4217 and dimensions are tested for compatibility.

4218 **Compute** The indicated computations are carried out.

4219 **Output** The result is written into the output matrix, possibly under control of a mask.

4220 Up to three argument matrices are used in the `GrB_extract` operation:

- 4221 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4222 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)
- 4223 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4224 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
4225 ibility as follows:

- 4226 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
4227 must be from one of the pre-defined types of Table 3.2.
- 4228 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 4229 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4230 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4231 mulation operator.

4232 Two domains are compatible with each other if values from one domain can be cast to values in
4233 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4234 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4235 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
4236 error listed above is returned.

4237 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are
4238 formed (\leftarrow denotes copy):

- 4239 1. Matrix $\tilde{C} \leftarrow C$.
- 4240 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:
 - 4241 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
4242 $j < \mathbf{ncols}(C)\} \rangle$.
 - 4243 (b) If `Mask` \neq `GrB_NULL`,
 - 4244 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
4245 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - 4246 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
4247 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - 4248 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
- 4249 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

- 4250 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4251 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 4252 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \text{nrows}$.
- 4253 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4254 (a) If `col_indices` = `GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$.
- 4255 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \text{ncols}$.

4256 The internal matrices and mask are checked for dimension compatibility. The following conditions
4257 must hold:

- 4258 1. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$.
- 4259 2. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$.
- 4260 3. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$.
- 4261 4. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$.

4262 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
4263 match error listed above is returned.

4264 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4265 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4266 We are now ready to carry out the extract and any additional associated operations. We describe
4267 this in terms of two intermediate matrices:

- 4268 • $\tilde{\mathbf{T}}$: The matrix holding the extraction from $\tilde{\mathbf{A}}$.
- 4269 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4270 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

4271
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{ (i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}}) \} \rangle.$$

4272 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 $\tilde{\mathbf{J}}$
4273 array is not in the range $[0, \text{ncols}(\tilde{\mathbf{A}}))$, the execution of `GrB_extract` ends and the index out-of-
4274 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
4275 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
4276 this point forward in the sequence.

4277 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4278 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

4279 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4280 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4281 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4282 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4283 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4284 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4285 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4286 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4289 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
4290 using what is called a *standard matrix mask and replace*. This is carried out under control of the
4291 mask which acts as a “write mask”.

4292 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
4293 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4294 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4295 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
4296 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
4297 mask are unchanged:

$$4298 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4299 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4300 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4301 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
4302 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4303 sequence.

4304 4.3.6.3 extract: Column (and row) variant

4305 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the
4306 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as
4307 well.

4308 C Syntax

```

4309         GrB_Info GrB_extract(GrB_Vector      w,
4310                               const GrB_Vector mask,
4311                               const GrB_BinaryOp accum,
4312                               const GrB_Matrix A,
4313                               const GrB_Index *row_indices,
4314                               GrB_Index nrows,
4315                               GrB_Index col_index,
4316                               const GrB_Descriptor desc);

```

4317 Parameters

4318 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4319 that may be accumulated with the result of the extract operation. On output, this
4320 vector holds the results of the operation.

4321 **mask** (IN) An optional “write” mask that controls which results from this operation are
4322 stored into the output vector **w**. The mask dimensions must match those of the
4323 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4324 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4325 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4326 dimensions of **w**), **GrB_NULL** should be specified.

4327 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4328 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4329 specified.

4330 **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4331 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations
4332 within the specified column of **A** from which elements are extracted. If elements in
4333 all rows of **A** are to be extracted in order, **GrB_ALL** should be specified. Regardless
4334 of execution mode and return value, this array may be manipulated by the caller
4335 after this operation returns without affecting any deferred computations for this
4336 operation.

4337 **nrows** (IN) The number of indices in the **row_indices** array. Must be equal to **size(w)**.

4338 **col_index** (IN) The index of the column of **A** from which to extract values. It must be in the
4339 range $[0, \mathbf{ncols}(A))$.

4340 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4341 should be specified. Non-default field/value pairs are listed as follows:
4342

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

4370 an optional binary accumulation operator (\odot) is provided, $w = w \odot A(:, \text{col_index})(\text{row_indices})$.
 4371 More explicitly:

$$4372 \quad \begin{aligned} w(i) &= A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4373 Logically, this operation occurs in three steps:

4374 **Setup** The internal matrices, vectors, and mask used in the computation are formed and their
 4375 domains and dimensions are tested for compatibility.

4376 **Compute** The indicated computations are carried out.

4377 **Output** The result is written into the output vector, possibly under control of a mask.

4378 Up to three argument vectors and matrices are used in this GrB_extract operation:

- 4379 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4380 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4381 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4382 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain
 4383 compatibility as follows:

- 4384 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
 4385 must be from one of the pre-defined types of Table 3.2.
- 4386 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(A)$.
- 4387 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4388 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4389 mulation operator.

4390 Two domains are compatible with each other if values from one domain can be cast to values in
 4391 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4392 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4393 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
 4394 error listed above is returned.

4395 From the arguments, the internal vector, matrix, mask, and index array used in the computation
 4396 are formed (\leftarrow denotes copy):

- 4397 1. Vector $\tilde{w} \leftarrow w$.
- 4398 2. One-dimensional mask, \tilde{m} , is computed from argument **mask** as follows:
 4399 (a) If **mask** = GrB_NULL, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.

4400 (b) If $\text{mask} \neq \text{GrB_NULL}$,
 4401 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 4402 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 4403 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
 4404 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
 4405 4. The internal row index array, $\widetilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
 4406 (a) If `indices = GrB_ALL`, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
 4407 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$.

4408 The internal vector, `mask`, and index array are checked for dimension compatibility. The following
 4409 conditions must hold:

- 4410 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4411 2. $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$.

4412 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
 4413 match error listed above is returned.

4414 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4415 1. $0 \leq \text{col_index} < \text{ncols}(\mathbf{A})$

4416 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above
 4417 is returned.

4418 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4419 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4420 We are now ready to carry out the extract and any additional associated operations. We describe
 4421 this in terms of two intermediate vectors:

- 4422 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from a column of $\widetilde{\mathbf{A}}$.
- 4423 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4424 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$4425 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4426 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`
 4427 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,
 4428 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result
 4429 vector, \mathbf{w} , is invalid from this point forward in the sequence.

4430 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

4431 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

4432 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4433 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4434 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4435 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4436 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

4437

$$4438 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4439

$$4440 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4441 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4442 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
4443 using what is called a *standard vector mask and replace*. This is carried out under control of the
4444 mask which acts as a “write mask”.

4445 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
4446 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4447 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4448 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4449 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
4450 mask are unchanged:

$$4451 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4452 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4453 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4454 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
4455 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4456 sequence.

4457 4.3.7 assign: Modifying sub-graphs

4458 Assign the contents of a subset of a matrix or vector.

4459 4.3.7.1 assign: Standard vector variant

4460 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.
4461 The size of the input vector is the same size as the index array provided.

4462 C Syntax

```

4463         GrB_Info GrB_assign(GrB_Vector      w,
4464                             const GrB_Vector mask,
4465                             const GrB_BinaryOp accum,
4466                             const GrB_Vector u,
4467                             const GrB_Index *indices,
4468                             GrB_Index      nindices,
4469                             const GrB_Descriptor desc);

```

4470 Parameters

4471 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4472 that may be accumulated with the result of the assign operation. On output, this
4473 vector holds the results of the operation.

4474 **mask** (IN) An optional “write” mask that controls which results from this operation are
4475 stored into the output vector **w**. The mask dimensions must match those of the
4476 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4477 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4478 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4479 dimensions of **w**), **GrB_NULL** should be specified.

4480 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4481 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4482 specified.

4483 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4484 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4485 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0
4486 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4487 and return value, this array may be manipulated by the caller after this operation
4488 returns without affecting any deferred computations for this operation. If this
4489 array contains duplicate values, it implies in assignment of more than one value to
4490 the same location which leads to undefined results.

4491 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4492 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4493 should be specified. Non-default field/value pairs are listed as follows:

4494

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(indices) = u**; or, if an optional binary accumulation operator (\odot) is provided, **w(indices) = w(indices) \odot u**. More explicitly:

$$\begin{aligned} \mathbf{w}(\mathbf{indices}[i]) &= \mathbf{u}(i), \forall i : 0 \leq i < \mathbf{nindices}, \text{ or} \\ \mathbf{w}(\mathbf{indices}[i]) &= \mathbf{w}(\mathbf{indices}[i]) \odot \mathbf{u}(i), \forall i : 0 \leq i < \mathbf{nindices}. \end{aligned}$$

4523 Logically, this operation occurs in three steps:

4524 **Setup** The internal vectors and mask used in the computation are formed and their domains
4525 and dimensions are tested for compatibility.

4526 **Compute** The indicated computations are carried out.

4527 **Output** The result is written into the output vector, possibly under control of a mask.

4528 Up to three argument vectors are used in the `GrB_assign` operation:

- 4529 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4530 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4531 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4532 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4533 bility as follows:

- 4534 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
4535 must be from one of the pre-defined types of Table 3.2.
- 4536 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 4537 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
4538 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
4539 mulation operator.

4540 Two domains are compatible with each other if values from one domain can be cast to values in
4541 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4542 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4543 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4544 error listed above is returned.

4545 From the arguments, the internal vectors, mask and index array used in the computation are formed
4546 (\leftarrow denotes copy):

- 4547 1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 4548 2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 4549 (a) If `mask` = `GrB_NULL`, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 4550 (b) If `mask` \neq `GrB_NULL`,
 - 4551 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 4552 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 4553 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.

4554 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4555 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:

4556 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i$, $\forall i : 0 \leq i < \text{nindices}$.

4557 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i]$, $\forall i : 0 \leq i < \text{nindices}$.

4558 The internal vector and mask are checked for dimension compatibility. The following conditions
4559 must hold:

4560 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4561 2. $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$.

4562 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4563 match error listed above is returned.

4564 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4565 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4566 We are now ready to carry out the assign and any additional associated operations. We describe
4567 this in terms of two intermediate vectors:

- 4568 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.
- 4569 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4570 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4571 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4572 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{w}}$, computation
4573 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`
4574 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4575 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4576 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4577 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4578 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4579 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4580 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
4581 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4582 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4583 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4584 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4585 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4587 where the difference operator refers to set difference.

- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

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

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.

4.3.7.2 assign: Standard matrix variant

Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices. The dimensions of the input matrix are the same size as the row and column index arrays provided.

C Syntax

```
GrB_Info GrB_assign(GrB_Matrix      C,
                    const GrB_Matrix Mask,
                    const GrB_BinaryOp accum,
                    const GrB_Matrix A,
```

```

4621         const GrB_Index      *row_indices,
4622         GrB_Index             nrows,
4623         const GrB_Index      *col_indices,
4624         GrB_Index             ncols,
4625         const GrB_Descriptor desc);

```

4626 Parameters

4627 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
4628 that may be accumulated with the result of the assign operation. On output, the
4629 matrix holds the results of the operation.

4630 **Mask** (IN) An optional “write” mask that controls which results from this operation are
4631 stored into the output matrix **C**. The mask dimensions must match those of the
4632 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4633 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
4634 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
4635 dimensions of **C**), **GrB_NULL** should be specified.

4636 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4637 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4638 specified.

4639 **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4640 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
4641 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
4642 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
4643 this array may be manipulated by the caller after this operation returns without
4644 affecting any deferred computations for this operation. If this array contains du-
4645 plicate values, it implies assignment of more than one value to the same location
4646 which leads to undefined results.

4647 **nrows** (IN) The number of values in the **row_indices** array. Must be equal to **nrows(A)**
4648 if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4649 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns
4650 of **C** that are assigned. If all columns of **C** are to be assigned in order from 0
4651 to **ncols** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4652 and return value, this array may be manipulated by the caller after this operation
4653 returns without affecting any deferred computations for this operation. If this
4654 array contains duplicate values, it implies assignment of more than one value to
4655 the same location which leads to undefined results.

4656 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **ncols(A)** if **A** is
4657 not transposed, or equal to **nrows(A)** if **A** is transposed.

4658 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4659 should be specified. Non-default field/value pairs are listed as follows:
 4660

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.

4662 Return Values

4663 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 4664 blocking mode, this indicates that the compatibility tests on
 4665 dimensions and domains for the input arguments passed suc-
 4666 cessfully. Either way, output matrix C is ready to be used in the
 4667 next method of the sequence.

4668 GrB_PANIC Unknown internal error.

4669 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 4670 opaque GraphBLAS objects (input or output) is in an invalid
 4671 state caused by a previous execution error. Call GrB_error() to
 4672 access any error messages generated by the implementation.

4673 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

4674 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
 4675 by a call to new (or Matrix_dup for matrix parameters).

4676 GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to nrows(C), or
 4677 a value in col_indices is greater than or equal to ncols(C). In
 4678 non-blocking mode, this can be reported as an execution error.

4679 GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrow \neq nrow(A),
 4680 or ncol \neq ncol(A).

4681 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with each
 4682 other or the corresponding domains of the accumulation oper-
 4683 ator, or the mask's domain is not compatible with bool (in the
 4684 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4685 GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices
 4686 is a NULL pointer, or both.

4687 Description

4688 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows
 4689 and columns in `C` in a specified order: $C(\text{row_indices}, \text{col_indices}) = A$; or, if an optional binary
 4690 accumulation operator (\odot) is provided, $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot$
 4691 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned} C(\text{row_indices}[i], \text{col_indices}[j]) &= A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4692 \quad C(\text{row_indices}[i], \text{col_indices}[j]) &= C(\text{row_indices}[i], \text{col_indices}[j]) \odot A(i, j), \\ &\quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4693 Logically, this operation occurs in three steps:

4694 Setup The internal matrices and mask used in the computation are formed and their domains
 4695 and dimensions are tested for compatibility.

4696 Compute The indicated computations are carried out.

4697 Output The result is written into the output matrix, possibly under control of a mask.

4698 Up to three argument matrices are used in the `GrB_assign` operation:

- 4699 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4700 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)
- 4701 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4702 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
 4703 ibility as follows:

- 4704 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 4705 must be from one of the pre-defined types of Table 3.2.
- 4706 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 4707 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4708 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4709 mulation operator.

4710 Two domains are compatible with each other if values from one domain can be cast to values in
 4711 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 4712 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4713 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4714 error listed above is returned.

4715 From the arguments, the internal matrices, mask, and index arrays used in the computation are
 4716 formed (\leftarrow denotes copy):

- 4717 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4718 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
- 4719 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
4720 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 4721 (b) If `Mask \neq GrB_NULL`,
- 4722 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
4723 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
- 4724 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
4725 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
- 4726 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4727 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 4728 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4729 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
- 4730 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 4731 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4732 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
- 4733 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4734 The internal matrices and mask are checked for dimension compatibility. The following conditions
4735 must hold:

- 4736 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 4737 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 4738 3. $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$.
- 4739 4. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$.

4740 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4741 match error listed above is returned.

4742 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4743 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4744 We are now ready to carry out the assign and any additional associated operations. We describe
4745 this in terms of two intermediate vectors:

- 4746 • $\tilde{\mathbf{T}}$: The matrix holding the contents from $\tilde{\mathbf{A}}$ in their destination locations relative to $\tilde{\mathbf{C}}$.
- 4747 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4748 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4749 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4750 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
 4751 $\tilde{\mathbf{J}}$ array is not in the range $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-
 4752 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
 4753 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
 4754 this point forward in the sequence.

4755 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 4756 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$4757 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4758 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4759 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 4760 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
 4761 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

4762 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4763 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4764 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4765 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4767 where the difference operator refers to set difference.

- 4768 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4769 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4770 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4771 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4772 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4773 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4774 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4775 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4777 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4778 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4779 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4780 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

4814 bool or any of the predefined “built-in” types in Table 3.2. If the default mask
 4815 is desired (i.e., a mask that is all true with the dimensions of a column of C),
 4816 GrB_NULL should be specified.

4817 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 4818 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 4819 specified.

4820 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column
 4821 of C.

4822 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
 4823 the specified column of C that are to be assigned. If all elements of the column
 4824 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB_ALL should
 4825 be specified. Regardless of execution mode and return value, this array may be
 4826 manipulated by the caller after this operation returns without affecting any de-
 4827 ferred computations for this operation. If this array contains duplicate values, it
 4828 implies in assignment of more than one value to the same location which leads to
 4829 undefined results.

4830 **nrows** (IN) The number of values in **row_indices** array. Must be equal to **size(u)**.

4831 **col_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4832 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4833 should be specified. Non-default field/value pairs are listed as follows:

4834

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.

4835

4836 Return Values

4837 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 4838 blocking mode, this indicates that the compatibility tests on
 4839 dimensions and domains for the input arguments passed suc-
 4840 cessfully. Either way, output matrix C is ready to be used in the
 4841 next method of the sequence.

4842 **GrB_PANIC** Unknown internal error.

4874 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4875 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
4876 compatibility as follows:

4877 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4878 must be from one of the pre-defined types of Table 3.2.

4879 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}(\mathbf{u})$.

4880 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4881 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4882 mulation operator.

4883 Two domains are compatible with each other if values from one domain can be cast to values in
4884 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
4885 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4886 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4887 error listed above is returned.

4888 The `col_index` parameter is checked for a valid value. The following condition must hold:

4889 1. $0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{C})$

4890 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
4891 is returned.

4892 From the arguments, the internal vectors, `mask`, and index array used in the computation are
4893 formed (\leftarrow denotes copy):

4894 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a column of \mathbf{C} as follows:

4895
$$\tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4896 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

4897 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$.

4898 (b) If `mask` \neq `GrB_NULL`,

4899 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

4900 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

4901 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

4902 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4903 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:

4904 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.

4905 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i]$, $\forall i : 0 \leq i < \text{nrows}$.

4906 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
4907 conditions must hold:

- 4908 1. $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4909 2. $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$.

4910 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4911 match error listed above is returned.

4912 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4913 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4914 We are now ready to carry out the assign and any additional associated operations. We describe
4915 this in terms of two intermediate vectors:

- 4916 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 4917 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4918 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4919 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4920 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
4921 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
4922 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4923 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4924 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4925 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4926 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4927 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4928 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
4929 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4930 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4931 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$4932 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4933 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4935 where the difference operator refers to set difference.

- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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

$$z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

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

Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final result matrix, $\mathbf{C}(:, \text{col_index})$. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(:, \text{col_index})$ on input to this operation are deleted and the new contents of the column is given by:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are copied into the column of the final result matrix, $\mathbf{C}(:, \text{col_index})$, and elements of this column that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} \mathbf{L}(\mathbf{C}) = & \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \\ & \{(i, \text{col_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ & \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

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

4.3.7.4 `assign`: Row variant

Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the output cannot be transposed, a different variant of `assign` is provided to assign to a column of a matrix.

4966 C Syntax

```
4967         GrB_Info GrB_assign(GrB_Matrix      C,  
4968                             const GrB_Vector mask,  
4969                             const GrB_BinaryOp accum,  
4970                             const GrB_Vector u,  
4971                             GrB_Index      row_index,  
4972                             const GrB_Index *col_indices,  
4973                             GrB_Index      ncols,  
4974                             const GrB_Descriptor desc);
```

4975 Parameters

4976 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values
4977 that may be accumulated with the result of the assign operation. On output, this
4978 matrix holds the results of the operation.

4979 **mask** (IN) An optional “write” mask that controls which results from this operation are
4980 stored into the specified row of the output matrix **C**. The mask dimensions must
4981 match those of a single row of the matrix **C**. If the **GrB_STRUCTURE** descriptor
4982 is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or
4983 any of the predefined “built-in” types in Table 3.2. If the default mask is desired
4984 (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB_NULL** should
4985 be specified.

4986 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4987 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4988 specified.

4989 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of
4990 **C**.

4991 **row_index** (IN) The index of the row in **C** to assign. Must be in the range $[0, \mathbf{nrows}(\mathbf{C})]$.

4992 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4993 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to
4994 be assigned in order from index 0 to $\mathbf{ncols} - 1$, then **GrB_ALL** should be specified.
4995 Regardless of execution mode and return value, this array may be manipulated by
4996 the caller after this operation returns without affecting any deferred computations
4997 for this operation. If this array contains duplicate values, it implies in assignment
4998 of more than one value to the same location which leads to undefined results.

4999 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **size(u)**.

5000 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5001 should be specified. Non-default field/value pairs are listed as follows:
5002

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:

5031 $C(\text{row_index}, :) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(\text{row_index}, :$
 5032 $) = C(\text{row_index}, :) \odot u$. Taking order of `col_indices` into account it is more explicitly written as:

5033 $C(\text{row_index}, \text{col_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$
 $C(\text{row_index}, \text{col_indices}[j]) = C(\text{row_index}, \text{col_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$

5034 Logically, this operation occurs in three steps:

5035 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
 5036 domains and dimensions are tested for compatibility.

5037 **Compute** The indicated computations are carried out.

5038 **Output** The result is written into the output matrix, possibly under control of a mask.

5039 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5040 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5041 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 5042 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5043 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
 5044 compatibility as follows:

- 5045 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 5046 must be from one of the pre-defined types of Table 3.2.
- 5047 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(u)$.
- 5048 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5049 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 5050 mulation operator.

5051 Two domains are compatible with each other if values from one domain can be cast to values in
 5052 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 5053 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5054 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 5055 error listed above is returned.

5056 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5057 1. $0 \leq \text{row_index} < \mathbf{nrows}(C)$

5058 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
 5059 is returned.

5060 From the arguments, the internal vectors, mask, and index array used in the computation are
 5061 formed (\leftarrow denotes copy):

5062 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a row of \mathbf{C} as follows:

$$5063 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5064 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

5065 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$.

5066 (b) If `mask \neq GrB_NULL`,

5067 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

5068 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

5069 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

5070 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5071 4. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:

5072 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.

5073 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

5074 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
5075 conditions must hold:

5076 1. $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5077 2. $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$.

5078 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
5079 match error listed above is returned.

5080 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5081 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5082 We are now ready to carry out the assign and any additional associated operations. We describe
5083 this in terms of two intermediate vectors:

- 5084 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 5085 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5086 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$5087 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5088 At this point, if any value of $\tilde{\mathbf{J}}[j]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
5089 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
5090 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
5091 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

5092 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5093 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}}$ is defined as

$$5094 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5095 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
5096 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
5097 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5098 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5099 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$5100 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5101 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5102 where the difference operator refers to set difference.

5103 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5104 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5105 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5106 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5107 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5108 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5109 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5110 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5111 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
5112 result matrix, $\mathbf{C}(\text{row_index}, :)$. This is carried out under control of the mask which acts as a “write
5113 mask”.

5114 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in $\mathbf{C}(\text{row_index}, :)$ on input to this
5115 operation are deleted and the new contents of the column is given by:

$$5116 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5117 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
5118 copied into the column of the final result matrix, $\mathbf{C}(\text{row_index}, :)$, and elements of this column
5119 that fall outside the set indicated by the mask are unchanged:

$$5120 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup$$

$$5121 \quad \{(\text{row_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5122 \quad \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5123 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
5124 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
5125 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.

5130 **4.3.7.5 assign: Constant vector variant**[Scott: NEW CONTENT]

5131 Assign the same value to a specified subset of vector elements. With the use of GrB_ALL, the entire
5132 destination vector can be filled with the constant.

5133 **C Syntax**

```
5134      GrB_Info GrB_assign(GrB_Vector      w,  
5135                          const GrB_Vector mask,  
5136                          const GrB_BinaryOp accum,  
5137                          <type>          val,  
5138                          const GrB_Index *indices,  
5139                          GrB_Index      nindices,  
5140                          const GrB_Descriptor desc);
```

```
5141      GrB_Info GrB_assign(GrB_Vector      w,  
5142                          const GrB_Vector mask,  
5143                          const GrB_BinaryOp accum,  
5144                          const GrB_Scalar s,  
5145                          const GrB_Index *indices,  
5146                          GrB_Index      nindices,  
5147                          const GrB_Descriptor desc);
```

5148 **Parameters**

5149 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5150 that may be accumulated with the result of the assign operation. On output, this
5151 vector holds the results of the operation.

5152 **mask** (IN) An optional “write” mask that controls which results from this operation are
5153 stored into the output vector w. The mask dimensions must match those of the
5154 vector w. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5155 of the mask vector must be of type bool or any of the predefined “built-in” types
5156 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
5157 dimensions of w), GrB_NULL should be specified.

5158 **accum** (IN) An optional binary operator used for accumulating entries into existing w
5159 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5160 specified.

5161 **val** (IN) Scalar value to assign to (a subset of) w.

5162 **s** (IN) Scalar value to assign to (a subset of) w.

5163 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
5164 w that are to be assigned. If all elements of w are to be assigned in order from 0

5165 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode
5166 and return value, this array may be manipulated by the caller after this operation
5167 returns without affecting any deferred computations for this operation. In this
5168 variant, the specific order of the values in the array has no effect on the result.
5169 Unlike other variants, if there are duplicated values in this array the result is still
5170 defined.

5171 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If
5172 `nindices` is zero, the operation becomes a NO-OP.

5173 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
5174 should be specified. Non-default field/value pairs are listed as follows:

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> .

5177 Return Values

5178 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5179 blocking mode, this indicates that the compatibility tests on
5180 dimensions and domains for the input arguments passed suc-
5181 cessfully. Either way, output vector `w` is ready to be used in the
5182 next method of the sequence.

5183 **GrB_PANIC** Unknown internal error.

5184 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
5185 opaque GraphBLAS objects (input or output) is in an invalid
5186 state caused by a previous execution error. Call `GrB_error()` to
5187 access any error messages generated by the implementation.

5188 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5189 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized
5190 by a call to `new` (or `dup` for vector parameters).

5191 **GrB_INDEX_OUT_OF_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-
5192 blocking mode, this can be reported as an execution error.

5193 **GrB_DIMENSION_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less
5194 than `size(w)`.

5225 4. If **accum** is not **GrB_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the
 5226 method, must be compatible with **D_{in2}(accum)** of the accumulation operator.

5227 Two domains are compatible with each other if values from one domain can be cast to values in
 5228 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 5229 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5230 any compatibility rule above is violated, execution of **GrB_assign** ends and the domain mismatch
 5231 error listed above is returned.

5232 From the arguments, the internal vectors, mask and index array used in the computation are formed
 5233 (\leftarrow denotes copy):

- 5234 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5235 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument **mask** as follows:
 - 5236 (a) If **mask** = **GrB_NULL**, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 5237 (b) If **mask** \neq **GrB_NULL**,
 - 5238 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 5239 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$.
 - 5240 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5241 3. Scalar $\tilde{s} \leftarrow s$ (**GrB_Scalar** version only).
- 5242 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument **indices** as follows:
 - 5243 (a) If **indices** = **GrB_ALL**, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$.
 - 5244 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$.

5245 The internal vector and mask are checked for dimension compatibility. The following conditions
 5246 must hold:

- 5247 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5248 2. $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$.

5249 If any compatibility rule above is violated, execution of **GrB_assign** ends and the dimension mis-
 5250 match error listed above is returned.

5251 From this point forward, in **GrB_NONBLOCKING** mode, the method can optionally exit with
 5252 **GrB_SUCCESS** return code and defer any computation and/or execution error codes.

5253 We are now ready to carry out the assign and any additional associated operations. We describe
 5254 this in terms of two intermediate vectors:

- 5255 • $\tilde{\mathbf{t}}$: The vector holding the copies of the scalar, either **val** or \tilde{s} , in their destination locations
 5256 relative to $\tilde{\mathbf{w}}$.

5257 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5258 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows. If a non-opaque scalar \mathbf{val} is provided:

$$5259 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5260 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 1$):

$$5261 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5262 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 0$):

$$5263 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5264 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
 5265 is not in the range $[0, \mathbf{size}(\tilde{\mathbf{w}}))$, the execution of `GrB_assign` ends and the index out-of-bounds
 5266 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a
 5267 sequence-terminating `GrB_wait()` is called. Regardless, the result vector, \mathbf{w} , is invalid from this
 5268 point forward in the sequence.

5269 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

5270 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$5271 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5272 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 5273 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
 5274 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

5275 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5276 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5277 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5278 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5280 where the difference operator refers to set difference. We note that in this case of assigning
 5281 a constant, $\{\tilde{\mathbf{I}}[k], \forall k\}$ and $\mathbf{ind}(\tilde{\mathbf{t}})$ are identical.

5282 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$5283 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5284 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 5285 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$5286 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$5287 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5289 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5290 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5292 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 5293 using what is called a *standard vector mask and replace*. This is carried out under control of the
 5294 mask which acts as a “write mask”.

- 5295 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 5296 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$5297 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5298 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 5299 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 5300 mask are unchanged:

$$5301 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5302 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 5303 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 5304 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 5305 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 5306 sequence.

5307 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5308 Assign the same value to a specified subset of matrix elements. With the use of GrB_ALL, the
 5309 entire destination matrix can be filled with the constant.

5310 C Syntax

```
5311      GrB_Info GrB_assign(GrB_Matrix      C,
5312                        const GrB_Matrix  Mask,
5313                        const GrB_BinaryOp accum,
5314                        <type>            val,
5315                        const GrB_Index    *row_indices,
5316                        GrB_Index          nrows,
5317                        const GrB_Index    *col_indices,
5318                        GrB_Index          ncols,
5319                        const GrB_Descriptor desc);
```

```
5320      GrB_Info GrB_assign(GrB_Matrix      C,
5321                        const GrB_Matrix  Mask,
5322                        const GrB_BinaryOp accum,
5323                        const GrB_Scalar    s,
5324                        const GrB_Index    *row_indices,
5325                        GrB_Index          nrows,
```

```

5326         const GrB_Index      *col_indices,
5327         GrB_Index             ncols,
5328         const GrB_Descriptor  desc);

```

5329 Parameters

5330 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5331 that may be accumulated with the result of the assign operation. On output, the
5332 matrix holds the results of the operation.

5333 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5334 stored into the output matrix **C**. The mask dimensions must match those of the
5335 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5336 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
5337 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5338 dimensions of **C**), **GrB_NULL** should be specified.

5339 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5340 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5341 specified.

5342 **val** (IN) Scalar value to assign to (a subset of) **C**.

5343 **s** (IN) Scalar value to assign to (a subset of) **C**.

5344 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
5345 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
5346 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
5347 this array may be manipulated by the caller after this operation returns without
5348 affecting any deferred computations for this operation. Unlike other variants, if
5349 there are duplicated values in this array the result is still defined.

5350 **nrows** (IN) The number of values in **row_indices** array. Must be in the range: [0, **nrows**(**C**)].
5351 If **nrows** is zero, the operation becomes a NO-OP.

5352 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**
5353 that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** – 1,
5354 then **GrB_ALL** should be specified. Regardless of execution mode and return value,
5355 this array may be manipulated by the caller after this operation returns without
5356 affecting any deferred computations for this operation. Unlike other variants, if
5357 there are duplicated values in this array the result is still defined.

5358 **ncols** (IN) The number of values in **col_indices** array. Must be in the range: [0, **ncols**(**C**)].
5359 If **ncols** is zero, the operation becomes a NO-OP.

5360 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5361 should be specified. Non-default field/value pairs are listed as follows:

5362

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`

5392 or $C(\text{row_indices}, \text{col_indices}) = s$ is performed. If an optional binary accumulation operator
 5393 (\odot) is provided, then either $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot \text{val}$ or
 5394 $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot s$ is performed. More explicitly, if a
 5395 non-opaque value val is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = \text{val}, \text{ or} \\ & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5397 Correspondingly, if a `GrB_Scalar` s is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = s, \text{ or} \\ & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5399 Logically, this operation occurs in three steps:

5400 Setup The internal vectors and mask used in the computation are formed and their domains
 5401 and dimensions are tested for compatibility.

5402 Compute The indicated computations are carried out.

5403 Output The result is written into the output matrix, possibly under control of a mask.

5404 Up to two argument matrices are used in the `GrB_assign` operation:

- 5405 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5406 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)

5407 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain
 5408 compatibility as follows:

- 5409 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 5410 must be from one of the pre-defined types of Table 3.2.
- 5411 2. $\mathbf{D}(C)$ must be compatible with either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5412 method.
- 5413 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5414 of the accumulation operator.
- 5415 4. If `accum` is not `GrB_NULL`, then either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5416 method, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5417 Two domains are compatible with each other if values from one domain can be cast to values in
 5418 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 5419 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5420 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 5421 error listed above is returned.

5422 From the arguments, the internal matrices, index arrays, and mask used in the computation are
 5423 formed (\leftarrow denotes copy):

- 5424 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 5425 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
 - 5426 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 5427 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 5428 (b) If `Mask \neq GrB_NULL`,
 - 5429 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 5430 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 5431 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 5432 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 5433 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 5434 3. Scalar $\tilde{s} \leftarrow s$ (`GrB_Scalar` version only).
- 5435 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
 - 5436 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
 - 5437 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 5438 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
 - 5439 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
 - 5440 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

5441 The internal matrix and mask are checked for dimension compatibility. The following conditions
 5442 must hold:

- 5443 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 5444 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 5445 3. $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$.
- 5446 4. $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$.

5447 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
 5448 match error listed above is returned.

5449 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5450 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5451 We are now ready to carry out the assign and any additional associated operations. We describe
 5452 this in terms of two intermediate matrices:

- 5453 • $\tilde{\mathbf{T}}$: The matrix holding the copies of the scalar, either `val` or \tilde{s} , in their destination locations
 5454 relative to $\tilde{\mathbf{C}}$.
- 5455 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5456 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows. If a non-opaque scalar `val` is provided:

$$5457 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5458 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 1):

$$5459 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5460 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 0):

$$5461 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5462 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
 5463 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
 5464 $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-bounds error listed above is
 5465 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating
 5466 `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from this point forward in the
 5467 sequence.

5468 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 5469 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$5470 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5471 \quad \{(i, j, Z_{ij}) \mid (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5472 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 5473 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
 5474 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

5475 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 5476 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5477 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5478 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \\ 5479$$

5480 where the difference operator refers to set difference. We note that, in this particular case of
 5481 assigning a constant to a matrix, the sets $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\}$ and $\mathbf{ind}(\tilde{\mathbf{T}})$ are identical.

- If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

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

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

$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

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

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

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

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

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

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

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

4.3.8 apply: Apply a function to the elements of an object

Computes the transformation of the values of the elements of a vector or a matrix using a unary function, or a binary function where one argument is bound to a scalar.

4.3.8.1 apply: Vector variant

Computes the transformation of the values of the elements of a vector using a unary function.

5512 C Syntax

```

5513      GrB_Info GrB_apply(GrB_Vector      w,
5514                        const GrB_Vector  mask,
5515                        const GrB_BinaryOp accum,
5516                        const GrB_UnaryOp  op,
5517                        const GrB_Vector  u,
5518                        const GrB_Descriptor desc);

```

5519 Parameters

5520 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5521 that may be accumulated with the result of the apply operation. On output, this
5522 vector holds the results of the operation.

5523 **mask** (IN) An optional “write” mask that controls which results from this operation are
5524 stored into the output vector **w**. The mask dimensions must match those of the
5525 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5526 of the mask vector must be of type **bool** or any of the predefined “built-in” types
5527 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5528 dimensions of **w**), **GrB_NULL** should be specified.

5529 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5530 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5531 specified.

5532 **op** (IN) A unary operator applied to each element of input vector **u**.

5533 **u** (IN) The GraphBLAS vector to which the unary function is applied.

5534 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5535 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 .

5538 Return Values

5539 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5540 blocking mode, this indicates that the compatibility tests on di-
5541 mensions and domains for the input arguments passed successfully.

5542 Either way, output vector w is ready to be used in the next method
5543 of the sequence.

5544 **GrB_PANIC** Unknown internal error.

5545 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
5546 GraphBLAS objects (input or output) is in an invalid state caused
5547 by a previous execution error. Call **GrB_error()** to access any error
5548 messages generated by the implementation.

5549 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5550 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
5551 a call to **new** (or **dup** for vector parameters).

5552 **GrB_DIMENSION_MISMATCH** $mask$, w and/or u dimensions are incompatible.

5553 **GrB_DOMAIN_MISMATCH** The domains of the various vectors are incompatible with the corre-
5554 sponding domains of the accumulation operator or unary function,
5555 or the mask's domain is not compatible with **bool** (in the case where
5556 $desc[GrB_MASK].GrB_STRUCTURE$ is not set).

5557 **Description**

5558 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
5559 GraphBLAS vector: $w = f(u)$; or, if an optional binary accumulation operator (\odot) is provided,
5560 $w = w \odot f(u)$.

5561 Logically, this operation occurs in three steps:

5562 **Setup** The internal vectors and mask used in the computation are formed and their domains
5563 and dimensions are tested for compatibility.

5564 **Compute** The indicated computations are carried out.

5565 **Output** The result is written into the output vector, possibly under control of a mask.

5566 Up to three argument vectors are used in this **GrB_apply** operation:

- 5567 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5568 2. $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$ (optional)
- 5569 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5570 The argument vectors, unary operator and the accumulation operator (if provided) are tested for
5571 domain compatibility as follows:

- 5572 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5573 must be from one of the pre-defined types of Table 3.2.
- 5574 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5575 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5576 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5577 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5578 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in}(\text{op})$.

5579 Two domains are compatible with each other if values from one domain can be cast to values in
5580 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5581 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5582 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
5583 error listed above is returned.

5584 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
5585 denotes copy):

- 5586 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5587 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 5588 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 5589 (b) If `mask` \neq `GrB_NULL`,
 - 5590 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 5591 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 5592 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5593 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5594 The internal vectors and masks are checked for dimension compatibility. The following conditions
5595 must hold:

- 5596 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5597 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

5598 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5599 error listed above is returned.

5600 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5601 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5602 We are now ready to carry out the apply and any additional associated operations. We describe
5603 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.

5664 Return Values

5665	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
5666		blocking mode, this indicates that the compatibility tests on
5667		dimensions and domains for the input arguments passed suc-
5668		cessfully. Either way, output matrix C is ready to be used in the
5669		next method of the sequence.
5670	GrB_PANIC	Unknown internal error.
5671	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
5672		opaque GraphBLAS objects (input or output) is in an invalid
5673		state caused by a previous execution error. Call GrB_error() to
5674		access any error messages generated by the implementation.
5675	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
5676	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
5677		by a call to new (or Matrix_dup for matrix parameters).
5678	GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, nrows \neq nrows (C), or
5679		ncols \neq ncols (C).
5680	GrB_DOMAIN_MISMATCH	The domains of the various matrices are incompatible with the
5681		corresponding domains of the accumulation operator or unary
5682		function, or the mask's domain is not compatible with bool (in
5683		the case where desc [GrB_MASK]. GrB_STRUCTURE is not set).

5684 Description

5685 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
 5686 GraphBLAS matrix: $C = f(A)$; or, if an optional binary accumulation operator (\odot) is provided,
 5687 $C = C \odot f(A)$.

5688 Logically, this operation occurs in three steps:

5689 **Setup** The internal matrices and mask used in the computation are formed and their domains
 5690 and dimensions are tested for compatibility.

5691 **Compute** The indicated computations are carried out.

5692 **Output** The result is written into the output matrix, possibly under control of a mask.

5693 Up to three argument matrices are used in the **GrB_apply** operation:

- 5694 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5695 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)

5696 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

5697 The argument matrices, unary operator and the accumulation operator (if provided) are tested for
5698 domain compatibility as follows:

- 5699 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\mathbf{Mask})$
5700 must be from one of the pre-defined types of Table 3.2.
- 5701 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the unary operator.
- 5702 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
5703 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the unary operator must be compatible with
5704 $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accumulation operator.
- 5705 4. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in}(\mathbf{op})$ of the unary operator.

5706 Two domains are compatible with each other if values from one domain can be cast to values in
5707 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
5708 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5709 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
5710 error listed above is returned.

5711 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
5712 are formed (\leftarrow denotes copy):

- 5713 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 5714 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument **Mask** as follows:
 - 5715 (a) If **Mask** = **GrB_NULL**, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
5716 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 5717 (b) If **Mask** \neq **GrB_NULL**,
 - 5718 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
5719 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 5720 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
5721 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 5722 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 5723 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

5724 The internal matrices and mask are checked for dimension compatibility. The following conditions
5725 must hold:

- 5726 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 5727 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 5728 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

5729 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

5730 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5731 error listed above is returned.

5732 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5733 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5734 We are now ready to carry out the apply and any additional associated operations. We describe
5735 this in terms of two intermediate matrices:

- 5736 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the unary operator to the input matrix $\tilde{\mathbf{A}}$.
- 5737 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5738 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$5739 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5740 where $f = \mathbf{f}(\mathbf{op})$.

5741 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 5742 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 5743 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5744 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5745 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5746 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5747 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5748 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5750 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5752 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5753 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5754 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5755 mask which acts as a “write mask”.

- 5756 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5757 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5758 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 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

```

5776 // bind-first + scalar value
5777 GrB_Info GrB_apply(GrB_Vector          w,
5778                  const GrB_Vector      mask,
5779                  const GrB_BinaryOp    accum,
5780                  const GrB_BinaryOp    op,
5781                  <type>                val,
5782                  const GrB_Vector      u,
5783                  const GrB_Descriptor  desc);

5784 // bind-first + GraphBLAS scalar
5785 GrB_Info GrB_apply(GrB_Vector          w,
5786                  const GrB_Vector      mask,
5787                  const GrB_BinaryOp    accum,
5788                  const GrB_BinaryOp    op,
5789                  const GrB_Scalar      s,
5790                  const GrB_Vector      u,
5791                  const GrB_Descriptor  desc);

5792 // bind-second + scalar value
5793 GrB_Info GrB_apply(GrB_Vector          w,
5794                  const GrB_Vector      mask,
```

```

5795         const GrB_BinaryOp      accum,
5796         const GrB_BinaryOp      op,
5797         const GrB_Vector        u,
5798         <type>                  val,
5799         const GrB_Descriptor    desc);

5800 // bind-second + GraphBLAS scalar
5801 GrB_Info GrB_apply(GrB_Vector      w,
5802                   const GrB_Vector mask,
5803                   const GrB_BinaryOp accum,
5804                   const GrB_BinaryOp op,
5805                   const GrB_Vector u,
5806                   const GrB_Scalar s,
5807                   const GrB_Descriptor desc);

```

5808 Parameters

5809 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5810 that may be accumulated with the result of the apply operation. On output, this
5811 vector holds the results of the operation.

5812 **mask** (IN) An optional “write” mask that controls which results from this operation are
5813 stored into the output vector **w**. The mask dimensions must match those of the
5814 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5815 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
5816 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
5817 dimensions of **w**), **GrB_NULL** should be specified.

5818 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5819 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5820 specified.

5821 **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar
5822 value, **val**.

5823 **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as
5824 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
5825 argument in the *bind-second* variant.

5826 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
5827 argument in the *bind-first* variant, or the right-hand (second) argument in the
5828 *bind-second* variant.

5829 **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand
5830 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
5831 the *bind-second* variant. It must not be empty.

5832 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
5833 should be specified. Non-default field/value pairs are listed as follows:

5834

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 .

5835

5836 Return Values

5837 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
5838 blocking mode, this indicates that the compatibility tests on di-
5839 mensions and domains for the input arguments passed successfully.
5840 Either way, output vector **w** is ready to be used in the next method
5841 of the sequence.

5842 GrB_PANIC Unknown internal error.

5843 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
5844 GraphBLAS objects (input or output) is in an invalid state caused
5845 by a previous execution error. Call GrB_error() to access any error
5846 messages generated by the implementation.

5847 GrB_OUT_OF_MEMORY Not enough memory available for operation.

5848 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
5849 a call to new (or dup for vector parameters).

5850 GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

5851 GrB_DOMAIN_MISMATCH The domains of the various vectors and scalar are incompatible with
5852 the corresponding domains of the binary operator or accumulation
5853 operator, or the mask's domain is not compatible with bool (in the
5854 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

5855 GrB_EMPTY_OBJECT The GrB_Scalar **s** used in the call is empty (**nvals(s) = 0**) and
5856 therefore a value cannot be passed to the binary operator.

5857 Description

5858 This variant of GrB_apply computes the result of applying a binary operator to the elements of a
5859 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5860 bind-first: $w = f(\text{val}, u)$ or $w = f(s, u)$

5861 bind-second: $w = f(u, \text{val})$ or $w = f(u, s)$,

5862 or if an optional binary accumulation operator (\odot) is provided:

5863 bind-first: $w = w \odot f(\text{val}, u)$ or $w = w \odot f(s, u)$

5864 bind-second: $w = w \odot f(u, \text{val})$ or $w = w \odot f(u, s)$.

5865 Logically, this operation occurs in three steps:

5866 **Setup** The internal vectors and mask used in the computation are formed and their domains
5867 and dimensions are tested for compatibility.

5868 **Compute** The indicated computations are carried out.

5869 **Output** The result is written into the output vector, possibly under control of a mask.

5870 Up to three argument vectors are used in this GrB_apply operation:

- 5871 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
5872 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
5873 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5874 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are
5875 tested for domain compatibility as follows:

- 5876 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
5877 must be from one of the pre-defined types of Table 3.2.
- 5878 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.
- 5879 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5880 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
5881 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5882 4. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.
- 5883 5. If bind-first:
- 5884 (a) $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.
- 5885 (b) If the non-opaque scalar **val** is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
5886 of the binary operator.
- 5887 (c) If the GrB_Scalar **s** is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
5888 binary operator.

- 5889 6. If bind-second:
- 5890 (a) $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the binary operator.
- 5891 (b) If the non-opaque scalar \mathbf{val} is provided, then $\mathbf{D}(\mathbf{val})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
- 5892 of the binary operator.
- 5893 (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
- 5894 binary operator.

5895 Two domains are compatible with each other if values from one domain can be cast to values in

5896 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5897 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5898 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5899 error listed above is returned.

5900 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow

5901 denotes copy):

- 5902 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5903 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 5904 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 5905 (b) If `mask \neq GrB_NULL`,
- 5906 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 5907 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 5908 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5909 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 5910 4. Scalar $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$ (GraphBLAS scalar case).

5911 The internal vectors and masks are checked for dimension compatibility. The following conditions

5912 must hold:

- 5913 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5914 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

5915 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5916 error listed above is returned.

5917 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5918 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5919 If an empty `GrB_Scalar` $\tilde{\mathbf{s}}$ is provided ($\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

5920 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

5921 `val` with the same domain as $\tilde{\mathbf{s}}$ and set `val = val($\tilde{\mathbf{s}}$)`.

5922 We are now ready to carry out the apply and any additional associated operations. We describe

5923 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.

5957 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

5958 Computes the transformation of the values of the stored elements of a matrix using a binary
5959 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the
5960 first argument to the binary operator and stored elements of the matrix are passed as the second
5961 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument
5962 and the specified scalar value is passed as the second argument. The scalar can be passed either as
5963 a non-opaque variable or as a GrB_Scalar object.

5964 C Syntax

```
5965 // bind-first + scalar value
5966 GrB_Info GrB_apply(GrB_Matrix      C,
5967                   const GrB_Matrix Mask,
5968                   const GrB_BinaryOp accum,
5969                   const GrB_BinaryOp op,
5970                   <type>           val,
5971                   const GrB_Matrix A,
5972                   const GrB_Descriptor desc);
```

```
5973 // bind-first + GraphBLAS scalar
5974 GrB_Info GrB_apply(GrB_Matrix      C,
5975                   const GrB_Matrix Mask,
5976                   const GrB_BinaryOp accum,
5977                   const GrB_BinaryOp op,
5978                   const GrB_Scalar s,
5979                   const GrB_Matrix A,
5980                   const GrB_Descriptor desc);
```

```
5981 // bind-second + scalar value
5982 GrB_Info GrB_apply(GrB_Matrix      C,
5983                   const GrB_Matrix Mask,
5984                   const GrB_BinaryOp accum,
5985                   const GrB_BinaryOp op,
5986                   const GrB_Matrix A,
5987                   <type>           val,
5988                   const GrB_Descriptor desc);
```

```
5989 // bind-second + GraphBLAS scalar
5990 GrB_Info GrB_apply(GrB_Matrix      C,
5991                   const GrB_Matrix Mask,
5992                   const GrB_BinaryOp accum,
5993                   const GrB_BinaryOp op,
5994                   const GrB_Matrix A,
```



```

5995         const GrB_Scalar      s,
5996         const GrB_Descriptor desc);

```

5997 Parameters

5998 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5999 that may be accumulated with the result of the apply operation. On output, the
6000 matrix holds the results of the operation.

6001 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6002 stored into the output matrix C. The mask dimensions must match those of the
6003 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
6004 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
6005 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the
6006 dimensions of C), `GrB_NULL` should be specified.

6007 **accum** (IN) An optional binary operator used for accumulating entries into existing C
6008 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
6009 specified.

6010 **op** (IN) A binary operator applied to each element of input matrix, A, with the element
6011 of the input matrix used as the left-hand argument, and the scalar value, `val`, used
6012 as the right-hand argument.

6013 **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as
6014 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
6015 argument in the *bind-second* variant.

6016 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
6017 argument in the *bind-first* variant, or the right-hand (second) argument in the
6018 *bind-second* variant.

6019 **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand
6020 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
6021 the *bind-second* variant. It must not be empty.

6022 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
6023 should be specified. Non-default field/value pairs are listed as follows:
6024

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.

6052 Description

6053 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a
 6054 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6055 bind-first: $C = f(\text{val}, A)$ or $C = f(s, A)$

6056 bind-second: $C = f(A, \text{val})$ or $C = f(A, s)$,

6057 or if an optional binary accumulation operator (\odot) is provided:

6058 bind-first: $C = C \odot f(\text{val}, A)$ or $C = C \odot f(s, A)$

6059 bind-second: $C = C \odot f(A, \text{val})$ or $C = C \odot f(A, s)$.

6060 Logically, this operation occurs in three steps:

6061 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6062 and dimensions are tested for compatibility.

6063 **Compute** The indicated computations are carried out.

6064 **Output** The result is written into the output matrix, possibly under control of a mask.

6065 Up to three argument matrices are used in the `GrB_apply` operation:

6066 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

6067 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)

6068 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6069 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are
 6070 tested for domain compatibility as follows:

6071 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 6072 must be from one of the pre-defined types of Table 3.2.

6073 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.

6074 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6075 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
 6076 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

6077 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

6078 5. If bind-first:

6079 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

6080 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
 6081 of the binary operator.

6082 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
 6083 binary operator.

6084 6. If `bind-second`:

6085 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

6086 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$
 6087 of the binary operator.

6088 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the
 6089 binary operator.

6090 Two domains are compatible with each other if values from one domain can be cast to values in
 6091 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6092 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6093 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
 6094 error listed above is returned.

6095 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
 6096 are formed (\leftarrow denotes copy):

6097 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.

6098 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:

6099 (a) If `Mask` = `GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 6100 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.

6101 (b) If `Mask` \neq `GrB_NULL`,

6102 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
 6103 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,

6104 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
 6105 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.

6106 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.

6107 3. Matrix $\tilde{\mathbf{A}}$ is computed from argument `A` as follows:

6108 `bind-first:` $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? A^T : A$

6109 `bind-second:` $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$

6110 4. Scalar $\tilde{s} \leftarrow s$ (`GraphBLAS` scalar case).

6111 The internal matrices and mask are checked for dimension compatibility. The following conditions
 6112 must hold:

6113 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.

6114 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.

6115 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6116 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6117 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6118 error listed above is returned.

6119 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6120 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6121 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

6122 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6123 `val` with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6124 We are now ready to carry out the apply and any additional associated operations. We describe
6125 this in terms of two intermediate matrices:

- 6126 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the binary operator to the input matrix $\tilde{\mathbf{A}}$.
- 6127 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6128 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as one of the following:

6129 bind-first: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6130 bind-second: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6131 where $f = \mathbf{f}(\mathbf{op})$.

6132 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6133 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6134 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6135 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6136 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6137 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6138 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6139 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6140 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6143 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6144 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6145 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6146 mask which acts as a “write mask”.

- 6147 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
6148 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6149 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6150 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6151 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6152 mask are unchanged:

$$6153 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6154 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
6155 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
6156 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
6157 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6158 sequence.

6159 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6160 Computes the transformation of the values of the stored elements of a vector using an index unary
6161 operator that is a function of the stored value, its location indices, and an user provided scalar
6162 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6163 C Syntax

```
6164      GrB_Info GrB_apply(GrB_Vector      w,
6165                        const GrB_Vector  mask,
6166                        const GrB_BinaryOp accum,
6167                        const GrB_IndexUnaryOp op,
6168                        const GrB_Vector  u,
6169                        <type>           val,
6170                        const GrB_Descriptor desc);
```

```
6171      GrB_Info GrB_apply(GrB_Vector      w,
6172                        const GrB_Vector  mask,
6173                        const GrB_BinaryOp accum,
6174                        const GrB_IndexUnaryOp op,
6175                        const GrB_Vector  u,
6176                        const GrB_Scalar  s,
6177                        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.

6209 GrB_PANIC Unknown internal error.

6210 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6211 opaque GraphBLAS objects (input or output) is in an invalid
6212 state caused by a previous execution error. Call GrB_error() to
6213 access any error messages generated by the implementation.

6214 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6215 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6216 by a call to new (or another constructor).

6217 GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

6218 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
6219 responding domains of the accumulation operator or index unary
6220 operator, or the mask's domain is not compatible with bool (in
6221 the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

6222 GrB_EMPTY_OBJECT The GrB_Scalar s used in the call is empty ($\mathbf{nvals}(s) = 0$) and
6223 therefore a value cannot be passed to the index unary operator.

6224 Description

6225 This variant of GrB_apply computes the result of applying an index unary operator to the elements
6226 of a GraphBLAS vector each composed with the element's index and a scalar constant, val or s:

$$6227 \quad \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \text{val}) \text{ or } \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}),$$

6228 or if an optional binary accumulation operator (\odot) is provided:

$$6229 \quad \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \text{val}) \text{ or } \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}).$$

6230 Logically, this operation occurs in three steps:

6231 **Setup** The internal vectors and mask used in the computation are formed and their domains
6232 and dimensions are tested for compatibility.

6233 **Compute** The indicated computations are carried out.

6234 **Output** The result is written into the output vector, possibly under control of a mask.

6235 Up to three argument vectors are used in this GrB_apply operation:

- 6236 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6237 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)

6238 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6239 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6240 are tested for domain compatibility as follows:

- 6241 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
6242 must be from one of the pre-defined types of Table 3.2.
- 6243 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the index unary operator.
- 6244 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6245 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be compatible
6246 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6247 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6248 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
6249 the index unary operator.
- 6250 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
6251 unary operator.

6252 Two domains are compatible with each other if values from one domain can be cast to values in
6253 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6254 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6255 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
6256 error listed above is returned.

6257 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6258 denotes copy):

- 6259 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6260 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 6261 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 6262 (b) If `mask` \neq `GrB_NULL`,
 - 6263 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,
 - 6264 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 6265 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6266 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 6267 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6268 The internal vectors and masks are checked for dimension compatibility. The following conditions
6269 must hold:

6270 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6271 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

6272 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6273 error listed above is returned.

6274 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6275 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6276 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6277 If a non-empty `GrB_Scalar`, \tilde{s} , is provided ($\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable `val`
6278 with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6279 We are now ready to carry out the apply and any additional associated operations. We describe
6280 this in terms of two intermediate vectors:

- 6281 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6282 $\tilde{\mathbf{u}}$.
- 6283 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6284 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6285 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6286 where $f_i = \mathbf{f}(\text{op})$.

6287 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6288 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 6289 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6290 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6291 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6292 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6293 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6294 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6295 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

6296 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6299 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
6300 using what is called a *standard vector mask and replace*. This is carried out under control of the
6301 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.

6314 4.3.8.6 apply: Matrix index unary operator variant[Scott: NEW CONTENT]

6315 Computes the transformation of the values of the stored elements of a matrix using an index unary
6316 operator that is a function of the stored value, its location indices, and an user provided scalar
6317 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6318 C Syntax

```
6319     GrB_Info GrB_apply(GrB_Matrix      C,
6320                       const GrB_Matrix Mask,
6321                       const GrB_BinaryOp accum,
6322                       const GrB_IndexUnaryOp op,
6323                       const GrB_Matrix A,
6324                       <type>          val,
6325                       const GrB_Descriptor desc);
```

```
6326     GrB_Info GrB_apply(GrB_Matrix      C,
6327                       const GrB_Matrix Mask,
6328                       const GrB_BinaryOp accum,
6329                       const GrB_IndexUnaryOp op,
6330                       const GrB_Matrix A,
6331                       const GrB_Scalar s,
6332                       const GrB_Descriptor desc);
```

6333 Parameters

6334 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6335 that may be accumulated with the result of the apply operation. On output, the
6336 matrix holds the results of the operation.

6337 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6338 stored into the output matrix **C**. The mask dimensions must match those of the
6339 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6340 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
6341 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6342 dimensions of **C**), **GrB_NULL** should be specified.

6343 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
6344 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6345 specified.

6346 **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
6347 to each element stored in the input matrix, **A**. It is a function of the stored element’s
6348 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6349 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6350 ator.

6351 **val** (IN) An additional scalar value that is passed to the index unary operator.

6352 **s** (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6353 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6354 should be specified. Non-default field/value pairs are listed as follows:

6355

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.

6356

6357 Return Values

6358 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
6359 blocking mode, this indicates that the compatibility tests on di-
6360 mensions and domains for the input arguments passed successfully.
6361 Either way, output matrix **C** is ready to be used in the next method
6362 of the sequence.

6363 **GrB_PANIC** Unknown internal error.

6364 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
6365 GraphBLAS objects (input or output) is in an invalid state caused

6366 by a previous execution error. Call `GrB_error()` to access any error
 6367 messages generated by the implementation.

6368 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

6369 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 6370 a call to `new` (or another constructor).

6371 **GrB_DIMENSION_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6372 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
 6373 corresponding domains of the accumulation operator or index unary
 6374 operator, or the mask's domain is not compatible with `bool` (in the
 6375 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6376 **GrB_EMPTY_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and
 6377 therefore a value cannot be passed to the index unary operator.

6378 Description

6379 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements
 6380 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar
 6381 constant, `val` or `s`:

$$6382 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6383 or if an optional binary accumulation operator (\odot) is provided:

$$6384 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6385 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
 6386 indices, respectively.

6387 Logically, this operation occurs in three steps:

6388 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6389 and dimensions are tested for compatibility.

6390 **Compute** The indicated computations are carried out.

6391 **Output** The result is written into the output matrix, possibly under control of a mask.

6392 Up to three argument matrices are used in the `GrB_apply` operation:

- 6393 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6394 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)

6395 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6396 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6397 are tested for domain compatibility as follows:

- 6398 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\mathbf{Mask})$
6399 must be from one of the pre-defined types of Table 3.2.
- 6400 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator.
- 6401 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6402 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be compatible
6403 with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accumulation operator.
- 6404 4. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
- 6405 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
6406 the index unary operator.
- 6407 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
6408 unary operator.

6409 Two domains are compatible with each other if values from one domain can be cast to values in
6410 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6411 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6412 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
6413 error listed above is returned.

6414 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation
6415 are formed (\leftarrow denotes copy):

- 6416 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 6417 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument **Mask** as follows:
 - 6418 (a) If **Mask** = **GrB_NULL**, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
6419 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 6420 (b) If **Mask** \neq **GrB_NULL**,
 - 6421 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
6422 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 6423 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
6424 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 6425 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 6426 3. Matrix $\tilde{\mathbf{A}}$ is computed from argument **A** as follows:

$$6427 \quad \tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$$
- 6428 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6429 The internal matrices and mask are checked for dimension compatibility. The following conditions
6430 must hold:

- 6431 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 6432 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 6433 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 6434 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6435 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6436 error listed above is returned.

6437 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6438 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6439 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6440 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6441 `val` with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6442 We are now ready to carry out the apply and any additional associated operations. We describe
6443 this in terms of two intermediate matrices:

- 6444 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6445 $\tilde{\mathbf{A}}$.
- 6446 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6447 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6448 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6449 where $f_i = \mathbf{f}(\mathbf{op})$.

6450 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6451 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6452 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6453 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6454 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6455 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 6456 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6457 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6458 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6459 \end{aligned}$$

6460 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6462 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 6463 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 6464 mask which acts as a “write mask”.

- 6465 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 6466 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6467 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6468 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 6469 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 6470 mask are unchanged:

$$6471 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6472 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 6473 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 6474 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 6475 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 6476 sequence.

6477 4.3.9 select:

6478 Apply a select operator to the stored elements of an object to determine whether or not to keep
 6479 them.

6480 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6481 Apply a select operator (an index unary operator) to the elements of a vector.

6482 C Syntax

```
6483 // scalar value variant
6484 GrB_Info GrB_select(GrB_Vector          w,
6485                    const GrB_Vector      mask,
6486                    const GrB_BinaryOp    accum,
6487                    const GrB_IndexUnaryOp op,
6488                    const GrB_Vector      u,
6489                    <type>                val,
6490                    const GrB_Descriptor   desc);
6491
6492 // GraphBLAS scalar variant
6493 GrB_Info GrB_select(GrB_Vector          w,
6494                    const GrB_Vector      mask,
```



```

6495         const GrB_BinaryOp      accum,
6496         const GrB_IndexUnaryOp  op,
6497         const GrB_Vector        u,
6498         const GrB_Scalar        s,
6499         const GrB_Descriptor    desc);
6500

```

6501 Parameters

6502 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6503 that may be accumulated with the result of the select operation. On output, this
6504 vector holds the results of the operation.

6505 **mask** (IN) An optional “write” mask that controls which results from this operation are
6506 stored into the output vector **w**. The mask dimensions must match those of the
6507 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6508 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6509 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6510 dimensions of **w**), **GrB_NULL** should be specified.

6511 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6512 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6513 specified.

6514 **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
6515 to each element stored in the input vector, **u**. It is a function of the stored element’s
6516 value, its location index, and a user supplied scalar value (either **s** or **val**).

6517 **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-
6518 ator.

6519 **val** (IN) An additional scalar value that is passed to the index unary operator.

6520 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6521 not be empty.

6522 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6523 should be specified. Non-default field/value pairs are listed as follows:

6524

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 .

6525

6526 Return Values

6527	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
6528		blocking mode, this indicates that the compatibility tests on di-
6529		mensions and domains for the input arguments passed success-
6530		fully. Either way, output vector w is ready to be used in the next
6531		method of the sequence.
6532	GrB_PANIC	Unknown internal error.
6533	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
6534		opaque GraphBLAS objects (input or output) is in an invalid
6535		state caused by a previous execution error. Call GrB_error() to
6536		access any error messages generated by the implementation.
6537	GrB_OUT_OF_MEMORY	Not enough memory available for operation.
6538	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
6539		by a call to one of its constructors.
6540	GrB_DIMENSION_MISMATCH	mask , w and/or u dimensions are incompatible.
6541	GrB_DOMAIN_MISMATCH	The domains of the various vectors are incompatible with the cor-
6542		responding domains of the accumulation operator or index unary
6543		operator, or the mask's domain is not compatible with bool (in
6544		the case where desc[GrB_MASK].GrB_STRUCTURE is not set).
6545	GrB_EMPTY_OBJECT	The GrB_Scalar s used in the call is empty (nvals(s) = 0) and
6546		therefore a value cannot be passed to the index unary operator.

6547 Description

6548 This variant of **GrB_select** computes the result of applying a index unary operator to select the
6549 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored
6550 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding
6551 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.
6552 This acts like a functional mask on the input vector as follows:

$$6553 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6554 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6555 Correspondingly, if a **GrB_Scalar s**, is provided:

$$6556 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6557 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6558 Logically, this operation occurs in three steps:

6559 **Setup** The internal vectors and mask used in the computation are formed and their domains
6560 and dimensions are tested for compatibility.

6561 **Compute** The indicated computations are carried out.

6562 **Output** The result is written into the output vector, possibly under control of a mask.

6563 Up to three argument vectors are used in this GrB_select operation:

- 6564 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
6565 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
6566 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6567 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6568 are tested for domain compatibility as follows:

- 6569 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\mathbf{mask})$
6570 must be from one of the pre-defined types of Table 3.2.
6571 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
6572 3. If **accum** is not GrB_NULL, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6573 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
6574 mulation operator.
6575 4. $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
6576 i.e., castable to **bool**.
6577 5. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
6578 6. $\mathbf{D}(\mathbf{val})$ or $\mathbf{D}(\mathbf{s})$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
6579 of the index unary operator.

6580 Two domains are compatible with each other if values from one domain can be cast to values in
6581 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
6582 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6583 any compatibility rule above is violated, execution of GrB_select ends and the domain mismatch
6584 error listed above is returned.

6585 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6586 denotes copy):

- 6587 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
6588 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument **mask** as follows:

6589 (a) If $\text{mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
6590 (b) If $\text{mask} \neq \text{GrB_NULL}$,
6591 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
6592 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
6593 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
6594 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
6595 4. Scalar $\widetilde{s} \leftarrow s$ (GrB_Scalar version only).

6596 The internal vectors and masks are checked for dimension compatibility. The following conditions
6597 must hold:

- 6598 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6599 2. $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$.

6600 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch
6601 error listed above is returned.

6602 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6603 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6604 If an empty `GrB_Scalar` \widetilde{s} is provided (i.e., $\text{nvals}(\widetilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6605 If a non-empty `GrB_Scalar`, \widetilde{s} , is provided (i.e., $\text{nvals}(\widetilde{s}) = 1$), we then create an internal variable
6606 `val` with the same domain as \widetilde{s} and set $\text{val} = \text{val}(\widetilde{s})$.

6607 We are now ready to carry out the `select` and any additional associated operations. We describe
6608 this in terms of two intermediate vectors:

- 6609 • $\widetilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6610 $\widetilde{\mathbf{u}}$.
- 6611 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6612 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$6613 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6614 where $f_i = \mathbf{f}(\text{op})$.

6615 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6616 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 6617 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$6618 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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);
```

```

6654 // GraphBLAS scalar variant
6655 GrB_Info GrB_select(GrB_Matrix          C,
6656                    const GrB_Matrix     Mask,
6657                    const GrB_BinaryOp    accum,
6658                    const GrB_IndexUnaryOp op,
6659                    const GrB_Matrix      A,
6660                    const GrB_Scalar      s,
6661                    const GrB_Descriptor   desc);

```

6662 Parameters

6663 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6664 that may be accumulated with the result of the select operation. On output, the
6665 matrix holds the results of the operation.

6666 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6667 stored into the output matrix **C**. The mask dimensions must match those of the
6668 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6669 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
6670 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6671 dimensions of **C**), **GrB_NULL** should be specified.

6672 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
6673 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6674 specified.

6675 **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
6676 to each element stored in the input matrix, **A**. It is a function of the stored element’s
6677 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6678 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6679 ator.

6680 **val** (IN) An additional scalar value that is passed to the index unary operator.

6681 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6682 not be empty.

6683 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6684 should be specified. Non-default field/value pairs are listed as follows:

6685

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:

6715 $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$, or

6716 $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$.

6717 Correspondingly, if a GrB_Scalar, s , is provided:

6718 $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$, or

6719 $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$.

6720 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
6721 indices, respectively.

6722 Logically, this operation occurs in three steps:

6723 **Setup** The internal matrices and mask used in the computation are formed and their domains
6724 and dimensions are tested for compatibility.

6725 **Compute** The indicated computations are carried out.

6726 **Output** The result is written into the output matrix, possibly under control of a mask.

6727 Up to three argument matrices are used in the GrB_select operation:

- 6728 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6729 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)
- 6730 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6731 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6732 are tested for domain compatibility as follows:

- 6733 1. If **Mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
6734 must be from one of the pre-defined types of Table 3.2.
- 6735 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 6736 3. If **accum** is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6737 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
6738 mulation operator.
- 6739 4. $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be from one of the pre-defined types of Table 3.2;
6740 i.e., castable to **bool**.
- 6741 5. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6742 6. $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\text{op})$
6743 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:

- 6778 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6779 $\tilde{\mathbf{A}}$.
- 6780 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6781 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6782 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6783 where $f_i = \mathbf{f}(\text{op})$.

6784 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6785 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6786 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6787 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6788 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6789 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6790 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6791 \\ 6792 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6793 \\ 6794 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6795 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6796 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6797 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6798 mask which acts as a “write mask”.

- 6799 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
6800 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6801 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6802 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6803 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6804 mask are unchanged:

$$6805 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6806 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6807 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6808 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
6809 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6810 sequence.

6811 4.3.10 reduce: Perform a reduction across the elements of an object

6812 Computes the reduction of the values of the elements of a vector or matrix.

6813 4.3.10.1 reduce: Standard matrix to vector variant

6814 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns
6815 is desired, the input matrix should be transposed using the descriptor.

6816 C Syntax

```
6817     GrB_Info GrB_reduce(GrB_Vector      w,  
6818                        const GrB_Vector mask,  
6819                        const GrB_BinaryOp accum,  
6820                        const GrB_Monoid op,  
6821                        const GrB_Matrix A,  
6822                        const GrB_Descriptor desc);  
6823  
6824     GrB_Info GrB_reduce(GrB_Vector      w,  
6825                        const GrB_Vector mask,  
6826                        const GrB_BinaryOp accum,  
6827                        const GrB_BinaryOp op,  
6828                        const GrB_Matrix A,  
6829                        const GrB_Descriptor desc);
```

6830 Parameters

6831 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6832 that may be accumulated with the result of the reduction operation. On output,
6833 this vector holds the results of the operation.

6834 **mask** (IN) An optional “write” mask that controls which results from this operation are
6835 stored into the output vector **w**. The mask dimensions must match those of the
6836 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6837 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6838 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the
6839 dimensions of **w**), **GrB_NULL** should be specified.

6840 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6841 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6842 specified.

6843 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.
6844 Depending on which type is passed, the following defines the binary operator with
6845 one domain, $F_b = \langle D, D, D, \oplus \rangle$, that is used:

6846 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 6847 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
 6848 monoid is ignored.

6849 If op is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in
 6850 both cases $\odot(\text{op})$ must be commutative and associative. Otherwise, the outcome
 6851 of the operation is undefined.

6852 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

6853 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 6854 should be specified. Non-default field/value pairs are listed as follows:
 6855

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.

6857 Return Values

6858 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 6859 blocking mode, this indicates that the compatibility tests on di-
 6860 mensions and domains for the input arguments passed successfully.
 6861 Either way, output vector w is ready to be used in the next method
 6862 of the sequence.

6863 **GrB_PANIC** Unknown internal error.

6864 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 6865 GraphBLAS objects (input or output) is in an invalid state caused
 6866 by a previous execution error. Call `GrB_error()` to access any error
 6867 messages generated by the implementation.

6868 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

6869 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 6870 a call to `new` (or `dup` for vector parameters).

6871 **GrB_DIMENSION_MISMATCH** mask, w and/or u dimensions are incompatible.

6872 **GrB_DOMAIN_MISMATCH** Either the domains of the various vectors and matrices are incom-
 6873 patible with the corresponding domains of the accumulation oper-
 6874 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).

6878 Description

6879 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows
 6880 of an input matrix: $w(i) = \bigoplus A(i, :) \forall i$; or, if an optional binary accumulation operator is provided,
 6881 $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$, where $\bigoplus = \odot(F_b)$ and $\odot = \odot(\text{accum})$.

6882 Logically, this operation occurs in three steps:

6883 **Setup** The internal vector, matrix and mask used in the computation are formed and their
 6884 domains and dimensions are tested for compatibility.

6885 **Compute** The indicated computations are carried out.

6886 **Output** The result is written into the output vector, possibly under control of a mask.

6887 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6888 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6889 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 6890 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6891 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested
 6892 for domain compatibility as follows:

- 6893 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 6894 must be from one of the pre-defined types of Table 3.2.
- 6895 2. $\mathbf{D}(w)$ must be compatible with the domain of the reduction binary operator, $\mathbf{D}(F_b)$.
- 6896 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6897 of the accumulation operator and $\mathbf{D}(F_b)$, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 6898 mulation operator.
- 6899 4. $\mathbf{D}(A)$ must be compatible with the domain of the binary reduction operator, $\mathbf{D}(F_b)$.

6900 Two domains are compatible with each other if values from one domain can be cast to values in
 6901 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 6902 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6903 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 6904 error listed above is returned.

6905 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 6906 denotes copy):

- 6907 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6908 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 6909 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 6910 (b) If `mask \neq GrB_NULL`,
- 6911 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 6912 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 6913 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6914 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

6915 The internal vectors and masks are checked for dimension compatibility. The following conditions
 6916 must hold:

- 6917 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6918 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6919 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-
 6920 match error listed above is returned.

6921 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 6922 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6923 We carry out the reduce and any additional associated operations. We describe this in terms of
 6924 two intermediate vectors:

- 6925 • $\tilde{\mathbf{t}}$: The vector holding the result from reducing along the rows of input matrix $\tilde{\mathbf{A}}$.
- 6926 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6927 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6928 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6929 The value of each of its elements is computed by

$$6930 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6931 where $\bigoplus = \odot(F_b)$.

6932 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6933 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

6934 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6935 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6936 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6937 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6938 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 6939 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6940 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6941 \end{aligned}$$

6942 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6944 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
6945 using what is called a *standard vector mask and replace*. This is carried out under control of the
6946 mask which acts as a “write mask”.

6947 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
6948 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$6949 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6950 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
6951 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
6952 mask are unchanged:

$$6953 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6954 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6955 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6956 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
6957 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6958 sequence.

6959 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

6960 Reduce all stored values into a single scalar.

6961 C Syntax

```
6962 // scalar value + monoid (only)
6963 GrB_Info GrB_reduce(<type>          *val,
6964                    const GrB_BinaryOp accum,
6965                    const GrB_Monoid  op,
6966                    const GrB_Vector  u,
```

```

6967             const GrB_Descriptor desc);
6968
6969 // GraphBLAS Scalar + monoid
6970 GrB_Info GrB_reduce(GrB_Scalar      s,
6971                    const GrB_BinaryOp accum,
6972                    const GrB_Monoid op,
6973                    const GrB_Vector u,
6974                    const GrB_Descriptor desc);
6975
6976 // GraphBLAS Scalar + binary operator
6977 GrB_Info GrB_reduce(GrB_Scalar      s,
6978                    const GrB_BinaryOp accum,
6979                    const GrB_BinaryOp op,
6980                    const GrB_Vector u,
6981                    const GrB_Descriptor desc);

```

6982 Parameters

6983 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
6984 a value that may be accumulated (optionally) with the result of the reduction
6985 operation. On output, this scalar holds the results of the operation.

6986 **accum** (IN) An optional binary operator used for accumulating entries into an exist-
6987 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,
6988 **GrB_NULL** should be specified.

6989 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
6990 the reduction operation. The \oplus operator must be commutative and associative;
6991 otherwise, the outcome of the operation is undefined.

6992 **u** (IN) The GraphBLAS vector on which reduction will be performed.

6993 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6994 should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
-------	-------	-------	-------------

6997 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
6998 tions. There are currently no non-default field/value pairs that can be set for this
6999 operation.

7000 Return Values

7001 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
7002 cessfully, and the output scalar (**s** or **val**) is ready to be used in the
7003 next method of the sequence.

7004	GrB_PANIC	Unknown internal error.
7005	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque
7006		GraphBLAS objects (input or output) is in an invalid state caused
7007		by a previous execution error. Call GrB_error() to access any error
7008		messages generated by the implementation.
7009	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
7010	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized by
7011		a call to a respective constructor.
7012	GrB_NULL_POINTER	val pointer is NULL.
7013	GrB_DOMAIN_MISMATCH	The domains of input and output arguments are incompatible with
7014		the corresponding domains of the accumulation operator, or reduce
7015		operator.

7016 Description

This variant of **GrB_reduce** computes the result of performing a reduction across all of the stored elements of an input vector storing the result into either **s** or **val**. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[\bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7017 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7018 Logically, this operation occurs in three steps:

7019 **Setup** The internal vector used in the computation is formed and its domain is tested for
7020 compatibility.

7021 **Compute** The indicated computations are carried out.

7022 **Output** The result is written into the output scalar.

7023 One vector argument is used in this **GrB_reduce** operation:

- 7024 1. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7025 The output scalar, argument vector, reduction operator and accumulation operator (if provided)
7026 are tested for domain compatibility as follows:

- 7027 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
7028 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

- 7029 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
 7030 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 7031 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 7032 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).

7033 Two domains are compatible with each other if values from one domain can be cast to values in
 7034 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7035 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7036 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 7037 error listed above is returned.

7038 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one
 7039 of the following occurs depending on the output variant:

$$7040 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7041 or

$$7042 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7043 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7044 For all other cases, the internal vector and scalar used in the computation is formed (\leftarrow denotes
 7045 copy):

- 7046 1. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 7047 2. Scalar $\tilde{s} \leftarrow \text{s}$ (GraphBLAS scalar case).

7048 We are now ready to carry out the reduction and any additional associated operations. An inter-
 7049 mediate scalar result t is computed as follows:

$$7050 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

7051 where $\oplus = \odot(\text{op})$.

7052 The final reduction value is computed as follows:

$$7053 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7054 or

$$7055 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7056 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
7057 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7058 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7059 Reduce all stored values into a single scalar.

7060 C Syntax

```
7061 // scalar value + monoid (only)
7062 GrB_Info GrB_reduce(<type>          *val,
7063                    const GrB_BinaryOp accum,
7064                    const GrB_Monoid  op,
7065                    const GrB_Matrix  A,
7066                    const GrB_Descriptor desc);
7067
7068 // GraphBLAS Scalar + monoid
7069 GrB_Info GrB_reduce(GrB_Scalar      s,
7070                    const GrB_BinaryOp accum,
7071                    const GrB_Monoid  op,
7072                    const GrB_Matrix  A,
7073                    const GrB_Descriptor desc);
7074
7075 // GraphBLAS Scalar + binary operator
7076 GrB_Info GrB_reduce(GrB_Scalar      s,
7077                    const GrB_BinaryOp accum,
7078                    const GrB_BinaryOp op,
7079                    const GrB_Matrix  A,
7080                    const GrB_Descriptor desc);
```

7081 Parameters

7082 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
7083 a value that may be accumulated (optionally) with the result of the reduction
7084 operation. On output, this scalar holds the results of the operation.

7085 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or
7086 **val**) value. If assignment rather than accumulation is desired, GrB_NULL should
7087 be specified.

7088 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
7089 the reduction operation. The \oplus operator must be commutative and associative;
7090 otherwise, the outcome of the operation is undefined.

7091 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

7092 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
7093 should be specified. Non-default field/value pairs are listed as follows:

7094

7095

Param	Field	Value	Description
-------	-------	-------	-------------

7096 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
7097 tions. There are currently no non-default field/value pairs that can be set for this
7098 operation.

7099 Return Values

7100 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
7101 cessfully, and the output scalar (s or val) is ready to be used in the
7102 next method of the sequence.

7103 GrB_PANIC Unknown internal error.

7104 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
7105 GraphBLAS objects (input or output) is in an invalid state caused
7106 by a previous execution error. Call GrB_error() to access any error
7107 messages generated by the implementation.

7108 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7109 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
7110 a call to a respective constructor.

7111 GrB_NULL_POINTER val pointer is NULL.

7112 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
7113 the corresponding domains of the accumulation operator, or reduce
7114 operator.

7115 Description

This variant of GrB_reduce computes the result of performing a reduction across all of the stored elements of an input matrix storing the result into either s or val. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[\bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7116 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

7117 Logically, this operation occurs in three steps:

7118 **Setup** The internal matrix used in the computation is formed and its domain is tested for
 7119 compatibility.

7120 **Compute** The indicated computations are carried out.

7121 **Output** The result is written into the output scalar.

7122 One matrix argument is used in this `GrB_reduce` operation:

7123 1. $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7124 The output scalar, argument matrix, reduction operator and accumulation operator (if provided)
 7125 are tested for domain compatibility as follows:

7126 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
 7127 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

7128 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
 7129 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 7130 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

7131 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).

7132 Two domains are compatible with each other if values from one domain can be cast to values in
 7133 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7134 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7135 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 7136 error listed above is returned.

7137 The number of values stored in the input, A , is checked. If there are no stored values in A , then
 7138 one of the following occurs depending on the output variant:

$$7139 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7140 or

$$7141 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7142 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7143 For all other cases, the internal matrix and scalar used in the computation is formed (\leftarrow denotes
 7144 copy):

7145 1. Matrix $\tilde{A} \leftarrow A$.

7146 2. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

7147 We are now ready to carry out the reduce and any additional associated operations. An intermediate
 7148 scalar result t is computed as follows:

$$7149 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7150 where $\oplus = \odot(\text{op})$.

7151 The final reduction value is computed as follows:

$$7152 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7153 or

$$7154 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7155 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
 7156 GrB_SUCCESS and the new contents of the output scalar is as defined above.

7157 4.3.11 transpose: Transpose rows and columns of a matrix

7158 This version computes a new matrix that is the transpose of the source matrix.

7159 C Syntax

```
7160      GrB_Info GrB_transpose(GrB_Matrix      C,
7161                           const GrB_Matrix Mask,
7162                           const GrB_BinaryOp accum,
7163                           const GrB_Matrix A,
7164                           const GrB_Descriptor desc);
```

7165 Parameters

7166 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 7167 that may be accumulated with the result of the transpose operation. On output,
 7168 the matrix holds the results of the operation.

7169 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 7170 stored into the output matrix C. The mask dimensions must match those of the
 7171 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 7172 of the Mask matrix must be of type bool or any of the predefined “built-in” types
 7173 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
 7174 dimensions of C), GrB_NULL should be specified.

7175 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
7176 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
7177 specified.

7178 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7179 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
7180 should be specified. Non-default field/value pairs are listed as follows:

7181

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.

7182

7183 **Return Values**

7184 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
7185 blocking mode, this indicates that the compatibility tests on di-
7186 mensions and domains for the input arguments passed successfully.
7187 Either way, output matrix **C** is ready to be used in the next method
7188 of the sequence.

7189 **GrB_PANIC** Unknown internal error.

7190 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
7191 GraphBLAS objects (input or output) is in an invalid state caused
7192 by a previous execution error. Call **GrB_error()** to access any error
7193 messages generated by the implementation.

7194 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

7195 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
7196 a call to **new** (or **Matrix_dup** for matrix parameters).

7197 **GrB_DIMENSION_MISMATCH** **mask**, **C** and/or **A** dimensions are incompatible.

7198 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the cor-
7199 responding domains of the accumulation operator, or the mask's do-
7200 main is not compatible with **bool** (in the case where **desc[GrB_MASK].GrB_STRUCTURE**
7201 is not set).

7202 Description

7203 GrB_transpose computes the result of performing a transpose of the input matrix: $C = A^T$; or, if an
 7204 optional binary accumulation operator (\odot) is provided, $C = C \odot A^T$. We note that the input matrix
 7205 A can itself be optionally transposed before the operation, which would cause either an assignment
 7206 from A to C or an accumulation of A into C.

7207 Logically, this operation occurs in three steps:

7208 **Setup** The internal matrix and mask used in the computation are formed and their domains
 7209 and dimensions are tested for compatibility.

7210 **Compute** The indicated computations are carried out.

7211 **Output** The result is written into the output matrix, possibly under control of a mask.

7212 Up to three matrix arguments are used in this GrB_transpose operation:

- 7213 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7214 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)
- 7215 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7216 The argument matrices and accumulation operator (if provided) are tested for domain compatibility
 7217 as follows:

- 7218 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
 7219 must be from one of the pre-defined types of Table 3.2.
- 7220 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$ of the input matrix.
- 7221 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 7222 of the accumulation operator and $\mathbf{D}(A)$ of the input matrix must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 7223 of the accumulation operator.

7224 Two domains are compatible with each other if values from one domain can be cast to values in
 7225 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all
 7226 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7227 any compatibility rule above is violated, execution of GrB_transpose ends and the domain mismatch
 7228 error listed above is returned.

7229 From the argument matrices, the internal matrices and mask used in the computation are formed
 7230 (\leftarrow denotes copy):

- 7231 1. Matrix $\tilde{C} \leftarrow C$.
- 7232 2. Two-dimensional mask, \tilde{M} , is computed from argument Mask as follows:

- 7233 (a) If $\text{Mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
7234 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 7235 (b) If $\text{Mask} \neq \text{GrB_NULL}$,
- 7236 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
7237 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
- 7238 ii. Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
7239 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
- 7240 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$.
- 7241 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

7242 The internal matrices and masks are checked for dimension compatibility. The following conditions
7243 must hold:

- 7244 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
- 7245 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
- 7246 3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.
- 7247 4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

7248 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension
7249 mismatch error listed above is returned.

7250 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
7251 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7252 We are now ready to carry out the matrix transposition and any additional associated operations.
7253 We describe this in terms of two intermediate matrices:

- 7254 • $\widetilde{\mathbf{T}}$: The matrix holding the transpose of $\widetilde{\mathbf{A}}$.
- 7255 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

7256 The intermediate matrix

$$7257 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7258 is created.

7259 The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7260 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- 7261 • If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$7262 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7263 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 7264 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned}
 7265 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 7266 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7267 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7268 \quad & \\
 7269 \quad &
 \end{aligned}$$

7270 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7271 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 7272 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 7273 mask which acts as a “write mask”.

- 7274 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 7275 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7276 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7277 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 7278 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 7279 mask are unchanged:

$$7280 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7281 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 7282 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 7283 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 7284 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 7285 sequence.

7286 4.3.12 kronecker: Kronecker product of two matrices

7287 Computes the Kronecker product of two matrices. The result is a matrix.

7288 C Syntax

```

7289      GrB_Info GrB_kronecker(GrB_Matrix      C,
7290                           const GrB_Matrix Mask,
7291                           const GrB_BinaryOp accum,
7292                           const GrB_Semiring op,
7293                           const GrB_Matrix A,
7294                           const GrB_Matrix B,
7295                           const GrB_Descriptor desc);
7296
  
```

```

7297 GrB_Info GrB_kronecker(GrB_Matrix      C,
7298                        const GrB_Matrix  Mask,
7299                        const GrB_BinaryOp accum,
7300                        const GrB_Monoid   op,
7301                        const GrB_Matrix  A,
7302                        const GrB_Matrix  B,
7303                        const GrB_Descriptor desc);
7304
7305 GrB_Info GrB_kronecker(GrB_Matrix      C,
7306                        const GrB_Matrix  Mask,
7307                        const GrB_BinaryOp accum,
7308                        const GrB_BinaryOp op,
7309                        const GrB_Matrix  A,
7310                        const GrB_Matrix  B,
7311                        const GrB_Descriptor desc);

```

7312 Parameters

7313 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
7314 that may be accumulated with the result of the Kronecker product. On output,
7315 the matrix holds the results of the operation.

7316 **Mask** (IN) An optional “write” mask that controls which results from this operation are
7317 stored into the output matrix C. The mask dimensions must match those of the
7318 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
7319 of the Mask matrix must be of type bool or any of the predefined “built-in” types
7320 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the
7321 dimensions of C), GrB_NULL should be specified.

7322 **accum** (IN) An optional binary operator used for accumulating entries into existing C
7323 entries. If assignment rather than accumulation is desired, GrB_NULL should be
7324 specified.

7325 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
7326 operation. Depending on which type is passed, the following defines the binary
7327 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

7328 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

7329 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-
7330 nored.

7331 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
7332 is ignored.

7333 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
7334 product.

7335 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
7336 product.

7337 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
7338 should be specified. Non-default field/value pairs are listed as follows:
7339

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.

7341 Return Values

7342 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
7343 blocking mode, this indicates that the compatibility tests on di-
7344 mensions and domains for the input arguments passed successfully.
7345 Either way, output matrix C is ready to be used in the next method
7346 of the sequence.

7347 GrB_PANIC Unknown internal error.

7348 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
7349 GraphBLAS objects (input or output) is in an invalid state caused
7350 by a previous execution error. Call GrB_error() to access any error
7351 messages generated by the implementation.

7352 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7353 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
7354 a call to new (or Matrix_dup for matrix parameters).

7355 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

7356 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
7357 corresponding domains of the binary operator (op) or accumulation
7358 operator, or the mask's domain is not compatible with bool (in the
7359 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

7360 Description

7361 GrB_kronecker computes the Kronecker product $C = A \otimes B$ or, if an optional binary accumulation
7362 operator (\odot) is provided, $C = C \odot (A \otimes B)$ (where matrices A and B can be optionally transposed).

7363 The Kronecker product is defined as follows:

7364

$$7365 \quad \mathbf{C} = \mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{0,0} \otimes \mathbf{B} & A_{0,1} \otimes \mathbf{B} & \dots & A_{0,n_A-1} \otimes \mathbf{B} \\ A_{1,0} \otimes \mathbf{B} & A_{1,1} \otimes \mathbf{B} & \dots & A_{1,n_A-1} \otimes \mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes \mathbf{B} & A_{m_A-1,1} \otimes \mathbf{B} & \dots & A_{m_A-1,n_A-1} \otimes \mathbf{B} \end{bmatrix}$$

7366 where $\mathbf{A} : \mathbb{S}^{m_A \times n_A}$, $\mathbf{B} : \mathbb{S}^{m_B \times n_B}$, and $\mathbf{C} : \mathbb{S}^{m_A m_B \times n_A n_B}$. More explicitly, the elements of the
7367 Kronecker product are defined as

$$7368 \quad \mathbf{C}(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7369 where \otimes is the multiplicative operator specified by the **op** parameter.

7370 Logically, this operation occurs in three steps:

7371 **Setup** The internal matrices and mask used in the computation are formed and their domains
7372 and dimensions are tested for compatibility.

7373 **Compute** The indicated computations are carried out.

7374 **Output** The result is written into the output matrix, possibly under control of a mask.

7375 Up to four argument matrices are used in the **GrB_kronecker** operation:

- 7376 1. $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 7377 2. $\mathbf{Mask} = \langle \mathbf{D}(\mathbf{Mask}), \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \mathbf{L}(\mathbf{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 7378 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 7379 4. $\mathbf{B} = \langle \mathbf{D}(\mathbf{B}), \mathbf{nrows}(\mathbf{B}), \mathbf{ncols}(\mathbf{B}), \mathbf{L}(\mathbf{B}) = \{(i, j, B_{ij})\} \rangle$

7380 The argument matrices, the "product" operator (**op**), and the accumulation operator (if provided)
7381 are tested for domain compatibility as follows:

- 7382 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\mathbf{Mask})$
7383 must be from one of the pre-defined types of Table 3.2.
- 7384 2. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 7385 3. $\mathbf{D}(\mathbf{B})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 7386 4. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 7387 5. If **accum** is not **GrB_NULL**, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
7388 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of **op** must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
7389 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.

7423 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$
7424 $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$ is created. The value of
7425 each of its elements is computed by

$$7426 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7427 where \otimes is the multiplicative operator specified by the `op` parameter.

7428 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7429 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 7430 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$7431 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7432 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
7433 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$7434 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7435 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7436 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7439 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7440 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
7441 using what is called a *standard matrix mask and replace*. This is carried out under control of the
7442 mask which acts as a “write mask”.

- 7443 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
7444 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7445 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7446 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
7447 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
7448 mask are unchanged:

$$7449 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7450 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
7451 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
7452 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
7453 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7454 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,...)

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.

- 7485 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified
7486 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats
7487 have been deferred.
- 7488 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7489 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return
7490 error if duplicate locations encountered.
- 7491 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7492 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7493 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type T (not to
7494 be confused with the proposed unary operator).
- 7495 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added
7496 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to
7497 a method that is not supported by the implementation.
- 7498 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque
7499 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7500 • (Issue BB-45) Removed language about annihilators.
- 7501 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7502 • Updated a number algorithms in the appendix to use new operations and methods.
- 7503 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the
7504 specification.
- 7505 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and
7506 `GRB_SUBVERSION`.
- 7507 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7508 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection
7509 fixed to read `eWiseAdd`/set union.
- 7510 • Typographical error (`NEQ` \rightarrow `NE`) in Description of Table 3.8.

7511 Changes in 1.3.0 (Released: 25 September 2019):

- 7512 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque
7513 GraphBLAS object as an argument.
- 7514 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7515 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a
7516 scalar.

7517
7518

7519

7520
7521

7522
7523

7524

7525
7526

7527
7528

7529
7530

7531

7532
7533

7534

7535
7536

7537
7538

7539

7540
7541

7542

7543
7544

7545
7546

7547

7548

7549

- (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.

- 7550 • Fixed miscellaneous typographical errors (such as \otimes , \oplus).
- 7551 Changes in 1.2.0:
- 7552 • Removed "provisional" clause.
- 7553 Changes in 1.1.0:
- 7554 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
 - 7555 `assign` operations.
 - 7556 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
 - 7557 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
 - 7558 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
 - 7559 from the domains of the binary operator provided.
 - 7560 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
 - 7561 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
 - 7562 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
 - 7563 which is returned when the capacities of the output arrays are insufficient to hold all of the
 - 7564 tuples.
 - 7565 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
 - 7566 face.
 - 7567 • Added replace flag (`z`) notation to Table 4.1.
 - 7568 • Updated the “Mathematical Description” of the `assign` operation in Table 4.1.
 - 7569 • Added triangle counting example.
 - 7570 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
 - 7571 sections of GraphBLAS operations when the respective text was the “standard” text (i.e.,
 - 7572 identical in a majority of the operations).
 - 7573 • Fixed typographical errors.
- 7574 Changes in 1.0.2:
- 7575 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
 - 7576 matrices and avoid casting issues in certain implementations.
 - 7577 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
 - 7578 erased outside the assigned area.
 - 7579 • Changes non-polymorphic interface:
 - 7580 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7581 – Renamed GrB_Vector_reduce_BinaryOp to GrB_Matrix_reduce_BinaryOp.
- 7582 – Renamed GrB_Vector_reduce_Monoid to GrB_Matrix_reduce_Monoid.
- 7583 • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7584 • 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.

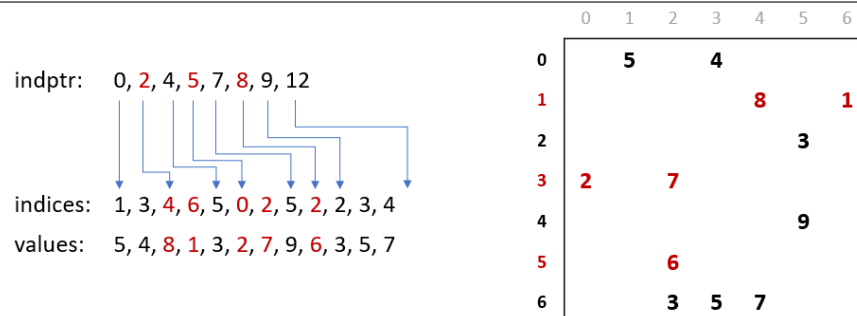


Figure B.1: Data layout for CSR format.

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.

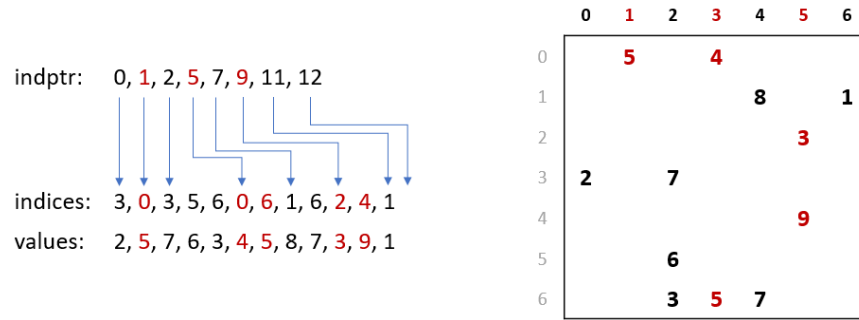


Figure B.2: Data layout for CSC format.

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.

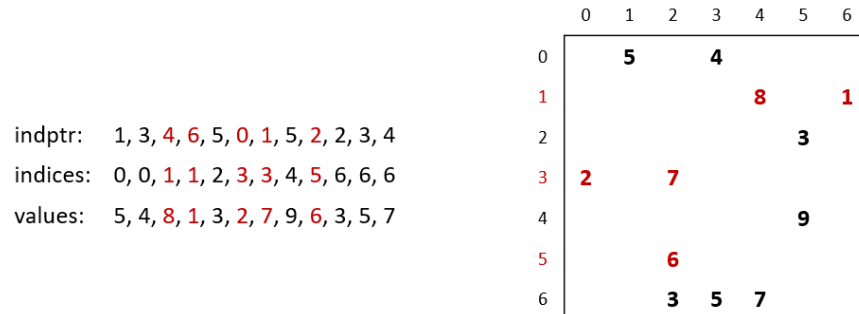


Figure B.3: Data layout for COO format.

7617 **Appendix C**

7618 **Examples**

C.1 Example: Level breadth-first search (BFS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i] = 0$ . (Vector  $v$  should be empty on input.)
11  */
12  GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
13  {
14      GrB_Index n;
15      GrB_Matrix_nrows(&n,A);                //  $n = \#$  of rows of  $A$ 
16
17      GrB_Vector_new(v,GrB_INT32,n);          // Vector<int32_t>  $v(n)$ 
18
19      GrB_Vector q;                          // vertices visited in each level
20      GrB_Vector_new(&q,GrB_BOOL,n);          // Vector<bool>  $q(n)$ 
21      GrB_Vector_setElement(q,(bool)true,s);  //  $q[s] = \text{true}$ , false everywhere else
22
23      /*
24       * BFS traversal and label the vertices.
25       */
26      int32_t d = 0;                          //  $d = \text{level in BFS traversal}$ 
27      bool succ = false;                      //  $\text{succ} == \text{true}$  when some successor found
28      do {
29          ++d;                                // next level (start with 1)
30          GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL); //  $v[q] = d$ 
31          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
32                q,A,GrB_DESC_RC);             //  $q[!v] = q \parallel A$ ; finds all the
33                                              // unvisited successors from current  $q$ 
34          GrB_reduce(&succ,GrB_NULL,GrB_LOR_MONOID_BOOL,
35                  q,GrB_NULL);                //  $\text{succ} = \parallel(q)$ 
36      } while (succ);                          // if there is no successor in  $q$ , we are done.
37
38      GrB_free(&q);                            //  $q$  vector no longer needed
39
40      return GrB_SUCCESS;
41  }

```

C.2 Example: Level BFS in GraphBLAS using apply

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i]$  does not have a stored element.
11  * Vector  $v$  should be uninitialized on input.
12  */
13  GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index n;
16      GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
17
18      GrB_Vector_new(v,GrB_INT32,n);     // Vector<int32_t>  $v(n) = 0$ 
19
20      GrB_Vector q;                     // vertices visited in each level
21      GrB_Vector_new(&q,GrB_BOOL,n);    // Vector<bool>  $q(n) = \text{false}$ 
22      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
23
24      /*
25       * BFS traversal and label the vertices.
26       */
27      int32_t level = 0;                // level = depth in BFS traversal
28      GrB_Index nvals;
29      do {
30          ++level;                      // next level (start with 1)
31          GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,
32                  GrB_SECOND_INT32,q,level,GrB_NULL); //  $v[q] = \text{level}$ 
33          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
34                  q,A,GrB_DESC_RC);    //  $q[!v] = q \vee A$ ; finds all the
35                                      // unvisited successors from current  $q$ 
36          GrB_Vector_nvals(&nvals, q);
37      } while (nvals);                 // if there is no successor in  $q$ , we are done.
38
39      GrB_free(&q);                    //  $q$  vector no longer needed
40
41      return GrB_SUCCESS;
42  }

```

C.3 Example: Parent BFS in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a binary  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS
9   * traversal of the graph and sets  $parents[i]$  to the index of vertex  $i$ 's parent.
10  * The parent of the root vertex,  $s$ , will be set to itself ( $parents[s] = s$ ). If
11  * vertex  $i$  is not reachable from  $s$ ,  $parents[i]$  will not contain a stored value.
12  */
13  GrB_Info BFS(GrB_Vector *parents, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index N;
16      GrB_Matrix_nrows(&N, A);           //  $N = \#$  vertices
17
18      GrB_Vector_new(parents, GrB_UINT64, N);
19      GrB_Vector_setElement(*parents, s, s); //  $parents[s] = s$ 
20
21      GrB_Vector wavefront;
22      GrB_Vector_new(&wavefront, GrB_UINT64, N);
23      GrB_Vector_setElement(wavefront, 1UL, s); //  $wavefront[s] = 1$ 
24
25      /*
26       * BFS traversal and label the vertices.
27       */
28      GrB_Index nvals;
29      GrB_Vector_nvals(&nvals, wavefront);
30
31      while (nvals > 0)
32      {
33          // convert all stored values in wavefront to their 0-based index
34          GrB_apply(wavefront, GrB_NULL, GrB_NULL, GrB_ROWINDEX_INT64,
35                  wavefront, 0UL, GrB_NULL);
36
37          // "FIRST" because left-multiplying wavefront rows. Masking out the parent
38          // list ensures wavefront values do not overwrite parents already stored.
39          GrB_vxm(wavefront, *parents, GrB_NULL, GrB_MIN_FIRST_SEMIRING_UINT64,
40                  wavefront, A, GrB_DESC_RSC);
41
42          // Don't need to mask here since we did it in vxm. Merges new parents in
43          // current wavefront with existing parents:  $parents += wavefront$ 
44          GrB_apply(*parents, GrB_NULL, GrB_PLUS_UINT64,
45                  GrB_IDENTITY_UINT64, wavefront, GrB_NULL);
46
47          GrB_Vector_nvals(&nvals, wavefront);
48      }
49
50      GrB_free(&wavefront);
51
52      return GrB_SUCCESS;
53  }

```


C.4 Example: Betweenness centrality (BC) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ ,
9   * compute the BC-metric vector  $\delta$ , which should be empty on input.
10  */
11 GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);                //  $n = \#$  of vertices in graph
15
16     GrB_Vector_new(delta, GrB_FP32, n);      // Vector<float>  $\delta(n)$ 
17
18     GrB_Matrix sigma;
19     GrB_Matrix_new(&sigma, GrB_INT32, n, n); //  $\text{Matrix}<\text{int32}_t> \text{sigma}(n, n)$ 
20                                           //  $\text{sigma}[d, k] = \#$  shortest paths to node  $k$  at level  $d$ 
21
22     GrB_Vector q;
23     GrB_Vector_new(&q, GrB_INT32, n);        // Vector<int32_t>  $q(n)$  of path counts
24     GrB_Vector_setElement(q, 1, s);          //  $q[s] = 1$ 
25
26     GrB_Vector p;
27     GrB_Vector_dup(&p, q);                   // Vector<int32_t>  $p(n)$  shortest path counts so far
28                                           //  $p = q$ 
29
30     GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
31             q, A, GrB_DESC_RC);              // get the first set of out neighbors
32
33     /*
34     * BFS phase
35     */
36     GrB_Index d = 0;                          // BFS level number
37     int32_t sum = 0;                          //  $\text{sum} == 0$  when BFS phase is complete
38
39     do {
40         GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\text{sigma}[d, :] = q$ 
41         GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
42         GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
43                 q, A, GrB_DESC_RC);        //  $q = \#$  paths to nodes reachable
44                                           // from current level
45         GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
46         ++d;
47     } while (sum);
48
49     /*
50     * BC computation phase
51     * ( $t_1, t_2, t_3, t_4$ ) 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); //  $t_1 = 1 + \delta$ 
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); //  $t_2 = \text{sigma}[i, :]$ 
63         GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t_2 = (1 + \delta) / \text{sigma}[i, :]$ 
64         GrB_mvx(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
65                 // add contributions made by

```

```

63         A, t2, GrB_NULL);
64     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, GrB_DESC_T0); // t4 = sigma[i-1,:]
65     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, GrB_TIMES_FP32, t4, t3, GrB_NULL); // t4 = sigma[i-1,:]*t3
66     GrB_eWiseAdd(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, *delta, t4, GrB_NULL); // accumulate into delta
67 }
68
69 GrB_free(&sigma);
70 GrB_free(&q); GrB_free(&p);
71 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
72
73 return GrB_SUCCESS;
74 }

```

C.5 Example: Batched BC in GraphBLAS

```

1  #include <stdlib.h>
2  #include "GraphBLAS.h" // in addition to other required C headers
3
4  // Compute partial BC metric for a subset of source vertices, s, in graph A
5  GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
6  {
7      GrB_Index n;
8      GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
9      GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
10
11     // index and value arrays needed to build numsp
12     GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
13     int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
14     for(int i=0; i<nsver; ++i) {
15         i_nsver[i] = i;
16         ones[i] = 1;
17     }
18
19     // numsp: structure holds the number of shortest paths for each node and starting vertex
20     // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
21     GrB_Matrix numsp;
22     GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
23     GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
24     free(i_nsver); free(ones);
25
26     // frontier: Holds the current frontier where values are path counts.
27     // Initialized to out vertices of each source node in s.
28     GrB_Matrix frontier;
29     GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
30     GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, GrB_DESC_RCT0);
31
32     // sigma: stores frontier information for each level of BFS phase. The memory
33     // for an entry in sigmas is only allocated within the do-while loop if needed.
34     // n is an upper bound on diameter.
35     GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n);
36
37     int32_t d = 0; // BFS level number
38     GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
39
40     // ----- The BFS phase (forward sweep) -----
41     do {
42         // sigmas[d](:,s) = dth level frontier from source vertex s
43         GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
44
45         GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
46                 GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
47         GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL, GrB_PLUS_INT32,
48                     numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
49         GrB_mxm(frontier, numsp, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
50                 A, frontier, GrB_DESC_RCT0); // f<!numsp> = A' +.* f (update frontier)
51         GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level
52         d++;
53     } while (nvals);
54
55     // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
56     GrB_Matrix nspinv;
57     GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
58     GrB_apply(nspinv, GrB_NULL, GrB_NULL,
59              GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
60
61     // bcu: BC updates for each vertex for each starting vertex in s
62     GrB_Matrix bcu;

```

```

63 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
64 GrB_assign(bcu, GrB_NULL, GrB_NULL,
65           1.0f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
66
67 GrB_Matrix w; // temporary workspace matrix
68 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
69
70 // ----- Tally phase (backward sweep) -----
71 for (int i=d-1; i>0; i--) {
72     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
73                 GrB_TIMES_FP32, bcu, nspinv, GrB_DESC_R); // w<sigmas[i]>=(1 ./ nsp).*bcu
74
75     // add contributions by successors and mask with that BFS level's frontier
76     GrB_mxm(w, sigmas[i-1], GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
77            A, w, GrB_DESC_R); // w<sigmas[i-1]> = (A +.* w)
78     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32, GrB_TIMES_FP32,
79                 w, numsp, GrB_NULL); // bcu += w .* numsp
80 }
81
82 // row reduce bcu and subtract "nsver" from every entry to account
83 // for 1 extra value per bcu row element.
84 GrB_reduce(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, bcu, GrB_NULL);
85 GrB_apply(*delta, GrB_NULL, GrB_NULL, GrB_MINUS_FP32, *delta, (float)nsver, GrB_NULL);
86
87 // Release resources
88 for (int i=0; i<d; i++) {
89     GrB_free(&(sigmas[i]));
90 }
91 free(sigmas);
92
93 GrB_free(&frontier); GrB_free(&numsp);
94 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
95
96 return GrB_SUCCESS;
97 }

```

C.6 Example: Maximal independent set (MIS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  // Assign a random number to each element scaled by the inverse of the node's degree.
8  // This will increase the probability that low degree nodes are selected and larger
9  // sets are selected.
10 void setRandom(void *out, const void *in)
11 {
12     uint32_t degree = *(uint32_t*)in;
13     *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
14 }
15
16 /*
17  * A variant of Luby's randomized algorithm [Luby 1985].
18  *
19  * Given a numeric n x n adjacency matrix A of an unweighted and undirected graph (where
20  * the value true represents an edge), compute a maximal set of independent vertices and
21  * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
22  * of the set (the iset vector should be uninitialized on input.)
23  */
24 GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
25 {
26     GrB_Index n;
27     GrB_Matrix_nrows(&n,A); // n = # of rows of A
28
29     GrB_Vector prob; // holds random probabilities for each node
30     GrB_Vector neighbor_max; // holds value of max neighbor probability
31     GrB_Vector new_members; // holds set of new members to iset
32     GrB_Vector new_neighbors; // holds set of new neighbors to new iset mbrs.
33     GrB_Vector candidates; // candidate members to iset
34
35     GrB_Vector_new(&prob,GrB_FP32,n);
36     GrB_Vector_new(&neighbor_max,GrB_FP32,n);
37     GrB_Vector_new(&new_members,GrB_BOOL,n);
38     GrB_Vector_new(&new_neighbors,GrB_BOOL,n);
39     GrB_Vector_new(&candidates,GrB_BOOL,n);
40     GrB_Vector_new(iset,GrB_BOOL,n); // Initialize independent set vector, bool
41
42     GrB_UnaryOp set_random;
43     GrB_UnaryOp_new(&set_random,setRandom,GrB_FP32,GrB_UINT32);
44
45     // compute the degree of each vertex.
46     GrB_Vector degrees;
47     GrB_Vector_new(&degrees,GrB_FP64,n);
48     GrB_reduce(degrees,GrB_NULL,GrB_NULL,GrB_PLUS_FP64,A,GrB_NULL);
49
50     // Isolated vertices are not candidates: candidates[degrees != 0] = true
51     GrB_assign(candidates,degrees,GrB_NULL,true,GrB_ALL,n,GrB_NULL);
52
53     // add all singletons to iset: iset[degree == 0] = 1
54     GrB_assign(*iset,degrees,GrB_NULL,true,GrB_ALL,n,GrB_DESC_RC);
55
56     // Iterate while there are candidates to check.
57     GrB_Index nvals;
58     GrB_Vector_nvals(&nvals,candidates);
59     while (nvals > 0) {
60         // compute a random probability scaled by inverse of degree
61         GrB_apply(prob,candidates,GrB_NULL,set_random,degrees,GrB_DESC_R);
62     }

```

```

63 // compute the max probability of all neighbors
64 GrB_mnv(neighbor_max, candidates, GrB_NULL, GrB_MAX_SECOND_SEMIRING_FP32, A, prob, GrB_DESC_R);
65
66 // select vertex if its probability is larger than all its active neighbors,
67 // and apply a "masked no-op" to remove stored falses
68 GrB_eWiseAdd(new_members, GrB_NULL, GrB_NULL, GrB_GT_FP64, prob, neighbor_max, GrB_NULL);
69 GrB_apply(new_members, new_members, GrB_NULL, GrB_IDENTITY_BOOL, new_members, GrB_DESC_R);
70
71 // add new members to independent set.
72 GrB_eWiseAdd(*iset, GrB_NULL, GrB_NULL, GrB_LOR, *iset, new_members, GrB_NULL);
73
74 // remove new members from set of candidates  $c = c \ominus !new$ 
75 GrB_eWiseMult(candidates, new_members, GrB_NULL,
76               GrB_LAND, candidates, candidates, GrB_DESC_RC);
77
78 GrB_Vector_nvals(&nvals, candidates);
79 if (nvals == 0) { break; } // early exit condition
80
81 // Neighbors of new members can also be removed from candidates
82 GrB_mnv(new_neighbors, candidates, GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL,
83         A, new_members, GrB_NULL);
84 GrB_eWiseMult(candidates, new_neighbors, GrB_NULL, GrB_LAND,
85               candidates, candidates, GrB_DESC_RC);
86
87 GrB_Vector_nvals(&nvals, candidates);
88 }
89
90 GrB_free(&neighbor_max); // free all objects "new'ed"
91 GrB_free(&new_members);
92 GrB_free(&new_neighbors);
93 GrB_free(&prob);
94 GrB_free(&candidates);
95 GrB_free(&set_random);
96 GrB_free(&degrees);
97
98 return GrB_SUCCESS;
99 }

```

C.7 Example: Counting triangles in GraphBLAS

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given an  $n \times n$  boolean adjacency matrix,  $A$ , of an undirected graph, computes
9  * the number of triangles in the graph.
10 */
11 uint64_t triangle_count(GrB_Matrix A)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);           //  $n = \#$  of vertices
15
16     //  $L$ :  $N \times N$ , lower-triangular, bool
17     GrB_Matrix L;
18     GrB_Matrix_new(&L, GrB_BOOL, n, n);
19     GrB_select(L, GrB_NULL, GrB_NULL, GrB_TRIL, A, 0UL, GrB_NULL);
20
21     GrB_Matrix C;
22     GrB_Matrix_new(&C, GrB_UINT64, n, n);
23
24     GrB_mxm(C, L, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_UINT64, L, L, GrB_NULL); //  $C \langle L \rangle = L +.* L$ 
25
26     uint64_t count;
27     GrB_reduce(&count, GrB_NULL, GrB_PLUS_MONOID_UINT64, C, GrB_NULL); // 1-norm of  $C$ 
28
29     GrB_free(&C);
30     GrB_free(&L);
31
32     return count;
33 }
```