

# The GraphBLAS C API Specification <sup>†</sup>:

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 07:56:28 EDT

<sup>†</sup>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	<b>1 Introduction</b>	<b>13</b>
30	<b>2 Basic concepts</b>	<b>15</b>
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	<b>3 Objects</b>	<b>31</b>
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	<b>4</b>	<b>Methods</b>	<b>51</b>
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	<b>5</b>	<b>Nonpolymorphic interface[Scott: NEW CONTENT]</b>	<b>265</b>
164	<b>A</b>	<b>Revision history</b>	<b>277</b>
165	<b>B</b>	<b>Non-opaque data format definitions</b>	<b>283</b>
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	<b>C Examples</b>	<b>285</b>
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. . . . . 46
196	3.12	Descriptor types and literals for fields and values. . . . . 48
197	3.13	Predefined GraphBLAS descriptors. . . . . 49
198	3.14	Enumeration literals and corresponding values returned by GraphBLAS methods
199		and operations. . . . . 50
200	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in
201		this specification. . . . . 117
202	5.1	Long-name, nonpolymorphic form of GraphBLAS methods. . . . . 265

203	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	266
204	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	267
205	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	268
206	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	269
207	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	270
208	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	271
209	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	272
210	5.9	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
211		CONTENT]	273
212	5.10	Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW	
213		CONTENT]	274
214	5.11	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	275

# 215 List of Figures

216	3.1 Hierarchy of algebraic object classes in GraphBLAS. . . . .	43
217	4.1 Flowchart for the GraphBLAS operations. . . . .	118
218	B.1 Data layout for CSR format. . . . .	283
219	B.2 Data layout for CSC format. . . . .	284
220	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*

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

- 471 • *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as param-  
472 eters to a GraphBLAS method are dimension (or shape) compatible if they have the correct  
473 number of dimensions and sizes for each dimension to satisfy the rules of the mathematical def-  
474 inition of the operation associated with the method. If any *dimension compatibility* rule above  
475 is violated, execution of the GraphBLAS method ends and the GrB\_DIMENSION\_MISMATCH  
476 error is returned.
- 477 • *domain compatible*: Two domains for which values from one domain can be cast to values in  
478 the other domain as per the rules of the C language. In particular, domains from Table 3.2  
479 are all compatible with each other, and a domain from a user-defined type is only compatible  
480 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS  
481 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.<sup>1</sup> 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

---

<sup>1</sup>More 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

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

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

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

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

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

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

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

### 786 3.3 Types (domains)

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

---

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

835 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  
836 domains,  $D_{out}$ ,  $D_{in_1}$ ,  $D_{in_2}$ , the domain of GraphBLAS indices, and an operation  $f_i : D_{in_1} \times I_{U64}^2 \times$   
837  $D_{in_2} \rightarrow D_{out}$  (where  $I_{U64}$  corresponds to the domain of a `GrB_Index`). For a given GraphBLAS  
838 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  $\mathbf{f}(F_i) = f_i$ .

839 User-defined operators can be created with calls to `GrB_UnaryOp_new`, `GrB_BinaryOp_new`, and  
840 `GrB_IndexUnaryOp_new`, respectively. See Section 4.2.2 for information on these methods. The  
841 GraphBLAS C API predefines a number of these operators. These are listed in Tables 3.5 and 3.6.  
842 Note that most entries in these tables represent a “family” of predefined operators for a set of  
843 different types represented by the  $T$ ,  $I$ , or  $F$  in their names. For example, the multiplicative  
844 inverse (`GrB_MINV_F`) function is only defined for floating-point types ( $F = \text{FP32}$  or  $\text{FP64}$ ). The  
845 division (`GrB_DIV_T`) function is defined for all types, but only if  $y \neq 0$  for integral and floating  
846 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*<sup>1</sup> 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*<sup>1</sup> 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.

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

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

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



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

GraphBLAS identifier	Domains, $T$ ( $T \times T \rightarrow T$ )	+ identity $\times$ annihilator	Description
<code>GrB_PLUS_TIMES_SEMIRING_T</code>	<code>UINT<math>x</math></code> <code>INT<math>x</math></code> <code>FP<math>x</math></code>	0 0 0	arithmetic semiring
<code>GrB_MIN_PLUS_SEMIRING_T</code>	<code>UINT<math>x</math></code> <code>INT<math>x</math></code> <code>FP<math>x</math></code>	<code>UINT<math>x</math>_MAX</code> <code>INT<math>x</math>_MAX</code> <code>INFINITY</code>	min-plus semiring
<code>GrB_MAX_PLUS_SEMIRING_T</code>	<code>INT<math>x</math></code> <code>FP<math>x</math></code>	<code>INT<math>x</math>_MIN</code> <code>-INFINITY</code>	max-plus semiring
<code>GrB_MIN_TIMES_SEMIRING_T</code>	<code>UINT<math>x</math></code>	<code>UINT<math>x</math>_MAX</code>	min-times semiring
<code>GrB_MIN_MAX_SEMIRING_T</code>	<code>UINT<math>x</math></code> <code>INT<math>x</math></code> <code>FP<math>x</math></code>	<code>UINT<math>x</math>_MAX</code> <code>INT<math>x</math>_MAX</code> <code>INFINITY</code>	min-max semiring
<code>GrB_MAX_MIN_SEMIRING_T</code>	<code>UINT<math>x</math></code> <code>INT<math>x</math></code> <code>FP<math>x</math></code>	0 <code>INT<math>x</math>_MIN</code> <code>-INFINITY</code>	max-min semiring
<code>GrB_MAX_TIMES_SEMIRING_T</code>	<code>UINT<math>x</math></code>	0	max-times semiring
<code>GrB_PLUS_MIN_SEMIRING_T</code>	<code>UINT<math>x</math></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
	$\text{INT}x$	$\text{INT}x\_MIN$	
	$\text{FP}x$	$-INFINITY$	

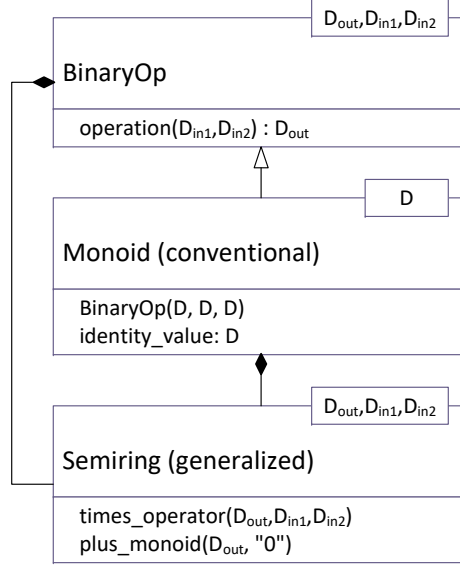


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

## 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*,  $\sigma$ , 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,  $\sigma$ , 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.

Some fields are read-only, such as the version number of the library, attempting to modify these fields with **set** will result in a `GrB_INVALID_VALUE` error.

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.

(a) Types used with GraphBLAS descriptors.

Field Name	Value	Implementing Objects	Type
GrB_OUTP	0	GrB_Descriptor	GrB_Desc_Value
GrB_MASK	1	GrB_Descriptor	GrB_Desc_Value
GrB_INP0	2	GrB_Descriptor	GrB_Desc_Value
GrB_INP1	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	104	Global, GrB_Descriptor	GrB_Index
GrB_STORAGE_ORIENTATION_HINT	200	Global, Collection	GrB_ROWMAJOR, GrB_COLMAJOR
GrB_STORAGE_FORMAT_HINT	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
GrB_Descriptor	Type of a GraphBLAS descriptor object.
GrB_Desc_Field	The descriptor field enumeration.
GrB_Desc_Value	The descriptor value enumeration.

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

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

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

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

---



Table 3.13: 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`.

## 1016 **Return Values**

1017 `GrB_SUCCESS` operation completed successfully.

1018 `GrB_PANIC` unknown internal error.

1019 `GrB_INVALID_VALUE` invalid mode specified, or method called multiple times.

## 1020 **Description**

1021 The `init` method creates and initializes a GraphBLAS C API context. The argument to `GrB_init`  
1022 defines the mode for the context. The two available modes are:

- 1023 • `GrB_BLOCKING`: In this mode, each method in a sequence returns after its computations have  
1024 completed and output arguments are available to subsequent statements in an application.  
1025 When executing in `GrB_BLOCKING` mode, the methods execute in program order.
- 1026 • `GrB_NONBLOCKING`: In this mode, methods in a sequence may return after arguments in  
1027 the method have been tested for dimension and domain compatibility within the method  
1028 but potentially before their computations complete. Output arguments are available to sub-  
1029 sequent GraphBLAS methods in an application. When executing in `GrB_NONBLOCKING`  
1030 mode, the methods in a sequence may execute in any order that preserves the mathematical  
1031 result defined by the sequence.

1032 An application can only create one context per execution instance. An application may only call  
1033 `GrB_Init` once. Calling `GrB_Init` more than once results in undefined behavior.

### 1034 **4.1.2 finalize: Finalize a GraphBLAS context**

1035 Terminates and frees any internal resources created to support the GraphBLAS C API context.

## 1036 **C Syntax**

1037 `GrB_Info GrB_finalize();`

## 1038 **Return Values**

1039 `GrB_SUCCESS` operation completed successfully.

1040 `GrB_PANIC` unknown internal error.

## 1041 **Description**

1042 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS  
1043 C API context. `GrB_finalize` may only be called after a context has been initialized by calling  
1044 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-  
1045 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined  
1046 behavior.

### 1047 **4.1.3 getVersion: Get the version number of the standard.**

1048 Query the library for the version number of the standard that this library implements.

## 1049 **C Syntax**

```
1050         GrB_Info GrB_getVersion(unsigned int *version,  
1051                                unsigned int *subversion);
```

## 1052 **Parameters**

1053 version (OUT) On successful return will hold the value of the major version number.

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

## 1055 **Return Values**

1056 GrB\_SUCCESS operation completed successfully.

1057 GrB\_PANIC unknown internal error.

## 1058 **Description**

1059 The `getVersion` method is used to query the major and minor version number of the GraphBLAS  
1060 C API specification that the library implements at runtime. To support compile time queries the  
1061 following two macros shall also be defined by the library.

```
1062         #define GRB_VERSION      2  
1063         #define GRB_SUBVERSION  0
```

## 1064 **4.2 Object methods**

1065 This section describes methods that setup and operate on GraphBLAS opaque objects but are not  
1066 part of the the GraphBLAS math specification.

## 1067 4.2.1 Query methods

1068 The methods in this section query and, depending on the field, set internal fields of many Graph-  
1069 BLAS objects.

### 1070 4.2.1.1 get: Query the value of an object

#### 1071 C Syntax

```
1072     GrB_Info GrB_<OBJ>_get(GrB_<OBJ> o, GrB_Field field, ...);
1073
1074     GrB_Info GrB_Scalar_get(GrB_Scalar s, GrB_Field field, ...);
1075     GrB_Info GrB_Vector_get(GrB_Vector v, GrB_Field field, ...);
1076     GrB_Info GrB_Matrix_get(GrB_Matrix A, GrB_Field field, ...);
1077
1078     GrB_Info GrB_UnaryOp_get(GrB_UnaryOp op, GrB_Field field, ...);
1079     GrB_Info GrB_IndexUnaryOp_get(GrB_IndexUnaryOp op, GrB_Field field, ...);
1080     GrB_Info GrB_BinaryOp_get(GrB_BinaryOp op, GrB_Field field, ...);
1081     GrB_Info GrB_Monoid_get(GrB_Monoid op, GrB_Field field, ...);
1082     GrB_Info GrB_Semiring_get(GrB_Semiring op, GrB_Field field, ...);
1083
1084     GrB_Info GrB_Descriptor_get(GrB_Descriptor op, GrB_Field field, ...);
1085     GrB_Info GrB_Type_get(GrB_Type op, GrB_Field field, ...);
1086
1087     GrB_Info GrB_Global_get(GrB_Field field, ...);
```

#### 1088 Parameters

1089 OBJ is replaced in each signature by the object type being queried.

1090 OBJ (IN) An existing GraphBLAS object which is being queried.

1091 field (IN) The internal field being queried.

1092 ... (OUT) A pointer to a variable dependent on field to be filled with the value of the  
1093 internal field.

#### 1094 Return Value

1095 GrB\_SUCCESS The method completed successfully.

1096 GrB\_PANIC unknown internal error.

1097 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1098 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

1099 GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 1100 Description

1101 Queries a field of an existing GraphBLAS object.

### 1102 4.2.1.2 Descriptor\_set: Set content of descriptor

1103 Sets the content for a field for an existing descriptor.

## 1104 C Syntax

```
1105 GrB_Info GrB_Descriptor_set(GrB_Descriptor desc,  
1106                             GrB_Desc_Field field,  
1107                             GrB_Desc_Value val);
```

## 1108 Parameters

1109 desc (IN) An existing GraphBLAS descriptor to be modified.

1110 field (IN) The field being set.

1111 val (IN) New value for the field being set.

## 1112 Return Values

1113 GrB\_SUCCESS operation completed successfully.

1114 GrB\_PANIC unknown internal error.

1115 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1116 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

1117 GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 1118 Description

1119 For a given descriptor, the GrB\_Descriptor\_set method can be called for each field in the descriptor  
1120 to set the value associated with that field. Valid values for the field parameter include the following:

1121 GrB\_OUTP refers to the output parameter (result) of the operation.

1122       GrB\_MASK refers to the mask parameter of the operation.

1123       GrB\_INP0 refers to the first input parameters of the operation (matrices and vectors).

1124       GrB\_INP1 refers to the second input parameters of the operation (matrices and vectors).

1125   Valid values for the val parameter are:

1126       GrB\_STRUCTURE Use only the structure of the stored values of the corresponding mask  
1127                      (GrB\_MASK) parameter.

1128       GrB\_COMP Use the complement of the corresponding mask (GrB\_MASK) param-  
1129                   eter. When combined with GrB\_STRUCTURE, the complement of the  
1130                   structure of the mask is used without evaluating the values stored.

1131       GrB\_TRAN Use the transpose of the corresponding matrix parameter (valid for input  
1132                   matrix parameters only).

1133       GrB\_REPLACE When assigning the masked values to the output matrix or vector, clear  
1134                   the matrix first (or clear the non-masked entries). The default behavior  
1135                   is to leave non-masked locations unchanged. Valid for the GrB\_OUTP  
1136                   parameter only.

1137   Descriptor values can only be set, and once set, cannot be cleared. As, in the case of GrB\_MASK,  
1138   multiple values can be set and all will apply (for example, both GrB\_COMP and GrB\_STRUCTURE).  
1139   A value for a given field may be set multiple times but will have no additional effect. Fields that  
1140   have no values set result in their default behavior, as defined in Section 3.7.

## 1141   4.2.2   Algebra methods

### 1142   4.2.2.1   Type\_new: Construct a new GraphBLAS (user-defined) type

1143   Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,  
1144   monoids, semirings, vectors and matrices.

## 1145   C Syntax

```
1146       GrB_Info GrB_Type_new(GrB_Type   *utype,  
1147                                   size_t       sizeof(ctype));
```

## 1148   Parameters

1149       utype (INOUT) On successful return, contains a handle to the newly created user-defined  
1150                   GraphBLAS type object.

1151       ctype (IN) A C type that defines the new GraphBLAS user-defined type.



## 1152 Return Values

1153                   GrB\_SUCCESS operation completed successfully.

1154                   GrB\_PANIC unknown internal error.

1155                   GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1156                   GrB\_NULL\_POINTER utype pointer is NULL.

## 1157 Description

1158 Given a C type `ctype`, the `Type_new` method returns in `utype` a handle to a new GraphBLAS type  
1159 that is equivalent to the C type. Variables of this `ctype` must be a struct, union, or fixed-size array.  
1160 In particular, given two variables, `src` and `dst`, of type `ctype`, the following operation must be a  
1161 valid way to copy the contents of `src` to `dst`:

1162                   `memcpy(&dst, &src, sizeof(ctype))`

1163 A new, user-defined type `utype` should be destroyed with a call to `GrB_free(utype)` when no longer  
1164 needed.

1165 It is not an error to call this method more than once on the same variable; however, the handle to  
1166 the previously created object will be overwritten.

### 1167 4.2.2.2 UnaryOp\_new: Construct a new GraphBLAS unary operator

1168 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types  
1169 (domains).

## 1170 C Syntax

```
1171                   GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1172                                           void (*unary_func)(void*, const void*),  
1173                                           GrB_Type d_out,  
1174                                           GrB_Type d_in);
```

## 1175 Parameters

1176                   unary\_op (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1177                   unary operator object.

1178                   unary\_func (IN) a pointer to a user-defined function that takes one input parameter of `d_in`'s  
1179                   type and returns a value of `d_out`'s type, both passed as `void` pointers. Specifically  
1180                   the signature of the function is expected to be of the form:

```

1181         void func(void *out, const void *in);
1182

```

1183 **d\_out** (IN) The `GrB_Type` of the return value of the unary operator being created. Should  
1184 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1185 BLAS type.

1186 **d\_in** (IN) The `GrB_Type` of the input argument of the unary operator being created.  
1187 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1188 GraphBLAS type.

## 1189 Return Values

1190 `GrB_SUCCESS` operation completed successfully.

1191 `GrB_PANIC` unknown internal error.

1192 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1193 `GrB_UNINITIALIZED_OBJECT` any `GrB_Type` parameter (for user-defined types) has not been ini-  
1194 tialized by a call to `GrB_Type_new`.

1195 `GrB_NULL_POINTER` `unary_op` or `unary_func` pointers are NULL.

## 1196 Description

1197 The `UnaryOp_new` method creates a new GraphBLAS unary operator

1198  $f_u = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in}), \text{unary\_func} \rangle$

1199 and returns a handle to it in `unary_op`.

1200 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments  
1201 are aliased. In other words, for all invocations of the function:

```

1202     unary_func(out, in);

```

1203 the value of `out` must be the same as if the following code was executed:

```

1204     D(d_in) *tmp = malloc(sizeof(D(d_in)));
1205     memcpy(tmp, in, sizeof(D(d_in)));
1206     unary_func(out, tmp);
1207     free(tmp);

```

1208 It is not an error to call this method more than once on the same variable; however, the handle to  
1209 the previously created object will be overwritten.

### 1210 4.2.2.3 BinaryOp\_new: Construct a new GraphBLAS binary operator

1211 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types  
1212 (domains).

### 1213 C Syntax

```
1214     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
1215                               void          (*binary_func)(void*,  
1216                                                         const void*,  
1217                                                         const void*),  
1218                               GrB_Type      d_out,  
1219                               GrB_Type      d_in1,  
1220                               GrB_Type      d_in2);
```

### 1221 Parameters

1222 binary\_op (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1223 binary operator object.

1224 binary\_func (IN) A pointer to a user-defined function that takes two input parameters of types  
1225 d\_in1 and d\_in2 and returns a value of type d\_out, all passed as void pointers.  
1226 Specifically the signature of the function is expected to be of the form:

```
1227         void func(void *out, const void *in1, const void *in2);
```

1229 d\_out (IN) The GrB\_Type of the return value of the binary operator being created. Should  
1230 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1231 BLAS type.

1232 d\_in1 (IN) The GrB\_Type of the left hand argument of the binary operator being created.  
1233 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1234 GraphBLAS type.

1235 d\_in2 (IN) The GrB\_Type of the right hand argument of the binary operator being cre-  
1236 ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-  
1237 defined GraphBLAS type.

### 1238 Return Values

1239 GrB\_SUCCESS operation completed successfully.

1240 GrB\_PANIC unknown internal error.

1241 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1242 GrB\_UNINITIALIZED\_OBJECT the GrB\_Type (for user-defined types) has not been initialized by a  
1243 call to GrB\_Type\_new.

1244 GrB\_NULL\_POINTER binary\_op or binary\_func pointer is NULL.

## 1245 Description

1246 The BinaryOp\_new method creates a new GraphBLAS binary operator

1247  $f_b = \langle \mathbf{D}(d\_out), \mathbf{D}(d\_in1), \mathbf{D}(d\_in2), \text{binary\_func} \rangle$

1248 and returns a handle to it in binary\_op.

1249 The implementation of binary\_func must be such that it works even if any of the d\_out, d\_in1, and  
1250 d\_in2 arguments are aliased to each other. In other words, for all invocations of the function:

1251 `binary_func(out, in1, in2);`

1252 the value of out must be the same as if the following code was executed:

```
1253     D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));  
1254     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));  
1255     memcpy(tmp1, in1, sizeof(D(d_in1)));  
1256     memcpy(tmp2, in2, sizeof(D(d_in2)));  
1257     binary_func(out, tmp1, tmp2);  
1258     free(tmp2);  
1259     free(tmp1);
```

1260 It is not an error to call this method more than once on the same variable; however, the handle to  
1261 the previously created object will be overwritten.

### 1262 4.2.2.4 Monoid\_new: Construct a new GraphBLAS monoid

1263 Creates a new monoid with specified binary operator and identity value.

## 1264 C Syntax

```
1265     GrB_Info GrB_Monoid_new(GrB_Monoid *monoid,  
1266                             GrB_BinaryOp binary_op,  
1267                             <type> identity);
```

## 1268 Parameters

1269        **monoid** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1270        monoid object.

1271        **binary\_op** (IN) An existing GraphBLAS associative binary operator whose input and output  
1272        types are the same.

1273        **identity** (IN) The value of the identity element of the monoid. Must be the same type as  
1274        the type used by the **binary\_op** operator.

## 1275 Return Values

1276        **GrB\_SUCCESS** operation completed successfully.

1277        **GrB\_PANIC** unknown internal error.

1278        **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1279 **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_BinaryOp** (for user-defined operators) has not been initial-  
1280        ized by a call to **GrB\_BinaryOp\_new**.

1281        **GrB\_NULL\_POINTER** monoid pointer is NULL.

1282        **GrB\_DOMAIN\_MISMATCH** all three argument types of the binary operator and the type of the  
1283        identity value are not the same.

## 1284 Description

1285 The **Monoid\_new** method creates a new monoid  $M = \langle \mathbf{D}(\text{binary\_op}), \text{binary\_op}, \text{identity} \rangle$  and re-  
1286 turns a handle to it in **monoid**.

1287 If **binary\_op** is not associative, the results of GraphBLAS operations that require associativity of  
1288 this monoid will be undefined.

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

### 1291 4.2.2.5 Semiring\_new: Construct a new GraphBLAS semiring

1292 Creates a new semiring with specified domain, operators, and elements.

## 1293 C Syntax

```
1294        GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1295                                    GrB_Monoid     add_op,  
1296                                    GrB_BinaryOp   mul_op);
```

## 1297 Parameters

- 1298        **semiring** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1299        semiring.
- 1300        **add\_op** (IN) An existing GraphBLAS commutative monoid that specifies the addition op-  
1301        erator and its identity.
- 1302        **mul\_op** (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-  
1303        plication operator. In addition, **mul\_op**'s output domain,  $\mathbf{D}_{out}(\text{mul\_op})$ , must be  
1304        the same as the **add\_op**'s domain  $\mathbf{D}(\text{add\_op})$ .

## 1305 Return Values

- 1306        **GrB\_SUCCESS** operation completed successfully.
- 1307        **GrB\_PANIC** unknown internal error.
- 1308        **GrB\_OUT\_OF\_MEMORY** not enough memory available for this method to complete.
- 1309 **GrB\_UNINITIALIZED\_OBJECT** the **add\_op** (for user-define monoids) object has not been initialized  
1310        with a call to **GrB\_Monoid\_new** or the **mul\_op** (for user-defined  
1311        operators) object has not been not been initialized by a call to  
1312        **GrB\_BinaryOp\_new**.
- 1313        **GrB\_NULL\_POINTER** semiring pointer is NULL.
- 1314        **GrB\_DOMAIN\_MISMATCH** the output domain of **mul\_op** does not match the domain of the  
1315        **add\_op** monoid.

## 1316 Description

1317 The **Semiring\_new** method creates a new semiring:

$$1318 \quad S = \langle \mathbf{D}_{out}(\text{mul\_op}), \mathbf{D}_{in_1}(\text{mul\_op}), \mathbf{D}_{in_2}(\text{mul\_op}), \text{add\_op}, \text{mul\_op}, \mathbf{0}(\text{add\_op}) \rangle$$

1319 and returns a handle to it in **semiring**. Note that  $\mathbf{D}_{out}(\text{mul\_op})$  must be the same as  $\mathbf{D}(\text{add\_op})$ .

1320 If **add\_op** is not commutative, then GraphBLAS operations using this semiring will be undefined.

1321 It is not an error to call this method more than once on the same variable; however, the handle to  
1322 the previously created object will be overwritten.

### 1323 4.2.2.6 IndexUnaryOp\_new: Construct a new GraphBLAS index unary operator [Scott: 1324 NEW CONTENT]

1325 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its  
1326 types (domains).

## 1327 C Syntax

```
1328     GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp  *index_unary_op,  
1329                                   void (*index_unary_func)(void*,  
1330                                                             const void*,  
1331                                                             GrB_Index,  
1332                                                             GrB_Index,  
1333                                                             const void*),  
1334                                   GrB_Type           d_out,  
1335                                   GrB_Type           d_in1,  
1336                                   GrB_Type           d_in2);
```

## 1337 Parameters

1338 **index\_unary\_op** (INOUT) On successful return, contains a handle to the newly created Graph-  
1339 BLAS index unary operator object.

1340 **index\_unary\_func** (IN) A pointer to a user-defined function that takes input parameters of types  
1341 **d\_in1**, **GrB\_Index**, **GrB\_Index** and **d\_in2** and returns a value of type **d\_out**. Ex-  
1342 cept for the **GrB\_Index** parameters, all are passed as **void** pointers. Specifically  
1343 the signature of the function is expected to be of the form:

```
1344         void func(void      *out,  
1345                   const void *in1,  
1346                   GrB_Index  row_index,  
1347                   GrB_Index  col_index,  
1348                   const void *in2);  
1349
```

1350 **d\_out** (IN) The **GrB\_Type** of the return value of the index unary operator being created.  
1351 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1352 GraphBLAS type.

1353 **d\_in1** (IN) The **GrB\_Type** of the first input argument of the index unary operator being  
1354 created and corresponds to the stored values of the **GrB\_Vector** or **GrB\_Matrix**  
1355 being operated on. Should be one of the predefined GraphBLAS types in Ta-  
1356 ble 3.2, or a user-defined GraphBLAS type.

1357 **d\_in2** (IN) The **GrB\_Type** of the last input argument of the index unary operator be-  
1358 ing created and corresponds to a scalar provided by the GraphBLAS operation  
1359 that uses this operator. Should be one of the predefined GraphBLAS types in  
1360 Table 3.2, or a user-defined GraphBLAS type.

## 1361 Return Values

1362 **GrB\_SUCCESS** operation completed successfully.





## 1391 C Syntax

```
1392         GrB_Info GrB_Scalar_new(GrB_Scalar *s,  
1393                                 GrB_Type    d);
```

## 1394 Parameters

1395 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1396 scalar.

1397 **d** (IN) The type corresponding to the domain of the scalar being created. Can be  
1398 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1399 GraphBLAS type.

## 1400 Return Values

1401 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1402 blocking mode, this indicates that the API checks for the input  
1403 arguments passed successfully. Either way, output scalar **s** is ready  
1404 to be used in the next method of the sequence.

1405 **GrB\_PANIC** Unknown internal error.

1406 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1407 GraphBLAS objects (input or output) is in an invalid state caused  
1408 by a previous execution error. Call **GrB\_error()** to access any error  
1409 messages generated by the implementation.

1410 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1411 **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
1412 (needed for user-defined types).

1413 **GrB\_NULL\_POINTER** The **s** pointer is NULL.

## 1414 Description

1415 Creates a new GraphBLAS scalar **s** of domain **D(d)** and empty **L(s)**. The method returns a handle  
1416 to the new scalar in **s**.

1417 It is not an error to call this method more than once on the same variable; however, the handle to  
1418 the previously created object will be overwritten.

### 1419 4.2.3.2 Scalar\_dup: Construct a copy of a GraphBLAS scalar

1420 Creates a new scalar with the same domain and contents as another scalar.

## 1421 C Syntax

```
1422         GrB_Info GrB_Scalar_dup(GrB_Scalar      *t,  
1423                                 const GrB_Scalar  s);
```

## 1424 Parameters

1425 **t** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1426 scalar.

1427 **s** (IN) The GraphBLAS scalar to be duplicated.

## 1428 Return Values

1429 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1430 blocking mode, this indicates that the API checks for the input  
1431 arguments passed successfully. Either way, output scalar **t** is ready  
1432 to be used in the next method of the sequence.

1433 **GrB\_PANIC** Unknown internal error.

1434 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1435 GraphBLAS objects (input or output) is in an invalid state caused  
1436 by a previous execution error. Call **GrB\_error()** to access any error  
1437 messages generated by the implementation.

1438 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1439 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to  
1440 **Scalar\_new** or **Scalar\_dup**.

1441 **GrB\_NULL\_POINTER** The **t** pointer is NULL.

## 1442 Description

1443 Creates a new scalar *t* of domain **D(s)** and contents **L(s)**. The method returns a handle to the new  
1444 scalar in **t**.

1445 It is not an error to call this method more than once with the same output variable; however, the  
1446 handle to the previously created object will be overwritten.

### 1447 4.2.3.3 Scalar\_clear: Clear/remove a stored value from a scalar

1448 Removes the stored value from a scalar.

## 1449 C Syntax

```
1450      GrB_Info GrB_Scalar_clear(GrB_Scalar s);
```

## 1451 Parameters

1452 `s` (INOUT) An existing GraphBLAS scalar to clear.

## 1453 Return Values

1454 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
1455 blocking mode, this indicates that the API checks for the input  
1456 arguments passed successfully. Either way, output scalar `s` is ready  
1457 to be used in the next method of the sequence.

1458 `GrB_PANIC` Unknown internal error.

1459 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
1460 GraphBLAS objects (input or output) is in an invalid state caused  
1461 by a previous execution error. Call `GrB_error()` to access any error  
1462 messages generated by the implementation.

1463 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1464 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to  
1465 `Scalar_new` or `Scalar_dup`.

## 1466 Description

1467 Removes the stored value from an existing scalar. After the call, `L(s)` is empty. The size of the  
1468 scalar does not change.

## 1469 4.2.3.4 `Scalar_nvals`: Number of stored elements in a scalar

1470 Retrieve the number of stored elements in a scalar (either zero or one).

## 1471 C Syntax

```
1472      GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,  
1473                                const GrB_Scalar s);
```

## 1474 Parameters

1475           nvals (OUT) On successful return, this is set to the number of stored elements in the  
1476           scalar (zero or one).

1477 s (IN) An existing GraphBLAS scalar being queried.

## 1478 Return Values

```

1479         GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1480         cessfully and the value of nvals has been set.

```

1481 GrB PANIC Unknown internal error.

1482	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the opaque
1483		GraphBLAS objects (input or output) is in an invalid state caused
1484		by a previous execution error. Call <b>GrB_error()</b> to access any error
1485		messages generated by the implementation.

```
1486 GrB_OUT_OF_MEMORY Not enough memory available for operation.
```

1487 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, s, has not been initialized by a call to  
1488 Scalar\_new or Scalar\_dup.

1489 GrB\_NULL\_POINTER The nvals pointer is NULL.

1490 **Description**

1491 Return **nvals**(s) in **nvals**. This is the number of stored elements in scalar s, which is the size of  
1492 **L**(s), and can only be either zero or one (see Section 3.5.1).

1493 **4.2.3.5** `Scalar_setElement`: Set the single element in a scalar

1494 Set the single element of a scalar to a given value.

1495 **C Syntax**

```

1496         GrB_Info GrB_Scalar_setElement(GrB_Scalar s,
1497                                         <type> val);

```

1498 **Parameters**

1499 s (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1500 **val** (IN) Scalar value to assign. The type must be compatible with the domain of **s**.

## 1501 Return Values

1502           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1503                       blocking mode, this indicates that the compatibility tests on in-  
1504                       dex/dimensions and domains for the input arguments passed suc-  
1505                       cessfully. Either way, the output scalar `s` is ready to be used in the  
1506                       next method of the sequence.

1507           GrB\_PANIC Unknown internal error.

1508           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1509                       GraphBLAS objects (input or output) is in an invalid state caused  
1510                       by a previous execution error. Call `GrB_error()` to access any error  
1511                       messages generated by the implementation.

1512           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1513 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, `s`, has not been initialized by a call to  
1514                       Scalar\_new or Scalar\_dup.

1515           GrB\_DOMAIN\_MISMATCH The domains of `s` and `val` are incompatible.

## 1516 Description

1517 First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: `D(val)` must  
1518 be compatible with `D(s)`. Two domains are compatible with each other if values from one domain  
1519 can be cast to values in the other domain as per the rules of the C language. In particular, domains  
1520 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-  
1521 patible with itself. If any compatibility rule above is violated, execution of `GrB_Scalar_setElement`  
1522 ends and the domain mismatch error listed above is returned.

1523 We are now ready to carry out the assignment `val`; that is:

$$1524 \qquad s(0) = \text{val}$$

1525 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1526 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1527 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1528 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be  
1529 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1530 4.2.3.6 Scalar\_extractElement: Extract a single element from a scalar.

1531 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

## 1532 C Syntax

```
1533      GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1534                                         const GrB_Scalar s);
```

## 1535 Parameters

1536 **val** (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain  
1537 of scalar **s**. On successful return, **val** holds the result of the operation, and any  
1538 previous value in **val** is overwritten.

1539 **s** (IN) The GraphBLAS scalar from which an element is extracted.

## 1540 Return Values

1541 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1542 cessfully. This indicates that the compatibility tests on dimensions  
1543 and domains for the input arguments passed successfully, and the  
1544 output scalar, **val**, has been computed and is ready to be used in  
1545 the next method of the sequence.

1546 **GrB\_PANIC** Unknown internal error.

1547 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1548 GraphBLAS objects (input or output) is in an invalid state caused  
1549 by a previous execution error. Call **GrB\_error()** to access any error  
1550 messages generated by the implementation.

1551 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1552 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to  
1553 **Scalar\_new** or **Scalar\_dup**.

1554 **GrB\_NULL\_POINTER** **val** pointer is NULL.

1555 **GrB\_DOMAIN\_MISMATCH** The domains of the scalar or scalar are incompatible.

1556 **GrB\_NO\_VALUE** There is no stored value in the scalar.

## 1557 Description

1558 First, **val** and input GraphBLAS scalar are tested for domain compatibility as follows: **D(val)**  
1559 must be compatible with **D(s)**. Two domains are compatible with each other if values from  
1560 one domain can be cast to values in the other domain as per the rules of the C language. In  
1561 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1562 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1563 **GrB\_Scalar\_extractElement** ends and the domain mismatch error listed above is returned.

1564 Then, if no value is currently stored in the GraphBLAS scalar, the method returns `GrB_NO_VALUE`  
1565 and `val` remains unchanged.

1566 Finally the extract into the output argument, `val` can be performed; that is:

1567 
$$\text{val} = \text{s}(0)$$

1568 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value  
1569 `GrB_SUCCESS`, the new contents of `val` are as defined above.

## 1570 4.2.4 Vector methods

### 1571 4.2.4.1 Vector\_new: Construct new vector

1572 Creates a new vector with specified domain and size.

#### 1573 C Syntax

```
1574 GrB_Info GrB_Vector_new(GrB_Vector *v,  
1575                          GrB_Type    d,  
1576                          GrB_Index   nsize);
```

#### 1577 Parameters

1578 `v` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1579 vector.

1580 `d` (IN) The type corresponding to the domain of the vector being created. Can be  
1581 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1582 GraphBLAS type.

1583 `nsize` (IN) The size of the vector being created.

#### 1584 Return Values

1585 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
1586 blocking mode, this indicates that the API checks for the input  
1587 arguments passed successfully. Either way, output vector `v` is ready  
1588 to be used in the next method of the sequence.

1589 `GrB_PANIC` Unknown internal error.

1590 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
1591 GraphBLAS objects (input or output) is in an invalid state caused  
1592 by a previous execution error. Call `GrB_error()` to access any error  
1593 messages generated by the implementation.

1594       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1595 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
1596                               (nEEDED for user-defined types).

1597       GrB\_NULL\_POINTER The v pointer is NULL.

1598       GrB\_INVALID\_VALUE nsize is zero or outside the range of the type GrB\_Index.

## 1599 Description

1600 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  
1601 to the new vector in v.

1602 It is not an error to call this method more than once on the same variable; however, the handle to  
1603 the previously created object will be overwritten.

### 1604 4.2.4.2 Vector\_dup: Construct a copy of a GraphBLAS vector

1605 Creates a new vector with the same domain, size, and contents as another vector.

## 1606 C Syntax

```
1607         GrB_Info GrB_Vector_dup(GrB_Vector      *w,
1608                                const GrB_Vector  u);
```

## 1609 Parameters

1610       w (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1611       vector.

1612       u (IN) The GraphBLAS vector to be duplicated.

## 1613 Return Values

1614       GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1615       blocking mode, this indicates that the API checks for the input  
1616       arguments passed successfully. Either way, output vector w is ready  
1617       to be used in the next method of the sequence.

1618       GrB\_PANIC Unknown internal error.

1619       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1620       GraphBLAS objects (input or output) is in an invalid state caused  
1621       by a previous execution error. Call GrB\_error() to access any error  
1622       messages generated by the implementation.



1623        GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1624 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector,  $u$ , has not been initialized by a call to  
1625        Vector\_new or Vector\_dup.

1626        GrB\_NULL\_POINTER The  $w$  pointer is NULL.

## 1627 Description

1628 Creates a new vector  $w$  of domain  $D(u)$ , size  $size(u)$ , and contents  $L(u)$ . The method returns a  
1629 handle to the new vector in  $w$ .

1630 It is not an error to call this method more than once on the same variable; however, the handle to  
1631 the previously created object will be overwritten.

### 1632 4.2.4.3 Vector\_resize: Resize a vector

1633 Changes the size of an existing vector.

## 1634 C Syntax

```
1635        GrB_Info GrB_Vector_resize(GrB_Vector w,  
1636                                    GrB_Index nsize);
```

## 1637 Parameters

1638         $w$  (INOUT) An existing Vector object that is being resized.

1639         $nsize$  (IN) The new size of the vector. It can be smaller or larger than the current size.

## 1640 Return Values

1641        GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1642        blocking mode, this indicates that the API checks for the input  
1643        arguments passed successfully. Either way, output vector  $w$  is ready  
1644        to be used in the next method of the sequence.

1645        GrB\_PANIC Unknown internal error.

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

1650        GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1651           GrB\_NULL\_POINTER The  $w$  pointer is NULL.

1652           GrB\_INVALID\_VALUE  $nsz$  is zero or outside the range of the type GrB\_Index.

## 1653 Description

1654 Changes the size of  $w$  to  $nsz$ . The domain  $\mathbf{D}(w)$  of vector  $w$  remains the same. The contents  $\mathbf{L}(w)$   
1655 are modified as described below.

1656 Let  $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$  when the method is called. When the method returns,  $w = \langle \mathbf{D}(w), nsz, \mathbf{L}'(w) \rangle$   
1657 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  
1658 than or equal to the new vector size ( $nsz$ ) are dropped.

### 1659 4.2.4.4 Vector\_clear: Clear a vector

1660 Removes all the elements (tuples) from a vector.

## 1661 C Syntax

1662           GrB\_Info GrB\_Vector\_clear(GrB\_Vector v);

## 1663 Parameters

1664            $v$  (INOUT) An existing GraphBLAS vector to clear.

## 1665 Return Values

1666           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1667 blocking mode, this indicates that the API checks for the input  
1668 arguments passed successfully. Either way, output vector  $v$  is ready  
1669 to be used in the next method of the sequence.

1670           GrB\_PANIC Unknown internal error.

1671           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1672 GraphBLAS objects (input or output) is in an invalid state caused  
1673 by a previous execution error. Call GrB\_error() to access any error  
1674 messages generated by the implementation.

1675           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1676           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector,  $v$ , has not been initialized by a call to  
1677 Vector\_new or Vector\_dup.

## 1678 Description

1679 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,  
1680  $L(v) = \emptyset$ . The size of the vector does not change.

### 1681 4.2.4.5 Vector\_size: Size of a vector

1682 Retrieve the size of a vector.

## 1683 C Syntax

```
1684      GrB_Info GrB_Vector_size(GrB_Index      *nsize,  
1685                             const GrB_Vector v);
```

## 1686 Parameters

1687 nsize (OUT) On successful return, is set to the size of the vector.

1688 v (IN) An existing GraphBLAS vector being queried.

## 1689 Return Values

1690 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1691 cessfully and the value of `nsize` has been set.

1692 GrB\_PANIC Unknown internal error.

1693 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1694 GraphBLAS objects (input or output) is in an invalid state caused  
1695 by a previous execution error. Call `GrB_error()` to access any error  
1696 messages generated by the implementation.

1697 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to  
1698 `Vector_new` or `Vector_dup`.

1699 GrB\_NULL\_POINTER `nsize` pointer is NULL.

## 1700 Description

1701 Return `size(v)` in `nsize`.

### 1702 4.2.4.6 Vector\_nvals: Number of stored elements in a vector

1703 Retrieve the number of stored elements (tuples) in a vector.

## 1704 C Syntax

```
1705         GrB_Info GrB_Vector_nvals(GrB_Index      *nvals,  
1706                                   const GrB_Vector v);
```

## 1707 Parameters

1708        **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)  
1709        in the vector.

1710        **v** (IN) An existing GraphBLAS vector being queried.

## 1711 Return Values

1712        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1713        cessfully and the value of **nvals** has been set.

1714        **GrB\_PANIC** Unknown internal error.

1715        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1716        GraphBLAS objects (input or output) is in an invalid state caused  
1717        by a previous execution error. Call **GrB\_error()** to access any error  
1718        messages generated by the implementation.

1719        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1720        **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to  
1721        **Vector\_new** or **Vector\_dup**.

1722        **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

## 1723 Description

1724        Return **nvals(v)** in **nvals**. This is the number of stored elements in vector **v**, which is the size of  
1725        **L(v)** (see Section 3.5.2).

### 1726 4.2.4.7 Vector\_build: Store elements from tuples into a vector

## 1727 C Syntax

```
1728         GrB_Info GrB_Vector_build(GrB_Vector      w,  
1729                                   const GrB_Index *indices,  
1730                                   const <type>    *values,  
1731                                   GrB_Index        n,  
1732                                   const GrB_BinaryOp dup);
```

## 1733 Parameters

- 1734            **w** (INOUT) An existing Vector object to store the result.
- 1735            **indices** (IN) Pointer to an array of indices.
- 1736            **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
1737            vector **w**.
- 1738            **n** (IN) The number of entries contained in each array (the same for **indices** and **values**).
- 1739            **dup** (IN) An associative and commutative binary operator to apply when duplicate  
1740            values for the same location are present in the input arrays. All three domains of  
1741            **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
1742            then duplicate locations will result in an error.

## 1743 Return Values

- 1744            **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1745            blocking mode, this indicates that the API checks for the input  
1746            arguments passed successfully. Either way, output vector **w** is  
1747            ready to be used in the next method of the sequence.
- 1748            **GrB\_PANIC** Unknown internal error.
- 1749            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
1750            opaque GraphBLAS objects (input or output) is in an invalid  
1751            state caused by a previous execution error. Call **GrB\_error()** to  
1752            access any error messages generated by the implementation.
- 1753            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.
- 1754            **GrB\_UNINITIALIZED\_OBJECT** Either **w** has not been initialized by a call to **GrB\_Vector\_new**  
1755            or by **GrB\_Vector\_dup**, or **dup** has not been initialized by a call  
1756            to **GrB\_BinaryOp\_new**.
- 1757            **GrB\_NULL\_POINTER** **indices** or **values** pointer is **NULL**.
- 1758            **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indices** is outside the allowed range for **w**.
- 1759            **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
1760            not all the same, or the domains of **values** and **w** are incompatible  
1761            with each other or  $D_{dup}$ .
- 1762            **GrB\_OUTPUT\_NOT\_EMPTY** Output vector **w** already contains valid tuples (elements). In  
1763            other words, **GrB\_Vector\_nvals(C)** returns a positive value.
- 1764            **GrB\_INVALID\_VALUE** **indices** contains a duplicate location and **dup** is **GrB\_NULL**.

## 1765 Description

1766 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  
 1767 from  $\mathbf{w}$  in its domain; otherwise,  $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$ .

1768 Each tuple  $\{\text{indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the output in the form of

$$1769 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1770 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
 1771 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{w}}$  as follows:

$$1772 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1773 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{w}}$  is copied into  $\mathbf{w}$  via typecasting its  
 1774 values to  $\mathbf{D}(\mathbf{w})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

1775 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1776 It is an error to call this function on an output object with existing elements. In other words,  
 1777 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1778 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`  
 1779 or `values`.

### 1780 4.2.4.8 Vector\_setElement: Set a single element in a vector

1781 Set one element of a vector to a given value.

## 1782 C Syntax

```
1783 // scalar value
1784 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1785                               <type>         val,
1786                               GrB_Index       index);
1787
1788 // GraphBLAS scalar
1789 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1790                               const GrB_Scalar s,
1791                               GrB_Index       index);
```

## 1792 Parameters

1793 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1794            **val** or **s** (IN) Scalar assign. Its domain (type) must be compatible with the domain of **w**.  
 1795            **index** (IN) The location of the element to be assigned.

## 1796 **Return Values**

1797            **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 1798            blocking mode, this indicates that the compatibility tests on in-  
 1799            dex/dimensions and domains for the input arguments passed suc-  
 1800            cessfully. Either way, the output vector **w** is ready to be used in  
 1801            the next method of the sequence.

1802            **GrB\_PANIC** Unknown internal error.

1803            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 1804            GraphBLAS objects (input or output) is in an invalid state caused  
 1805            by a previous execution error. Call **GrB\_error()** to access any error  
 1806            messages generated by the implementation.

1807            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1808            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been  
 1809            initialized by a call to a respective constructor.

1810            **GrB\_INVALID\_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1811            **GrB\_DOMAIN\_MISMATCH** The domains of the vector and the scalar are incompatible.

## 1812 **Description**

1813            First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
 1814            must be compatible with **D(w)**. Two domains are compatible with each other if values from  
 1815            one domain can be cast to values in the other domain as per the rules of the C language. In  
 1816            particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 1817            defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 1818            **GrB\_Vector\_setElement** ends and the domain mismatch error listed above is returned.

1819            Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1820 \qquad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1821            If this condition is violated, execution of **GrB\_Vector\_setElement** ends and the invalid index error  
 1822            listed above is returned.

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

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

1823 In the case of a transparent scalar or if  $\mathbf{L}(\mathbf{s})$  is not empty, then a value will be stored at the  
 1824 specified location in  $\mathbf{w}$ , overwriting any value that may have been stored there before. In the case  
 1825 of a GraphBLAS scalar, if  $\mathbf{L}(\mathbf{s})$  is empty, then any value stored at the specified location in  $\mathbf{w}$  will  
 1826 be removed.

1827 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 1828 of  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 1829 return value GrB\_SUCCESS and the new contents of vector  $\mathbf{w}$  is as defined above but may not be  
 1830 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1831 4.2.4.9 Vector\_removeElement: Remove an element from a vector

1832 Remove (annihilate) one stored element from a vector.

#### 1833 C Syntax

```
1834      GrB_Info GrB_Vector_removeElement(GrB_Vector  w,
1835                                     GrB_Index   index);
```

#### 1836 Parameters

1837  $\mathbf{w}$  (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1838  $\mathbf{index}$  (IN) The location of the element to be removed.

#### 1839 Return Values

1840 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 1841 blocking mode, this indicates that the compatibility tests on in-  
 1842 dex/dimensions and domains for the input arguments passed suc-  
 1843 cessfully. Either way, the output vector  $\mathbf{w}$  is ready to be used in  
 1844 the next method of the sequence.

1845 GrB\_PANIC Unknown internal error.

1846 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1847 GraphBLAS objects (input or output) is in an invalid state caused  
 1848 by a previous execution error. Call GrB\_error() to access any error  
 1849 messages generated by the implementation.

1850 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1851 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector,  $\mathbf{w}$ , has not been initialized by a call to  
 1852 Vector\_new or Vector\_dup.

1853 GrB\_INVALID\_INDEX  $\mathbf{index}$  specifies a location that is outside the dimensions of  $\mathbf{w}$ .



## 1854 Description

1855 First, the `index` parameter is checked for a valid value where the following condition must hold:

$$1856 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1857 If this condition is violated, execution of `GrB_Vector_removeElement` ends and the invalid index  
1858 error listed above is returned.

1859 We are now ready to carry out the removal of a value that may be stored at the location specified  
1860 by `index`. If a value does not exist at the specified location in  $\mathbf{w}$ , no error is reported and the  
1861 operation has no effect on the state of  $\mathbf{w}$ . In either case, the following will be true on return from  
1862 the method: `index`  $\notin$  `ind(w)`.

1863 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1864 of  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1865 return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may not be  
1866 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1867 4.2.4.10 Vector\_extractElement: Extract a single element from a vector.

1868 Extract one element of a vector into a scalar.

## 1869 C Syntax

```
1870 // scalar value
1871 GrB_Info GrB_Vector_extractElement(<type>          *val,
1872                                   const GrB_Vector u,
1873                                   GrB_Index         index);
1874
1875 // GraphBLAS scalar
1876 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1877                                   const GrB_Vector u,
1878                                   GrB_Index         index);
```

## 1879 Parameters

1880 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector  
1881 `u`. On successful return, this scalar holds the result of the extract. Any previous  
1882 value stored in `val` or `s` is overwritten.

1883 `u` (IN) The GraphBLAS vector from which an element is extracted.

1884 `index` (IN) The location in `u` to extract.

## 1885 Return Values

1886           GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 1887                           cessfully. This indicates that the compatibility tests on dimensions  
 1888                           and domains for the input arguments passed successfully, and the  
 1889                           output scalar, **val** or **s**, has been computed and is ready to be used  
 1890                           in the next method of the sequence.

1891           GrB\_NO\_VALUE When using the transparent scalar, **val**, this is returned when there  
 1892                           is no stored value at specified location.

1893           GrB\_PANIC Unknown internal error.

1894           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1895                           GraphBLAS objects (input or output) is in an invalid state caused  
 1896                           by a previous execution error. Call **GrB\_error()** to access any error  
 1897                           messages generated by the implementation.

1898           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1899 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, **u**, or scalar, **s**, has not been initialized by  
 1900                           a call to a corresponding constructor.

1901           GrB\_NULL\_POINTER **val** pointer is NULL.

1902           GrB\_INVALID\_INDEX **index** specifies a location that is outside the dimensions of **w**.

1903           GrB\_DOMAIN\_MISMATCH The domains of the vector and scalar are incompatible.

## 1904 Description

1905 First, the scalar and input vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
 1906 must be compatible with **D(u)**. Two domains are compatible with each other if values from  
 1907 one domain can be cast to values in the other domain as per the rules of the C language. In  
 1908 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 1909 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 1910 **GrB\_Vector\_extractElement** ends and the domain mismatch error listed above is returned.

1911 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1912 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1913 If this condition is violated, execution of **GrB\_Vector\_extractElement** ends and the invalid index  
 1914 error listed above is returned.

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

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

1915 If  $\text{index} \in \mathbf{ind}(u)$ , then the corresponding value from  $u$  is copied into  $s$  or  $val$  with casting as  
 1916 necessary. If  $\text{index} \notin \mathbf{ind}(u)$ , then one of the follow occurs depending on output scalar type:

- 1917 • The GraphBLAS scalar,  $s$ , is cleared and `GrB_SUCCESS` is returned.
- 1918 • The non-opaque scalar,  $val$ , is unchanged, and `GrB_NO_VALUE` is returned.

1919 When using the non-opaque scalar variant ( $val$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 1920 mode, the new contents of  $val$  are as defined above if the method exits with return value `GrB_SUCCESS`  
 1921 or `GrB_NO_VALUE`.

1922 When using the GraphBLAS scalar variant ( $s$ ) with a `GrB_SUCCESS` return value, the method  
 1923 exits and the new contents of  $s$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
 1924 In `GrB_NONBLOCKING` mode, the new contents of  $s$  is as defined above but may not be fully  
 1925 computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1926 4.2.4.11 Vector\_extractTuples: Extract tuples from a vector

1927 Extract the contents of a GraphBLAS vector into non-opaque data structures.

#### 1928 C Syntax

```
1929      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,
1930                                     <type>          *values,
1931                                     GrB_Index        *n,
1932                                     const GrB_Vector  v);
1933
```

1934 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored  
 1935 values' indices.

1936 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
 1937 the stored values whose type is compatible with  $\mathbf{D}(v)$ .

1938 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the  
 1939 values and indices arrays can hold. Upon return, it will contain the number of  
 1940 values written to the arrays.

1941 **v** (IN) An existing GraphBLAS vector.

#### 1942 Return Values

1943 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
 1944 cessfully. This indicates that the compatibility tests on the input  
 1945 argument passed successfully, and the output arrays, **indices** and  
 1946 **values**, have been computed.

1947                   GrB\_PANIC Unknown internal error.

1948           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1949                   GraphBLAS objects (input or output) is in an invalid state caused  
1950                   by a previous execution error. Call `GrB_error()` to access any error  
1951                   messages generated by the implementation.

1952           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1953           GrB\_INSUFFICIENT\_SPACE Not enough space in `indices` and `values` (as indicated by the `n` pa-  
1954                   rameter) to hold all of the tuples that will be extracted.

1955 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to  
1956                   Vector\_new or Vector\_dup.

1957           GrB\_NULL\_POINTER `indices`, `values`, or `n` pointer is NULL.

1958           GrB\_DOMAIN\_MISMATCH The domains of the `v` vector or `values` array are incompatible with  
1959                   one another.

## 1960 Description

1961 This method will extract all the tuples from the GraphBLAS vector `v`. The values associated  
1962 with those tuples are placed in the `values` array and the indices are placed in the `indices` array.  
1963 Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least  
1964 `GrB_Vector_nvals(v)` elements before calling this function.

1965 Upon return of this function, `n` will be set to the number of values (and indices) copied. Also, the  
1966 entries of `indices` are unique, but not necessarily sorted. Each tuple  $(i, v_i)$  in `v` is unzipped and  
1967 copied into a distinct  $k$ th location in output vectors:

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

1968 where  $0 \leq k < \text{GrB\_Vector\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if `indices[k]`  
1969 and `values[k]` exist upon return, so does `indices[j]` and `values[j]` for all  $j$  such that  $0 \leq j < k$ .

1970 Note that if the value in `n` on input is less than the number of values contained in the vector `v`,  
1971 then a `GrB_INSUFFICIENT_SPACE` error is returned because it is undefined which subset of values  
1972 would be extracted otherwise.

1973 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value  
1974 `GrB_SUCCESS`, the new contents of the arrays `indices` and `values` are as defined above.

## 1975 4.2.5 Matrix methods

### 1976 4.2.5.1 Matrix\_new: Construct new matrix

1977 Creates a new matrix with specified domain and dimensions.

## 1978 C Syntax

```
1979         GrB_Info GrB_Matrix_new(GrB_Matrix *A,  
1980                                 GrB_Type      d,  
1981                                 GrB_Index     nrows,  
1982                                 GrB_Index     ncols);
```

## 1983 Parameters

1984 **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1985 matrix.

1986 **d** (IN) The type corresponding to the domain of the matrix being created. Can be  
1987 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1988 GraphBLAS type.

1989 **nrows** (IN) The number of rows of the matrix being created.

1990 **ncols** (IN) The number of columns of the matrix being created.

## 1991 Return Values

1992 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1993 blocking mode, this indicates that the API checks for the input ar-  
1994 guments passed successfully. Either way, output matrix **A** is ready  
1995 to be used in the next method of the sequence.

1996 **GrB\_PANIC** Unknown internal error.

1997 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1998 GraphBLAS objects (input or output) is in an invalid state caused  
1999 by a previous execution error. Call **GrB\_error()** to access any error  
2000 messages generated by the implementation.

2001 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2002 **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2003 (needed for user-defined types).

2004 **GrB\_NULL\_POINTER** The **A** pointer is **NULL**.

2005 **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

## 2006 Description

2007 Creates a new matrix **A** of domain **D**(**d**), size **nrows**  $\times$  **ncols**, and empty **L**(**A**). The method returns  
2008 a handle to the new matrix in **A**.

2009 It is not an error to call this method more than once on the same variable; however, the handle to  
2010 the previously created object will be overwritten.

#### 2011 **4.2.5.2 Matrix\_dup: Construct a copy of a GraphBLAS matrix**

2012 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

#### 2013 **C Syntax**

```
2014         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
2015                                const GrB_Matrix  A);
```

#### 2016 **Parameters**

2017 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2018 matrix.

2019 A (IN) The GraphBLAS matrix to be duplicated.

#### 2020 **Return Values**

2021 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2022 blocking mode, this indicates that the API checks for the input  
2023 arguments passed successfully. Either way, output matrix C is ready  
2024 to be used in the next method of the sequence.

2025 GrB\_PANIC Unknown internal error.

2026 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2027 GraphBLAS objects (input or output) is in an invalid state caused  
2028 by a previous execution error. Call GrB\_error() to access any error  
2029 messages generated by the implementation.

2030 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2031 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2032 any matrix constructor.

2033 GrB\_NULL\_POINTER The C pointer is NULL.

#### 2034 **Description**

2035 Creates a new matrix **C** of domain **D(A)**, size **nrows(A) × ncols(A)**, and contents **L(A)**. It returns  
2036 a handle to it in C.

2037 It is not an error to call this method more than once on the same variable; however, the handle to  
2038 the previously created object will be overwritten.

#### 2039 4.2.5.3 Matrix\_diag: Construct a diagonal GraphBLAS matrix

2040 Creates a new matrix with the same domain and contents as a GrB\_Vector, and square dimensions  
2041 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

#### 2042 C Syntax

```
2043         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
2044                                 const GrB_Vector  v,  
2045                                 int64_t           k);
```

#### 2046 Parameters

2047 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2048 matrix. The matrix is square with each dimension equal to  $\text{size}(\mathbf{v}) + |k|$ .

2049 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the  
2050 matrix.

2051 k (IN) The diagonal to which the vector is assigned.  $k = 0$  represents the main  
2052 diagonal,  $k > 0$  is above the main diagonal, and  $k < 0$  is below.

#### 2053 Return Values

2054 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2055 blocking mode, this indicates that the API checks for the input  
2056 arguments passed successfully. Either way, output matrix C is ready  
2057 to be used in the next method of the sequence.

2058 GrB\_PANIC Unknown internal error.

2059 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2060 GraphBLAS objects (input or output) is in an invalid state caused  
2061 by a previous execution error. Call GrB\_error() to access any error  
2062 messages generated by the implementation.

2063 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

2064 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, v, has not been initialized by a call to  
2065 Vector\_new or Vector\_dup.

2066 GrB\_NULL\_POINTER The C pointer is NULL.

## 2067 Description

2068 Creates a new matrix **C** of domain **D(v)**, size  $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$ , and contents

$$\begin{aligned} 2069 \quad \mathbf{L}(\mathbf{C}) &= \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or} \\ 2070 \quad \mathbf{L}(\mathbf{C}) &= \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0. \end{aligned}$$

2071 It returns a handle to it in **C**. It is not an error to call this method more than once on the same  
2072 variable; however, the handle to the previously created object will be overwritten.

## 2073 4.2.5.4 Matrix\_resize: Resize a matrix

2074 Changes the dimensions of an existing matrix.

## 2075 C Syntax

```
2076      GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
2077                                GrB_Index  nrows,  
2078                                GrB_Index  ncols);
```

## 2079 Parameters

2080 **C** (INOUT) An existing Matrix object that is being resized.

2081 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the  
2082 current number of rows.

2083 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than  
2084 the current number of columns.

## 2085 Return Values

2086 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2087 blocking mode, this indicates that the API checks for the input  
2088 arguments passed successfully. Either way, output matrix **C** is ready  
2089 to be used in the next method of the sequence.

2090 **GrB\_PANIC** Unknown internal error.

2091 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2092 GraphBLAS objects (input or output) is in an invalid state caused  
2093 by a previous execution error. Call **GrB\_error()** to access any error  
2094 messages generated by the implementation.

2095 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.



2096           GrB\_NULL\_POINTER The C pointer is NULL.

2097           GrB\_INVALID\_VALUE nrows or ncols is zero or outside the range of the type GrB\_Index.

## 2098   Description

2099   Changes the number of rows and columns of C to nrows and ncols, respectively. The domain  $\mathbf{D}(\mathbf{C})$   
2100   of matrix C remains the same. The contents  $\mathbf{L}(\mathbf{C})$  are modified as described below.

2101   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  
2102   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  
2103   greater than or equal to ncols are dropped.  
2104

### 2105   4.2.5.5   Matrix\_clear: Clear a matrix

2106   Removes all elements (tuples) from a matrix.

## 2107   C Syntax

2108           GrB\_Info GrB\_Matrix\_clear(GrB\_Matrix A);

## 2109   Parameters

2110           A (IN) An existing GraphBLAS matrix to clear.

## 2111   Return Values

2112           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2113                       blocking mode, this indicates that the API checks for the input ar-  
2114                       guments passed successfully. Either way, output matrix A is ready  
2115                       to be used in the next method of the sequence.

2116           GrB\_PANIC Unknown internal error.

2117           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2118                       GraphBLAS objects (input or output) is in an invalid state caused  
2119                       by a previous execution error. Call GrB\_error() to access any error  
2120                       messages generated by the implementation.

2121           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2122           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2123                       any matrix constructor.

2124 **Description**

2125 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,  
2126  $\mathbf{L}(\mathbf{A}) = \emptyset$ . The dimensions of the matrix do not change.

2127 **4.2.5.6 Matrix\_nrows: Number of rows in a matrix**

2128 Retrieve the number of rows in a matrix.

2129 **C Syntax**

```
2130         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2131                                   const GrB_Matrix A);
```

2132 **Parameters**

2133 nrows (OUT) On successful return, contains the number of rows in the matrix.

2134 A (IN) An existing GraphBLAS matrix being queried.

2135 **Return Values**

2136 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2137 cessfully and the value of `nrows` has been set.

2138 GrB\_PANIC Unknown internal error.

2139 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2140 GraphBLAS objects (input or output) is in an invalid state caused  
2141 by a previous execution error. Call `GrB_error()` to access any error  
2142 messages generated by the implementation.

2143 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2144 any matrix constructor.

2145 GrB\_NULL\_POINTER `nrows` pointer is NULL.

2146 **Description**

2147 Return `nrows(A)` in `nrows` (the number of rows).

2148 **4.2.5.7 Matrix\_ncols: Number of columns in a matrix**

2149 Retrieve the number of columns in a matrix.

## 2150 C Syntax

```
2151      GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2152                               const GrB_Matrix A);
```

## 2153 Parameters

2154 ncols (OUT) On successful return, contains the number of columns in the matrix.

2155 A (IN) An existing GraphBLAS matrix being queried.

## 2156 Return Values

2157 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2158 cessfully and the value of ncols has been set.

2159 GrB\_PANIC Unknown internal error.

2160 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2161 GraphBLAS objects (input or output) is in an invalid state caused  
2162 by a previous execution error. Call GrB\_error() to access any error  
2163 messages generated by the implementation.

2164 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2165 any matrix constructor.

2166 GrB\_NULL\_POINTER ncols pointer is NULL.

## 2167 Description

2168 Return **ncols(A)** in ncols (the number of columns).

## 2169 4.2.5.8 Matrix\_nvals: Number of stored elements in a matrix

2170 Retrieve the number of stored elements (tuples) in a matrix.

## 2171 C Syntax

```
2172      GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2173                               const GrB_Matrix A);
```

2174 **Parameters**

2175            **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in  
2176            the matrix.

2177            **A** (IN) An existing GraphBLAS matrix being queried.

2178 **Return Values**

2179            **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2180            cessfully and the value of **nvals** has been set.

2181            **GrB\_PANIC** Unknown internal error.

2182            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2183            GraphBLAS objects (input or output) is in an invalid state caused  
2184            by a previous execution error. Call **GrB\_error()** to access any error  
2185            messages generated by the implementation.

2186            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2187            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2188            any matrix constructor.

2189            **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

2190 **Description**

2191 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of  
2192 **L(A)** (see Section 3.5.3).

2193 **4.2.5.9 Matrix\_build: Store elements from tuples into a matrix**

2194 **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);
```

2195 **Parameters**

2196            **C** (INOUT) An existing Matrix object to store the result.

2197 **row\_indices** (IN) Pointer to an array of row indices.

2198 **col\_indices** (IN) Pointer to an array of column indices.

2199 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
2200 matrix, **C**.

2201 **n** (IN) The number of entries contained in each array (the same for **row\_indices**,  
2202 **col\_indices**, and **values**).

2203 **dup** (IN) An associative and commutative binary operator to apply when duplicate  
2204 values for the same location are present in the input arrays. All three domains of  
2205 **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
2206 then duplicate locations will result in an error.

## 2207 Return Values

2208 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2209 blocking mode, this indicates that the API checks for the input  
2210 arguments passed successfully. Either way, output matrix **C** is  
2211 ready to be used in the next method of the sequence.

2212 **GrB\_PANIC** Unknown internal error.

2213 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2214 opaque GraphBLAS objects (input or output) is in an invalid  
2215 state caused by a previous execution error. Call **GrB\_error()** to  
2216 access any error messages generated by the implementation.

2217 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2218 **GrB\_UNINITIALIZED\_OBJECT** Either **C** has not been initialized by a call to any matrix construc-  
2219 tor, or **dup** has not been initialized by a call to **GrB\_BinaryOp\_new**.

2220 **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices** or **values** pointer is **NULL**.

2221 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** or **col\_indices** is outside the allowed range  
2222 for **C**.

2223 **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
2224 not all the same, or the domains of **values** and **C** are incompatible  
2225 with each other or  $D_{dup}$ .

2226 **GrB\_OUTPUT\_NOT\_EMPTY** Output matrix **C** already contains valid tuples (elements). In  
2227 other words, **GrB\_Matrix\_nvals(C)** returns a positive value.

2228 **GrB\_INVALID\_VALUE** indices contains a duplicate location and **dup** is **GrB\_NULL**.

## 2229 Description

2230 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  
 2231 only differs from  $\mathbf{C}$  in its domain; otherwise,  $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$ .

2232 Each tuple  $\{\text{row\_indices}[k], \text{col\_indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the  
 2233 output in the form of

$$2234 \quad \tilde{\mathbf{C}}(\text{row\_indices}[k], \text{col\_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2235 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
 2236 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{C}}$  as follows:

$$2237 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row\_indices}[k]=i \wedge \text{col\_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2238 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{C}}$  is copied into  $\mathbf{C}$  via typecasting its  
 2239 values to  $\mathbf{D}(\mathbf{C})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

2240 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2241 It is an error to call this function on an output object with existing elements. In other words,  
 2242 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2243 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,  
 2244 `col_indices`, or `values`.

### 2245 4.2.5.10 Matrix\_setElement: Set a single element in matrix

2246 Set one element of a matrix to a given value.

## 2247 C Syntax

```
2248 // scalar value
2249 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2250                               <type>         val,
2251                               GrB_Index        row_index,
2252                               GrB_Index        col_index);
2253
2254 // GraphBLAS scalar
2255 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2256                               const GrB_Scalar s,
2257                               GrB_Index        row_index,
2258                               GrB_Index        col_index);
```

## 2259 Parameters

2260           **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.  
2261           **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of  
2262           **C**.  
2263           **row\_index** (IN) Row index of element to be assigned  
2264           **col\_index** (IN) Column index of element to be assigned

## 2265 Return Values

2266           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2267           blocking mode, this indicates that the compatibility tests on in-  
2268           dex/dimensions and domains for the input arguments passed suc-  
2269           cessfully. Either way, the output matrix **C** is ready to be used in  
2270           the next method of the sequence.  
2271           **GrB\_PANIC** Unknown internal error.  
2272           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2273           GraphBLAS objects (input or output) is in an invalid state caused  
2274           by a previous execution error. Call **GrB\_error()** to access any error  
2275           messages generated by the implementation.  
2276           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.  
2277           **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been  
2278           initialized by a call to a respective constructor.  
2279           **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e., not less  
2280           than **nrows(C)** or **ncols(C)**, respectively).  
2281           **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and the scalar are incompatible.

## 2282 Description

2283 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or  
2284 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from  
2285 one domain can be cast to values in the other domain as per the rules of the C language. In  
2286 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
2287 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
2288 **GrB\_Matrix\_setElement** ends and the domain mismatch error listed above is returned.

2289 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2290 \quad & 0 \leq \text{row\_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col\_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2291 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid  
 2292 index error listed above is returned.

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

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

2293 In the case of a transparent scalar or if  $\mathbf{L}(s)$  is not empty, then a value will be stored at the  
 2294 specified location in  $C$ , overwriting any value that may have been stored there before. In the case  
 2295 of a GraphBLAS scalar and if  $\mathbf{L}(s)$  is empty, then any value stored at the specified location in  $C$   
 2296 will be removed.

2297 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
 2298 of  $C$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
 2299 return value `GrB_SUCCESS` and the new content of vector  $C$  is as defined above but may not be  
 2300 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 2301 **4.2.5.11 Matrix\_removeElement: Remove an element from a matrix**

2302 Remove (annihilate) one stored element from a matrix.

#### 2303 **C Syntax**

```
2304      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2305                                         GrB_Index   row_index,
2306                                         GrB_Index   col_index);
```

#### 2307 **Parameters**

2308 `C` (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2309 `row_index` (IN) Row index of element to be removed

2310 `col_index` (IN) Column index of element to be removed

#### 2311 **Return Values**

2312 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 2313 blocking mode, this indicates that the compatibility tests on in-  
 2314 dex/dimensions and domains for the input arguments passed suc-  
 2315 cessfully. Either way, the output matrix  $C$  is ready to be used in  
 2316 the next method of the sequence.

2317 `GrB_PANIC` Unknown internal error.



2318       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 2319       GraphBLAS objects (input or output) is in an invalid state caused  
 2320       by a previous execution error. Call GrB\_error() to access any error  
 2321       messages generated by the implementation.

2322       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2323 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to  
 2324       any matrix constructor.

2325       GrB\_INVALID\_INDEX row\_index or col\_index is outside the allowable range (i.e., not less  
 2326       than nrows(C) or ncols(C), respectively).

## 2327 Description

2328 First, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned}
 &0 \leq \text{row\_index} < \text{nrows}(\mathbf{C}), \\
 &0 \leq \text{col\_index} < \text{ncols}(\mathbf{C})
 \end{aligned}$$

2330 If either of these conditions is violated, execution of GrB\_Matrix\_removeElement ends and the  
 2331 invalid index error listed above is returned.

2332 We are now ready to carry out the removal of a value that may be stored at the location specified by  
 2333 (row\_index, col\_index). If a value does not exist at the specified location in C, no error is reported  
 2334 and the operation has no effect on the state of C. In either case, the following will be true on return  
 2335 from this method: (row\_index, col\_index)  $\notin$  ind(C)

2336 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 2337 of C is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 2338 return value GrB\_SUCCESS and the new content of vector C is as defined above but may not be  
 2339 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2340 4.2.5.12 Matrix\_extractElement: Extract a single element from a matrix

2341 Extract one element of a matrix into a scalar.

## 2342 C Syntax

```

2343 // scalar value
2344 GrB_Info GrB_Matrix_extractElement(<type>          *val,
2345                                   const GrB_Matrix A,
2346                                   GrB_Index         row_index,
2347                                   GrB_Index         col_index);
2348
2349 // GraphBLAS scalar

```

```

2350         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2351                                           const GrB_Matrix A,
2352                                           GrB_Index      row_index,
2353                                           GrB_Index      col_index);
2354

```

## 2355 Parameters

2356     **val or s** (INOUT) An existing scalar whose domain is compatible with the domain of matrix  
2357     **A**. On successful return, this scalar holds the result of the extract. Any previous  
2358     value stored in **val** or **s** is overwritten.

2359     **A** (IN) The GraphBLAS matrix from which an element is extracted.

2360     **row\_index** (IN) The row index of location in **A** to extract.

2361     **col\_index** (IN) The column index of location in **A** to extract.

## 2362 Return Values

2363     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2364     cessfully. This indicates that the compatibility tests on dimensions  
2365     and domains for the input arguments passed successfully, and the  
2366     output scalar, **val** or **s**, has been computed and is ready to be used  
2367     in the next method of the sequence.

2368     **GrB\_NO\_VALUE** When using the transparent scalar, **val**, this is returned when there  
2369     is no stored value at specified location.

2370     **GrB\_PANIC** Unknown internal error.

2371     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2372     GraphBLAS objects (input or output) is in an invalid state caused  
2373     by a previous execution error. Call **GrB\_error()** to access any error  
2374     messages generated by the implementation.

2375     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2376     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by  
2377     a call to a corresponding constructor.

2378     **GrB\_NULL\_POINTER** **val** pointer is **NULL**.

2379     **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e. less than  
2380     zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-  
2381     tively).

2382     **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and scalar are incompatible.

## 2383 Description

2384 First, the scalar and input matrix are tested for domain compatibility as follows:  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\mathbf{s})$   
 2385 must be compatible with  $\mathbf{D}(\mathbf{A})$ . Two domains are compatible with each other if values from  
 2386 one domain can be cast to values in the other domain as per the rules of the C language. In  
 2387 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 2388 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 2389 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

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

$$\begin{aligned} 2391 \quad & 0 \leq \text{row\_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2392 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index  
 2393 error listed above is returned.

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

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

2394 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  $\text{val}$   
 2395 with casting as necessary. If  $(\text{row\_index}, \text{col\_index}) \notin \mathbf{ind}(\mathbf{A})$ , then one of the follow occurs  
 2396 depending on output scalar type:

- 2397 • The GraphBLAS scalar,  $\mathbf{s}$ , is cleared and `GrB_SUCCESS` is returned.
- 2398 • The non-opaque scalar,  $\text{val}$ , is unchanged, and `GrB_NO_VALUE` is returned.

2399 When using the non-opaque scalar variant ( $\text{val}$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 2400 mode, the new contents of  $\text{val}$  are as defined above if the method exits with return value `GrB_SUCCESS`  
 2401 or `GrB_NO_VALUE`.

2402 When using the GraphBLAS scalar variant ( $\mathbf{s}$ ) with a `GrB_SUCCESS` return value, the method  
 2403 exits and the new contents of  $\mathbf{s}$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
 2404 In `GrB_NONBLOCKING` mode, the new contents of  $\mathbf{s}$  is as defined above but may not be fully  
 2405 computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2406 4.2.5.13 Matrix\_extractTuples: Extract tuples from a matrix

2407 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

## 2408 C Syntax

```
2409         GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2410                                         GrB_Index      *col_indices,
```

```

2411                                     <type>          *values,
2412                                     GrB_Index        *n,
2413                                     const GrB_Matrix  A);

```

## 2414 Parameters

2415     **row\_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the  
2416     row indices.

2417     **col\_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the  
2418     column indices.

2419     **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
2420     the stored values whose type is compatible with  $\mathbf{D}(\mathbf{A})$ .

2421     **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,  
2422     **row\_indices**, and **col\_indices** arrays can hold. Upon return, it will contain the  
2423     number of values written to the arrays.

2424     **A** (IN) An existing GraphBLAS matrix.

## 2425 Return Values

2426     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2427     cessfully. This indicates that the compatibility tests on the input  
2428     argument passed successfully, and the output arrays, **indices** and  
2429     **values**, have been computed.

2430     **GrB\_PANIC** Unknown internal error.

2431     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2432     GraphBLAS objects (input or output) is in an invalid state caused  
2433     by a previous execution error. Call **GrB\_error()** to access any error  
2434     messages generated by the implementation.

2435     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2436     **GrB\_INSUFFICIENT\_SPACE** Not enough space in **row\_indices**, **col\_indices**, and **values** (as indi-  
2437     cated by the **n** parameter) to hold all of the tuples that will be  
2438     extracted.

2439     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2440     any matrix constructor.

2441     **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices**, **values** or **n** pointer is NULL.

2442     **GrB\_DOMAIN\_MISMATCH** The domains of the **A** matrix and **values** array are incompatible  
2443     with one another.

## 2444 Description

2445 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with  
2446 those tuples are placed in the **values** array, the column indices are placed in the **col\_indices** array,  
2447 and the row indices are placed in the **row\_indices** array. These output arrays are pre-allocated by  
2448 the user before calling this function such that each output array has enough space to hold at least  
2449 **GrB\_Matrix\_nvals(A)** elements.

2450 Upon return of this function, a pair of  $\{\text{row\_indices}[k], \text{col\_indices}[k]\}$  are unique for every valid  
2451  $k$ , but they are not required to be sorted in any particular order. Each tuple  $(i, j, A_{ij})$  in **A** is  
2452 unzipped and copied into a distinct  $k$ th location in output vectors:

$$\{\text{row\_indices}[k], \text{col\_indices}[k], \text{values}[k]\} \leftarrow (i, j, A_{ij}),$$

2453 where  $0 \leq k < \text{GrB\_Matrix\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if **row\_indices**[ $k$ ],  
2454 **col\_indices**[ $k$ ] and **values**[ $k$ ] exist upon return, so does **row\_indices**[ $j$ ], **col\_indices**[ $j$ ] and **values**[ $j$ ] for  
2455 all  $j$  such that  $0 \leq j < k$ .

2456 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,  
2457 then a **GrB\_INSUFFICIENT\_SPACE** error is returned since it is undefined which subset of values  
2458 would be extracted.

2459 In both **GrB\_BLOCKING** mode **GrB\_NONBLOCKING** mode if the method exits with return value  
2460 **GrB\_SUCCESS**, the new contents of the arrays **row\_indices**, **col\_indices** and **values** are as defined  
2461 above.

### 2462 4.2.5.14 Matrix\_exportHint: Provide a hint as to which storage format might be most 2463 efficient for exporting a matrix

## 2464 C Syntax

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

## 2465 Parameters

2466 hint (OUT) Pointer to a value of type **GrB\_Format**.

2467 A (IN) A GraphBLAS matrix object.

## 2468 Return Values

2469 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2470 cessfully and the value of **hint** has been set.

2471 **GrB\_PANIC** Unknown internal error.

2472           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2473           opaque GraphBLAS objects (input or output) is in an invalid  
2474           state caused by a previous execution error. Call GrB\_error() to  
2475           access any error messages generated by the implementation.

2476           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2477           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2478           any matrix constructor.

2479           GrB\_NULL\_POINTER hint is NULL.

2480           GrB\_NO\_VALUE If the implementation does not have a preferred format, it may  
2481           return the value GrB\_NO\_VALUE.

## 2482 Description

2483 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for  
2484 exporting the matrix A. GraphBLAS implementations might return the current storage format of  
2485 the matrix, or the format to which it could most efficiently be exported. However, implementations  
2486 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is  
2487 free to refuse to provide a format hint, returning GrB\_NO\_VALUE.

### 2488 4.2.5.15 Matrix\_exportSize: Return the array sizes necessary to export a GraphBLAS 2489 matrix object

## 2490 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);

```

## 2491 Parameters

2492           n\_indptr (OUT) Pointer to a value of type GrB\_Index.

2493           n\_indices (OUT) Pointer to a value of type GrB\_Index.

2494           n\_values (OUT) Pointer to a value of type GrB\_Index.

2495           format (IN) a value indicating the format in which the matrix will be exported, as defined  
2496           in Section 3.5.3.1.

2497           A (IN) A GraphBLAS matrix object.

## 2498 Return Values

2499                   GrB\_SUCCESS In blocking mode or non-blocking mode, the operation com-  
2500                   pleted successfully. This indicates that the API checks for the  
2501                   input arguments passed successfully, and the number of elements  
2502                   necessary for the export buffers have been written to `n_indptr`,  
2503                   `n_indices`, and `n_values`, respectively.

2504                   GrB\_PANIC Unknown internal error.

2505                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2506                   opaque GraphBLAS objects (input or output) is in an invalid  
2507                   state caused by a previous execution error. Call `GrB_error()` to  
2508                   access any error messages generated by the implementation.

2509                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2510                   GrB\_UNINITIALIZED\_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to  
2511                   any matrix constructor.

2512                   GrB\_NULL\_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

## 2513 Description

2514 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to  
2515 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and  
2516 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be  
2517 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user  
2518 as one of the values defined in Section 3.5.3.1.

### 2519 4.2.5.16 Matrix\_export: Export a GraphBLAS matrix to a pre-defined format

## 2520 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);
```

## 2521 Parameters

2522        **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-  
2523        dices, depending on the value of **format**. It must be large enough to hold at  
2524        least **n\_indptr** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2525        **GrB\_Matrix\_exportSize()** method.

2526        **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements  
2527        in **values**, depending on the value of **format**. It must be large enough to hold at  
2528        least **n\_indices** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2529        **GrB\_Matrix\_exportSize()** method.

2530        **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-  
2531        ment must match the type of the values stored in **A**. It must be large enough  
2532        to hold at least **n\_values** elements of that type, where **n\_values** was returned from  
2533        **GrB\_Matrix\_exportSize**.

2534        **n\_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**  
2535        array can hold. Upon return, it will contain the number of elements written to the  
2536        array.

2537        **n\_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**  
2538        array can hold. Upon return, it will contain the number of elements written to the  
2539        array.

2540        **n\_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**  
2541        array can hold. Upon return, it will contain the number of elements written to the  
2542        array.

2543        **format** (IN) a value indicating the format in which the matrix will be exported, as defined  
2544        in Section 3.5.3.1.

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

## 2546 Return Values

2547        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2548        cessfully. This indicates that the compatibility tests on the input  
2549        argument passed successfully, and the output arrays, **indptr**, **in-**  
2550        **dices** and **values**, have been computed.

2551        **GrB\_PANIC** Unknown internal error.

2552        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2553        opaque GraphBLAS objects (input or output) is in an invalid  
2554        state caused by a previous execution error. Call **GrB\_error()** to  
2555        access any error messages generated by the implementation.

2556        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.



2557       GrB\_INSUFFICIENT\_SPACE Not enough space in `indptr`, `indices`, and/or `values` (as indicated  
2558                                   by the corresponding `n_*` parameter) to hold all of the corre-  
2559                                   sponding elements that will be extacted.

2560       GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2561                                   any matrix constructor.

2562       GrB\_NULL\_POINTER `indptr`, `indices`, `values` `n_indptr`, `n_indices`, `n_values` pointer is  
2563                                   NULL.

2564       GrB\_DOMAIN\_MISMATCH The domain of `A` does not match with the type of `values`.

## 2565 Description

2566 Given a matrix `A`, this method exports the contents of the matrix into one of the pre-defined  
2567 `GrB_Format` formats from Section 3.5.3.1. The user-allocated arrays pointed to by `indptr`, `indices`,  
2568 and `values` must be at least large enough to hold the corresponding number of elements returned  
2569 by calling `GrB_Matrix_exportSize`. The value of `format` can be chosen arbitrarily, but a call to  
2570 `GrB_Matrix_exportHint` may suggest a format that results in the most efficient export. Details  
2571 of the contents of `indptr`, `indices`, and `values` corresponding to each supported format is given in  
2572 Appendix B.

### 2573 4.2.5.17 Matrix\_import: Import a matrix into a GraphBLAS object

## 2574 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);

```

## 2575 Parameters

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

2578       `d` (IN) The type corresponding to the domain of the matrix being created. Can be  
2579           one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
2580           GraphBLAS type.

2581        **nrows** (IN) Integer value holding the number of rows in the matrix.

2582        **ncols** (IN) Integer value holding the number of columns in the matrix.

2583        **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the  
2584        value of **format**.

2585        **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending  
2586        on the value of **format**.

2587        **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2588        **n\_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2589        **n\_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2590        **n\_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2591        **format** (IN) a value indicating the format of the matrix being imported, as defined in  
2592        Section 3.5.3.1.

## 2593 **Return Values**

2594        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2595        blocking mode, this indicates that the API checks for the input  
2596        arguments passed successfully and the input arrays have been  
2597        consumed. Either way, output matrix **A** is ready to be used in  
2598        the next method of the sequence.

2599        **GrB\_PANIC** Unknown internal error.

2600        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2601        **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2602        (needed for user-defined types).

2603        **GrB\_NULL\_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2604        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices  
2605        in **A** and or the size of **values**, **n\_values**, depending on the value  
2606        of **format**.

2607        **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

2608        **GrB\_DOMAIN\_MISMATCH** The domain given in parameter **d** does not match the element  
2609        type of **values**.

## 2610 Description

2611 Creates a new matrix **A** of domain **D**(d) and dimension **nrows**  $\times$  **ncols**. The new GraphBLAS  
2612 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.  
2613 The method returns a handle to the new matrix in **A**. The structure of the data being imported is  
2614 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of  
2615 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2616 It is not an error to call this method more than once on the same output matrix; however, the  
2617 handle to the previously created object will be overwritten.

## 2618 4.2.5.18 Matrix\_serializeSize: Compute the serialize buffer size

2619 Compute the buffer size (in bytes) necessary to serialize a GrB\_Matrix using GrB\_Matrix\_serialize.

## 2620 C Syntax

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

## 2621 Parameters

2622 size (OUT) Pointer to GrB\_Index value where size in bytes of serialized object will be  
2623 written.

2624 A (IN) A GraphBLAS matrix object.

## 2625 Return Values

2626 GrB\_SUCCESS The operation completed successfully and the value pointed to  
2627 by \*size has been computed and is ready to use.

2628 GrB\_PANIC Unknown internal error.

2629 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2630 GrB\_NULL\_POINTER size is NULL.

## 2631 Description

2632 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object A.  
2633 Users may then allocate a buffer of size bytes to pass as a parameter to GrB\_Matrix\_serialize.

2634 **4.2.5.19 Matrix\_serialize: Serialize a GraphBLAS matrix.**

2635 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2636 **C Syntax**

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

2637 **Parameters**

2638 **serialized\_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be  
2639 written.

2640 **serialized\_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized\_data**.  
2641 On output, the number of bytes written to **serialized\_data**.

2642 **A** (IN) A GraphBLAS matrix object.

2643 **Return Values**

2644 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2645 cessfully. This indicates that the compatibility tests on the in-  
2646 put argument passed successfully, and the output buffer **serial-  
2647 ized\_data** and **serialized\_size**, have been computed and are ready  
2648 to use.

2649 **GrB\_PANIC** Unknown internal error.

2650 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2651 opaque GraphBLAS objects (input or output) is in an invalid  
2652 state caused by a previous execution error. Call **GrB\_error()** to  
2653 access any error messages generated by the implementation.

2654 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2655 **GrB\_NULL\_POINTER** **serialized\_data** or **serialize\_size** is NULL.

2656 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2657 any matrix constructor.

2658 **GrB\_INSUFFICIENT\_SPACE** The size of the buffer **serialized\_data** (provided as an input **seri-  
2659 alized\_size**) was not large enough.

## 2660 Description

2661 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,  
2662 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must  
2663 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the  
2664 serialized matrix written to `serialized_data` is provided upon completion as an output written to  
2665 `serialized_size`.

2666 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created  
2667 with one library implementation is not necessarily valid for deserialization with another implemen-  
2668 tation.

### 2669 4.2.5.20 Matrix\_deserialize: Deserialize a GraphBLAS matrix.

2670 Construct a new GraphBLAS matrix from a serialized object.

## 2671 C Syntax

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

## 2672 Parameters

2673 A (INOUT) On a successful return, contains a handle to the newly created Graph-  
2674 BLAS matrix.

2675 d (IN) the type of the matrix that was serialized in `serialized_data`.

2676 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2677 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

## 2678 Return Values

2679 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2680 blocking mode, this indicates that the API checks for the input  
2681 arguments passed successfully. Either way, output matrix A is  
2682 ready to be used in the next method of the sequence.

2683 GrB\_PANIC Unknown internal error.

2684 GrB\_INVALID\_OBJECT This is returned if `serialized_data` is invalid or corrupted.

2685 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2686 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
2687 (needed for user-defined types).

2688 GrB\_NULL\_POINTER serialized\_data or A is NULL.

2689 GrB\_DOMAIN\_MISMATCH The type given in d does not match the type of the matrix  
2690 serialized in serialized\_data.

## 2691 Description

2692 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object  
2693 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The  
2694 domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized  
2695 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2696 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix  
2697 serialized in one library implementation can be deserialized by another.

2698 It is not an error to call this method more than once on the same output matrix; however, the  
2699 handle to the previously created object will be overwritten.

## 2700 4.2.6 Descriptor methods

2701 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-  
2702 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

### 2703 4.2.6.1 Descriptor\_new: Create new descriptor

2704 Creates a new (empty or default) descriptor.

## 2705 C Syntax

2706 GrB\_Info GrB\_Descriptor\_new(GrB\_Descriptor \*desc);

## 2707 Parameters

2708 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2709 descriptor.

## 2710 Return Value

2711 GrB\_SUCCESS The method completed successfully.

2712 GrB\_PANIC unknown internal error.

2713        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2714        GrB\_NULL\_POINTER desc pointer is NULL.

## 2715    **Description**

2716    Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can  
2717    be populated by calls to Descriptor\_set.

2718    It is not an error to call this method more than once on the same variable; however, the handle to  
2719    the previously created object will be overwritten.

### 2720    **4.2.6.2    Descriptor\_set: Set content of descriptor**

2721    Sets the content for a field for an existing descriptor.

## 2722    **C Syntax**

```
2723        GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2724                                    GrB_Desc_Field        field,  
2725                                    GrB_Desc_Value        val);
```

## 2726    **Parameters**

2727        desc (IN) An existing GraphBLAS descriptor to be modified.

2728        field (IN) The field being set.

2729        val (IN) New value for the field being set.

## 2730    **Return Values**

2731        GrB\_SUCCESS operation completed successfully.

2732        GrB\_PANIC unknown internal error.

2733        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2734    GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

2735        GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 2736 Description

2737 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor  
2738 to set the value associated with that field. Valid values for the `field` parameter include the following:

2739 `GrB_OUTP` refers to the output parameter (result) of the operation.

2740 `GrB_MASK` refers to the mask parameter of the operation.

2741 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2742 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2743 Valid values for the `val` parameter are:

2744 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask  
2745 (`GrB_MASK`) parameter.

2746 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-  
2747 eter. When combined with `GrB_STRUCTURE`, the complement of the  
2748 structure of the mask is used without evaluating the values stored.

2749 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input  
2750 matrix parameters only).

2751 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear  
2752 the matrix first (or clear the non-masked entries). The default behavior  
2753 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`  
2754 parameter only.

2755 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,  
2756 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).  
2757 A value for a given field may be set multiple times but will have no additional effect. Fields that  
2758 have no values set result in their default behavior, as defined in Section 3.7.

## 2759 4.2.7 free: Destroy an object and release its resources

2760 Destroys a previously created GraphBLAS object and releases any resources associated with the  
2761 object.

## 2762 C Syntax

2763 `GrB_Info GrB_free(<GrB_Object> *obj);`



## 2764 Parameters

2765       obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have  
2766       been created by an explicit call to a GraphBLAS constructor. It can be any of the  
2767       opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,  
2768       binary op, unary op, or type. On successful completion of GrB\_free, obj behaves  
2769       as an uninitialized object.

## 2770 Return Values

2771       GrB\_SUCCESS operation completed successfully

2772       GrB\_PANIC unknown internal error. If this return value is encountered when  
2773       in nonblocking mode, the error responsible for the panic condition  
2774       could be from any method involved in the computation of the input  
2775       object. The GrB\_error() method should be called for additional  
2776       information.

## 2777 Description

2778 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime  
2779 system. A call to GrB\_free frees those resources so they are available for use by other GraphBLAS  
2780 objects.

2781 The parameter passed into GrB\_free is a handle referencing a GraphBLAS opaque object of a data  
2782 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-  
2783 structor. The behavior of a program that calls GrB\_free on a pre-defined object is implementation  
2784 defined.

2785 After the GrB\_free method returns, the object referenced by the input handle is destroyed and the  
2786 handle has the value GrB\_INVALID\_HANDLE. The handle can be used in subsequent GraphBLAS  
2787 methods but only after the handle has been reinitialized with a call the the appropriate \_new or  
2788 \_dup method.

2789 Note that unlike other GraphBLAS methods, calling GrB\_free with an object with an invalid handle  
2790 is legal. The system may attempt to free resources that might be associated with that object, if  
2791 possible, and return normally.

2792 When using GrB\_free it is possible to create a dangling reference to an object. This would occur  
2793 when a handle is assigned to a second variable of the same opaque type. This creates two handles  
2794 that reference the same object. If GrB\_free is called with one of the variables, the object is destroyed  
2795 and the handle associated with the other variable no longer references a valid object. This is not an  
2796 error condition that the implementation of the GraphBLAS API can be expected to catch, hence  
2797 programmers must take care to prevent this situation from occurring.

2798 **4.2.8 wait: Return once an object is either *complete* or *materialized***

2799 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2800 **C Syntax**

2801 `GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);`

2802 **Parameters**

2803 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an  
2804 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2805 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2806 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another  
2807 thread (completion) or all computing to produce `obj` by all GraphBLAS operations  
2808 in its sequence have finished (materialization).

2809 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the  
2810 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or  
2811 `GrB_MATERIALIZE`.

2812 **Return values**

2813 `GrB_SUCCESS` operation completed successfully.

2814 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-  
2815 pletion of pending operations.

2816 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion  
2817 of pending operations.

2818 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2819 or other constructor, method.

2820 `GrB_PANIC` unknown internal error.

2821 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`  
2822 `GrB_MATERIALIZE`.

2823 **Description**

2824 On successful return from `GrB_wait()`, the input object, `obj` is in one of two states depending on  
2825 the mode of `GrB_wait`:

- 2826 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program  
2827 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another  
2828 thread. This result occurs when the mode parameter is set to `GrB_COMPLETE`.
- 2829 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on  
2830 behalf of `obj` and error information is available. This result occurs when the mode parameter  
2831 is set to `GrB_MATERIALIZE`.

2832 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,  
2833 `GrB_wait(obj,mode)` has no effect when called in blocking mode.

2834 In non-blocking mode, the status of any pending method calls, other than those associated with pro-  
2835 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to `GrB_wait(obj,mode)`.  
2836 Methods in the sequence for `obj`, however, most likely would be impacted by a call to `GrB_wait(obj,mode)`;  
2837 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be  
2838 finished prior to the return from `GrB_wait(obj,mode)`.

#### 2839 4.2.9 error: Retrieve an error string

2840 Retrieve an error-message about any errors encountered during the processing associated with an  
2841 object.

### 2842 C Syntax

```
2843         GrB_Info GrB_error(const char          **error,
2844                           const GrB_Object      obj);
```

#### 2845 Parameters

2846 `error` (OUT) A pointer to a null-terminated string. The contents of the string are im-  
2847 plementation defined.

2848 `obj` (IN) An existing GraphBLAS object. The object must have been created by an  
2849 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2850 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2851 or type.

#### 2852 Return value

2853 `GrB_SUCCESS` operation completed successfully.

2854 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2855 or other constructor, method.

2856 `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	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

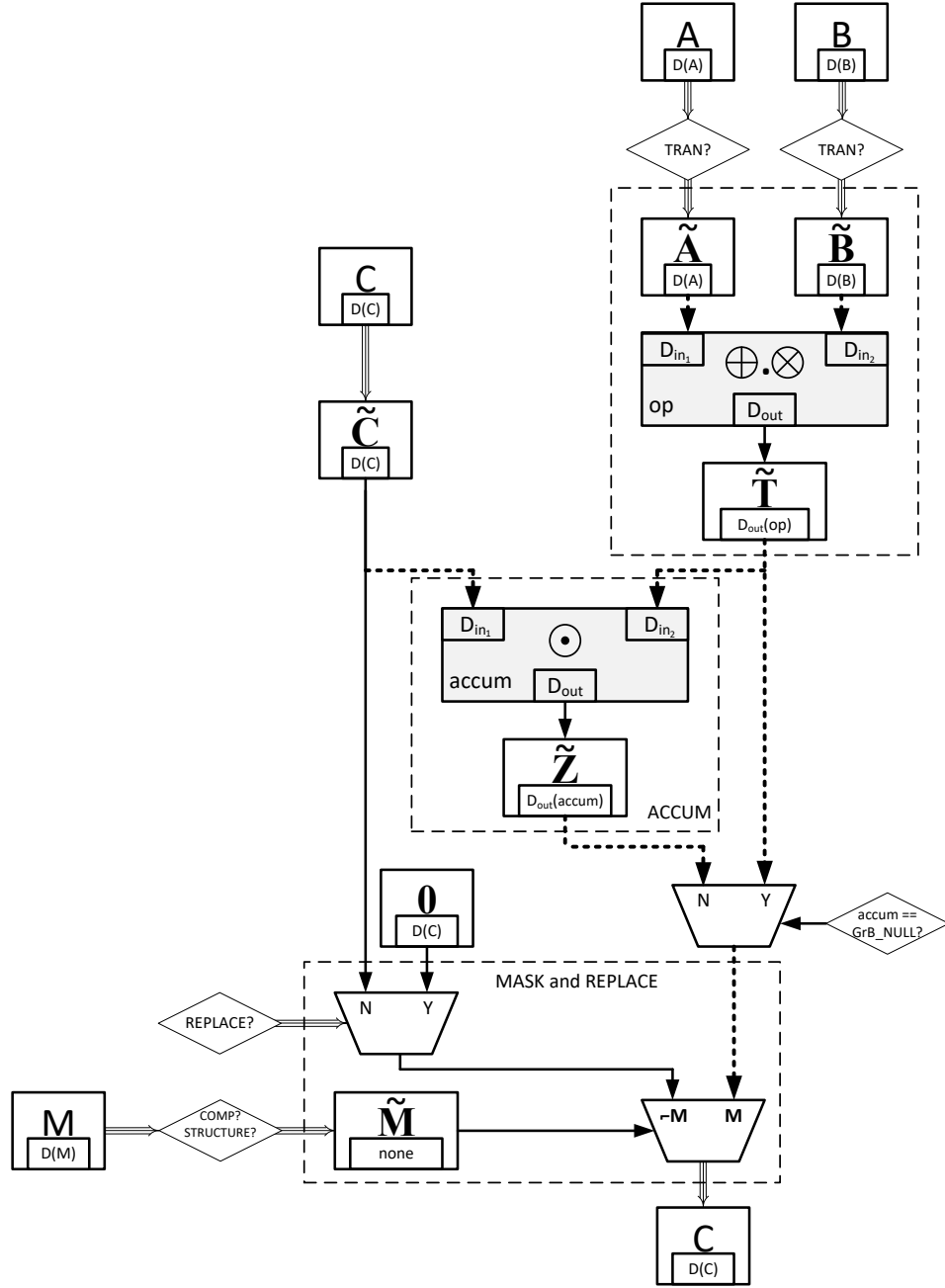


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows ( $\Rightarrow$ ) denote where “as if copy” takes place (including both collections and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

2926 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-  
 2927 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at  
 2928 the point of their declaration, as shown in the following examples:

```
2929         GrB_Type          type = GrB_INVALID_HANDLE;
2930         GrB_Semiring      semiring = GrB_INVALID_HANDLE;
2931         GrB_Matrix        matrix = GrB_INVALID_HANDLE;
```

## 2932 Compliance

2933 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.  
 2934 That is, for each operation we give a recipe for producing its outcome. Any implementation that  
 2935 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error  
 2936 model (Section 2.6) is a conforming implementation.

### 2937 4.3.1 mxm: Matrix-matrix multiply

2938 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

## 2939 C Syntax

```
2940         GrB_Info GrB_mxm(GrB_Matrix          C,
2941                         const GrB_Matrix      Mask,
2942                         const GrB_BinaryOp     accum,
2943                         const GrB_Semiring     op,
2944                         const GrB_Matrix       A,
2945                         const GrB_Matrix       B,
2946                         const GrB_Descriptor   desc);
```

## 2947 Parameters

2948 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 2949 that may be accumulated with the result of the matrix product. On output, the  
 2950 matrix holds the results of the operation.

2951 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 2952 stored into the output matrix C. The mask dimensions must match those of the  
 2953 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
 2954 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
 2955 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 2956 dimensions of C), `GrB_NULL` should be specified.



2957 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
 2958 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 2959 specified.

2960 **op** (IN) The semiring used in the matrix-matrix multiply.

2961 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 2962 multiplication.

2963 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 2964 multiplication.

2965 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 2966 should be specified. Non-default field/value pairs are listed as follows:  
 2967

Param	Field	Value	Description
<b>C</b>	<b>GrB_OUTP</b>	<b>GrB_REPLACE</b>	Output matrix <b>C</b> is cleared (all elements removed) before the result is stored in it.
<b>Mask</b>	<b>GrB_MASK</b>	<b>GrB_STRUCTURE</b>	The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined.
<b>Mask</b>	<b>GrB_MASK</b>	<b>GrB_COMP</b>	Use the complement of <b>Mask</b> .
<b>A</b>	<b>GrB_INP0</b>	<b>GrB_TRAN</b>	Use transpose of <b>A</b> for the operation.
<b>B</b>	<b>GrB_INP1</b>	<b>GrB_TRAN</b>	Use transpose of <b>B</b> for the operation.

## 2969 Return Values

2970 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 2971 blocking mode, this indicates that the compatibility tests on di-  
 2972 mensions and domains for the input arguments passed successfully.  
 2973 Either way, output matrix **C** is ready to be used in the next method  
 2974 of the sequence.

2975 **GrB\_PANIC** Unknown internal error.

2976 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 2977 GraphBLAS objects (input or output) is in an invalid state caused  
 2978 by a previous execution error. Call **GrB\_error()** to access any error  
 2979 messages generated by the implementation.

2980 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

2981 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 2982 a call to **new** (or **Matrix\_dup** for matrix parameters).

2983 **GrB\_DIMENSION\_MISMATCH** Mask and/or matrix dimensions are incompatible.

2984 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 2985 corresponding domains of the semiring or accumulation operator,  
 2986 or the mask's domain is not compatible with `bool` (in the case where  
 2987 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 2988 Description

2989 GrB\_mxm computes the matrix product  $C = A \oplus . \otimes B$  or, if an optional binary accumulation operator  
 2990  $(\odot)$  is provided,  $C = C \odot (A \oplus . \otimes B)$  (where matrices  $A$  and  $B$  can be optionally transposed).  
 2991 Logically, this operation occurs in three steps:

2992 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 2993 and dimensions are tested for compatibility.

2994 **Compute** The indicated computations are carried out.

2995 **Output** The result is written into the output matrix, possibly under control of a mask.

2996 Up to four argument matrices are used in the GrB\_mxm operation:

- 2997 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 2998 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)
- 2999 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3000 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3001 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for  
 3002 domain compatibility as follows:

- 3003 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3004 must be from one of the pre-defined types of Table 3.2.
- 3005 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3006 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3007 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3008 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3009 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 3010 of the accumulation operator.

3011 Two domains are compatible with each other if values from one domain can be cast to values in  
 3012 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3013 all compatible with each other. A domain from a user-defined type is only compatible with itself.

3014 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch  
3015 error listed above is returned.

3016 From the argument matrices, the internal matrices and mask used in the computation are formed  
3017 ( $\leftarrow$  denotes copy):

- 3018 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3019 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3020 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
3021  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3022 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3023 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
3024  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3025 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
3026  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 3027 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3028 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3029 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3030 The internal matrices and masks are checked for dimension compatibility. The following conditions  
3031 must hold:

- 3032 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 3033 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 3034 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 3035 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .
- 3036 5.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .

3037 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch  
3038 error listed above is returned.

3039 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3040 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3041 We are now ready to carry out the matrix multiplication and any additional associated operations.  
3042 We describe this in terms of two intermediate matrices:

- 3043 •  $\tilde{\mathbf{T}}$ : The matrix holding the product of matrices  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3044 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3045 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

$$3047 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3048 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring  $\mathbf{op}$ , respectively.

3049 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3050 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3051 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3052 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3053 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
3054 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 3055 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3056 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3057 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3058 \quad & \\ 3059 \end{aligned}$$

3060 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3061 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
3062 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
3063 mask which acts as a “write mask”.

- 3064 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
3065 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3066 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3067 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
3068 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
3069 mask are unchanged:

$$3070 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3071 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
3072 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
3073 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
3074 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3075 sequence.

### 3076 4.3.2 vxm: Vector-matrix multiply

3077 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

### 3078 C Syntax

```
3079         GrB_Info GrB_vxm(GrB_Vector          w,  
3080                           const GrB_Vector    mask,  
3081                           const GrB_BinaryOp   accum,  
3082                           const GrB_Semiring   op,  
3083                           const GrB_Vector    u,  
3084                           const GrB_Matrix    A,  
3085                           const GrB_Descriptor desc);
```

### 3086 Parameters

3087 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3088 that may be accumulated with the result of the vector-matrix product. On output,  
3089 this vector holds the results of the operation.

3090 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3091 stored into the output vector **w**. The mask dimensions must match those of the  
3092 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3093 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3094 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3095 dimensions of **w**), **GrB\_NULL** should be specified.

3096 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3097 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3098 specified.

3099 **op** (IN) Semiring used in the vector-matrix multiply.

3100 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3101 multiplication.

3102 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
3103 multiplication.

3104 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
3105 should be specified. Non-default field/value pairs are listed as follows:  
3106

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation.

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

**GrB\_PANIC** Unknown internal error.

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

**GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

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

**GrB\_DIMENSION\_MISMATCH** Mask, vector, and/or matrix dimensions are incompatible.

**GrB\_DOMAIN\_MISMATCH** The domains of the various vectors/matrices are incompatible with the corresponding domains of the semiring or accumulation operator, or the mask's domain is not compatible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## Description

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

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

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

3134     **Output** The result is written into the output vector, possibly under control of a mask.

3135 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3136     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3137     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3138     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3139     4.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3140 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
 3141 tested for domain compatibility as follows:

- 3142     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3143         must be from one of the pre-defined types of Table 3.2.
- 3144     2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the semiring.
- 3145     3.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the semiring.
- 3146     4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring.
- 3147     5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3148         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$   
 3149         of the accumulation operator.

3150 Two domains are compatible with each other if values from one domain can be cast to values in  
 3151 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3152 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
 3153 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch  
 3154 error listed above is returned.

3155 From the argument vectors and matrices, the internal matrices and mask used in the computation  
 3156 are formed ( $\leftarrow$  denotes copy):

- 3157     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3158     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3159         (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3160         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3161             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 3162             ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 3163         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3164     3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3165 4. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

3166 The internal matrices and masks are checked for shape compatibility. The following conditions  
3167 must hold:

3168 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$ .

3169 2.  $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$ .

3170 3.  $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$ .

3171 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch  
3172 error listed above is returned.

3173 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3174 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3175 We are now ready to carry out the vector-matrix multiplication and any additional associated  
3176 operations. We describe this in terms of two intermediate vectors:

- 3177 •  $\tilde{\mathbf{t}}$ : The vector holding the product of vector  $\tilde{\mathbf{u}}^T$  and matrix  $\tilde{\mathbf{A}}$ .
- 3178 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3179 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.  
3180 The value of each of its elements is computed by

$$3181 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3182 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring `op`, respectively.

3183 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3184 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3185 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3186 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3187 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3188 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3189 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3190 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3191 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3192 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3193 \end{aligned}$$

3194 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.



3195 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3196 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3197 mask which acts as a “write mask”.

- 3198 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3199 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3200 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3201 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3202 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3203 mask are unchanged:

$$3204 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3205 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3206 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3207 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3208 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3209 sequence.

### 3210 4.3.3 mxv: Matrix-vector multiply

3211 Multiplies a matrix by a vector on a semiring. The result is a vector.

## 3212 C Syntax

```
3213      GrB_Info GrB_mxv(GrB_Vector      w,
3214                      const GrB_Vector mask,
3215                      const GrB_BinaryOp accum,
3216                      const GrB_Semiring op,
3217                      const GrB_Matrix A,
3218                      const GrB_Vector u,
3219                      const GrB_Descriptor desc);
```

## 3220 Parameters

3221 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
 3222 that may be accumulated with the result of the matrix-vector product. On output,  
 3223 this vector holds the results of the operation.

3224 **mask** (IN) An optional “write” mask that controls which results from this operation are  
 3225 stored into the output vector  $\mathbf{w}$ . The mask dimensions must match those of the  
 3226 vector  $\mathbf{w}$ . If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain

3227 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types  
 3228 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 3229 dimensions of `w`), `GrB_NULL` should be specified.

3230 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`  
 3231 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
 3232 specified.

3233 `op` (IN) Semiring used in the vector-matrix multiply.

3234 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3235 multiplication.

3236 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3237 multiplication.

3238 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
 3239 should be specified. Non-default field/value pairs are listed as follows:

3240

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.

3241

## 3242 Return Values

3243 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 3244 blocking mode, this indicates that the compatibility tests on di-  
 3245 mensions and domains for the input arguments passed successfully.  
 3246 Either way, output vector `w` is ready to be used in the next method  
 3247 of the sequence.

3248 `GrB_PANIC` Unknown internal error.

3249 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
 3250 GraphBLAS objects (input or output) is in an invalid state caused  
 3251 by a previous execution error. Call `GrB_error()` to access any error  
 3252 messages generated by the implementation.

3253 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3254 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by  
 3255 a call to `new` (or `dup` for matrix or vector parameters).

3256 GrB\_DIMENSION\_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3257 GrB\_DOMAIN\_MISMATCH The domains of the various vectors/matrices are incompatible with  
3258 the corresponding domains of the semiring or accumulation opera-  
3259 tor, or the mask's domain is not compatible with **bool** (in the case  
3260 where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 3261 Description

3262 GrB\_mvx computes the matrix-vector product  $w = A \oplus . \otimes u$ , or, if an optional binary accumulation  
3263 operator ( $\odot$ ) is provided,  $w = w \odot (A \oplus . \otimes u)$  (where matrix  $A$  can be optionally transposed).  
3264 Logically, this operation occurs in three steps:

3265 **Setup** The internal vectors, matrices and mask used in the computation are formed and their  
3266 domains/dimensions are tested for compatibility.

3267 **Compute** The indicated computations are carried out.

3268 **Output** The result is written into the output vector, possibly under control of a mask.

3269 Up to four argument vectors or matrices are used in the GrB\_mvx operation:

- 3270 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3271 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3272 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3273 4.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3274 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
3275 tested for domain compatibility as follows:

- 3276 1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
3277 must be from one of the pre-defined types of Table 3.2.
- 3278 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3279 3.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3280 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3281 5. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3282 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
3283 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

$$t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3316 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring **op**, respectively.  
 3317 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3318 • If **accum** = **GrB\_NULL**, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3319 • If **accum** is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3320 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3321 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3322 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3323 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3324 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3325 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3326 \end{aligned}$$

3327 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.  
 3328

3329 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector **w**,  
 3330 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3331 mask which acts as a “write mask”.

- 3332 • If **desc[GrB\_OUTP].GrB\_REPLACE** is set, then any values in **w** on input to this operation are  
 3333 deleted and the content of the new output vector, **w**, is defined as,

$$3334 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3335 • If **desc[GrB\_OUTP].GrB\_REPLACE** is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3336 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the  
 3337 mask are unchanged:

$$3338 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3339 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 3340 of vector **w** is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 3341 exits with return value **GrB\_SUCCESS** and the new content of vector **w** is as defined above but  
 3342 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3343 sequence.

#### 3344 4.3.4 eWiseMult: Element-wise multiplication

3345 **Note:** The difference between **eWiseAdd** and **eWiseMult** is not about the element-wise operation  
 3346 but how the index sets are treated. **eWiseAdd** returns an object whose indices are the “union” of  
 3347 the indices of the inputs whereas **eWiseMult** returns an object whose indices are the “intersection”  
 3348 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3349 the set of values from the resulting index set.

#### 3350 4.3.4.1 eWiseMult: Vector variant

3351 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-  
3352 ducing a third vector as result.

#### 3353 C Syntax

```
3354     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3355                           const GrB_Vector mask,  
3356                           const GrB_BinaryOp accum,  
3357                           const GrB_Semiring op,  
3358                           const GrB_Vector u,  
3359                           const GrB_Vector v,  
3360                           const GrB_Descriptor desc);  
3361  
3362     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3363                           const GrB_Vector mask,  
3364                           const GrB_BinaryOp accum,  
3365                           const GrB_Monoid op,  
3366                           const GrB_Vector u,  
3367                           const GrB_Vector v,  
3368                           const GrB_Descriptor desc);  
3369  
3370     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3371                           const GrB_Vector mask,  
3372                           const GrB_BinaryOp accum,  
3373                           const GrB_BinaryOp op,  
3374                           const GrB_Vector u,  
3375                           const GrB_Vector v,  
3376                           const GrB_Descriptor desc);
```

#### 3377 Parameters

3378 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3379 that may be accumulated with the result of the element-wise operation. On output,  
3380 this vector holds the results of the operation.

3381 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3382 stored into the output vector **w**. The mask dimensions must match those of the  
3383 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3384 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3385 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3386 dimensions of **w**), **GrB\_NULL** should be specified.

3387 **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) The semiring, monoid, or binary operator used in the element-wise “product” operation. Depending on which type is passed, the following defines the binary operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

Monoid:  $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$ ; the identity element is ignored.

Semiring:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$ ; the additive monoid is ignored.

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

**v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the operation.

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

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

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector **w** is ready to be used in the next method of the sequence.

**GrB\_PANIC** Unknown internal error.

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

**GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3418 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3419 a call to `new` (or `dup` for vector parameters).

3420 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3421 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
 3422 responding domains of the binary operator (`op`) or accumulation  
 3423 operator, or the mask's domain is not compatible with `bool` (in the  
 3424 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3425 Description

3426 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:  
 3427  $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$ .  
 3428 Logically, this operation occurs in three steps:

3429 **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3430 and dimensions are tested for compatibility.

3431 **Compute** The indicated computations are carried out.

3432 **Output** The result is written into the output vector, possibly under control of a mask.

3433 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3434 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3435 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3436 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3437 4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3438 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)  
 3439 are tested for domain compatibility as follows:

- 3440 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3441 must be from one of the pre-defined types of Table 3.2.
- 3442 2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3443 3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3444 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3445 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3446 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
 3447 the accumulation operator.



3448 Two domains are compatible with each other if values from one domain can be cast to values in  
 3449 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3450 compatible with each other. A domain from a user-defined type is only compatible with itself. If any  
 3451 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3452 error listed above is returned.

3453 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3454 denotes copy):

- 3455 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3456 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3457 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3458 (b) If `mask  $\neq$  GrB_NULL`,
    - 3459 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 3460 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 3461 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3462 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 3463 4. Vector  $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3464 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 3465 must hold:

- 3466 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}}) = \mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{v}})$ .

3467 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
 3468 mismatch error listed above is returned.

3469 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3470 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3471 We are now ready to carry out the element-wise “product” and any additional associated operations.  
 3472 We describe this in terms of two intermediate vectors:

- 3473 •  $\tilde{\mathbf{t}}$ : The vector holding the element-wise “product” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- 3474 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3475 The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The  
 3476 value of each of its elements is computed by:

$$3477 \quad t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}))$$

3478 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

3479 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

3480 • If  $\text{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3481 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3482 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3483 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$3484 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3485

$$3486 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3487

$$3488 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3489 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3490 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
3491 using what is called a *standard vector mask and replace*. This is carried out under control of the  
3492 mask which acts as a “write mask”.

3493 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is set, then any values in  $\mathbf{w}$  on input to this operation are  
3494 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3495 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3496 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
3497 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
3498 mask are unchanged:

$$3499 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3500 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
3501 of vector  $\mathbf{w}$  is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
3502 exits with return value **GrB\_SUCCESS** and the new content of vector  $\mathbf{w}$  is as defined above but  
3503 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3504 sequence.

#### 3505 4.3.4.2 eWiseMult: Matrix variant

3506 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-  
3507 ducing a third matrix as result.

## 3508 C Syntax

```

3509         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3510                                const GrB_Matrix Mask,
3511                                const GrB_BinaryOp accum,
3512                                const GrB_Semiring op,
3513                                const GrB_Matrix A,
3514                                const GrB_Matrix B,
3515                                const GrB_Descriptor desc);
3516
3517         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3518                                const GrB_Matrix Mask,
3519                                const GrB_BinaryOp accum,
3520                                const GrB_Monoid op,
3521                                const GrB_Matrix A,
3522                                const GrB_Matrix B,
3523                                const GrB_Descriptor desc);
3524
3525         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3526                                const GrB_Matrix Mask,
3527                                const GrB_BinaryOp accum,
3528                                const GrB_BinaryOp op,
3529                                const GrB_Matrix A,
3530                                const GrB_Matrix B,
3531                                const GrB_Descriptor desc);

```

## 3532 Parameters

3533 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3534 that may be accumulated with the result of the element-wise operation. On output,  
3535 the matrix holds the results of the operation.

3536 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3537 stored into the output matrix C. The mask dimensions must match those of the  
3538 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3539 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types  
3540 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3541 dimensions of C), `GrB_NULL` should be specified.

3542 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3543 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3544 specified.

3545 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
3546 operation. Depending on which type is passed, the following defines the binary  
3547 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3548 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
 3549 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-  
 3550 nored.  
 3551 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  
 3552 is ignored.

3553 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3554 operation.

3555 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 3556 operation.

3557 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 3558 should be specified. Non-default field/value pairs are listed as follows:  
 3559

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.

## 3561 Return Values

3562 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 3563 blocking mode, this indicates that the compatibility tests on di-  
 3564 mensions and domains for the input arguments passed successfully.  
 3565 Either way, output matrix C is ready to be used in the next method  
 3566 of the sequence.

3567 GrB\_PANIC Unknown internal error.

3568 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 3569 GraphBLAS objects (input or output) is in an invalid state caused  
 3570 by a previous execution error. Call GrB\_error() to access any error  
 3571 messages generated by the implementation.

3572 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3573 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3574 a call to new (or Matrix\_dup for matrix parameters).

3575 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

3576 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3577 corresponding domains of the binary operator (`op`) or accumulation  
 3578 operator, or the mask's domain is not compatible with `bool` (in the  
 3579 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3580 Description

3581 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:  
 3582  $C = A \otimes B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$ .  
 3583 Logically, this operation occurs in three steps:

3584 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3585 and dimensions are tested for compatibility.

3586 **Compute** The indicated computations are carried out.

3587 **Output** The result is written into the output matrix, possibly under control of a mask.

3588 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3589 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3590 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)
- 3591 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3592 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3593 The argument matrices, the “product” operator (`op`), and the accumulation operator (if provided)  
 3594 are tested for domain compatibility as follows:

- 3595 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3596 must be from one of the pre-defined types of Table 3.2.
- 3597 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3598 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3599 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3600 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3601 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3602 the accumulation operator.

3603 Two domains are compatible with each other if values from one domain can be cast to values in  
 3604 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3605 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3606 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3607 error listed above is returned.

3608 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3609 ( $\leftarrow$  denotes copy):

- 3610 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3611 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3612 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3613  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3614 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3615 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3616  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3617 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3618  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$ .
  - 3619 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3620 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3621 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3622 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3623 must hold:

- 3624 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
- 3625 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

3626 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
 3627 mismatch error listed above is returned.

3628 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3629 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3630 We are now ready to carry out the element-wise “product” and any additional associated operations.  
 3631 We describe this in terms of two intermediate matrices:

- 3632 •  $\tilde{\mathbf{T}}$ : The matrix holding the element-wise product of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3633 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3634 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$   
 3635 is created. The value of each of its elements is computed by

$$3636 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3637 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

3638 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

3639 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3640 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3641 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 3642 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$3643 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3644 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3645 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3646 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3649 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3650 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3651 mask which acts as a “write mask”.

3652 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3653 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3654 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3655 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3656 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3657 mask are unchanged:

$$3658 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3659 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 3660 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 3661 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 3662 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3663 sequence.

#### 3664 4.3.5 eWiseAdd: Element-wise addition

3665 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation  
 3666 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of  
 3667 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”  
 3668 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3669 the set of values from the resulting index set.

#### 3670 4.3.5.1 eWiseAdd: Vector variant

3671 Perform element-wise (general) addition on the elements of two vectors, producing a third vector  
3672 as result.

#### 3673 C Syntax

```
3674     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3675                          const GrB_Vector mask,  
3676                          const GrB_BinaryOp accum,  
3677                          const GrB_Semiring op,  
3678                          const GrB_Vector u,  
3679                          const GrB_Vector v,  
3680                          const GrB_Descriptor desc);  
3681  
3682     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3683                          const GrB_Vector mask,  
3684                          const GrB_BinaryOp accum,  
3685                          const GrB_Monoid op,  
3686                          const GrB_Vector u,  
3687                          const GrB_Vector v,  
3688                          const GrB_Descriptor desc);  
3689  
3690     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3691                          const GrB_Vector mask,  
3692                          const GrB_BinaryOp accum,  
3693                          const GrB_BinaryOp op,  
3694                          const GrB_Vector u,  
3695                          const GrB_Vector v,  
3696                          const GrB_Descriptor desc);
```

#### 3697 Parameters

3698 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3699 that may be accumulated with the result of the element-wise operation. On output,  
3700 this vector holds the results of the operation.

3701 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3702 stored into the output vector **w**. The mask dimensions must match those of the  
3703 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3704 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3705 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3706 dimensions of **w**), **GrB\_NULL** should be specified.

3707 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**



3708 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3709 specified.

3710 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3711 operation. Depending on which type is passed, the following defines the binary  
3712 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3713 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3714 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-  
3715 nored.

3716 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-  
3717 nary op and additive identity are ignored.

3718 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3719 operation.

3720 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
3721 operation.

3722 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
3723 should be specified. Non-default field/value pairs are listed as follows:  
3724

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 Return Values

3727 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
3728 blocking mode, this indicates that the compatibility tests on di-  
3729 mensions and domains for the input arguments passed successfully.  
3730 Either way, output vector w is ready to be used in the next method  
3731 of the sequence.

3732 GrB\_PANIC Unknown internal error.

3733 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
3734 GraphBLAS objects (input or output) is in an invalid state caused  
3735 by a previous execution error. Call GrB\_error() to access any error  
3736 messages generated by the implementation.

3737 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3738 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
3739 a call to `new` (or `dup` for vector parameters).

3740 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3741 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
3742 responding domains of the binary operator (`op`) or accumulation  
3743 operator, or the mask's domain is not compatible with `bool` (in the  
3744 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3745 Description

3746 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors:  $w =$   
3747  $u \oplus v$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot (u \oplus v)$ . Logically,  
3748 this operation occurs in three steps:

3749 **Setup** The internal vectors and mask used in the computation are formed and their domains  
3750 and dimensions are tested for compatibility.

3751 **Compute** The indicated computations are carried out.

3752 **Output** The result is written into the output vector, possibly under control of a mask.

3753 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3754 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3755 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3756 3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3757 4.  $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3758 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are  
3759 tested for domain compatibility as follows:

- 3760 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
3761 must be from one of the pre-defined types of Table 3.2.
- 3762 2.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3763 3.  $\mathbf{D}(v)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3764 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3765 5.  $\mathbf{D}(u)$  and  $\mathbf{D}(v)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3766 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3767 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
3768 the accumulation operator.

3769 Two domains are compatible with each other if values from one domain can be cast to values in  
 3770 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3771 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 3772 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch  
 3773 error listed above is returned.

3774 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3775 denotes copy):

- 3776 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3777 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3778 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 3779 (b) If `mask  $\neq$  GrB_NULL`,
    - 3780 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 3781 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 3782 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3783 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 3784 4. Vector  $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3785 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 3786 must hold:

- 3787 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$ .

3788 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension  
 3789 mismatch error listed above is returned.

3790 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3791 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3792 We are now ready to carry out the element-wise “sum” and any additional associated operations.  
 3793 We describe this in terms of two intermediate vectors:

- 3794 •  $\tilde{\mathbf{t}}$ : The vector holding the element-wise “sum” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- 3795 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3796 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  
 3797 value of each of its elements is computed by:

$$\begin{aligned}
 3798 \quad t_i &= (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})) \\
 3799 \quad t_i &= \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))) \\
 3800
 \end{aligned}$$

3801  
3802

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3803

where the difference operator in the previous expressions refers to set difference.

3804

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

3805

- If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

3806

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

3807

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

3808

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  $\tilde{\mathbf{t}}$ .

3809

3810

$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

3811

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

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

3816

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”.

3817

3818

3819

- 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,

3820

3821

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

3822

- 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:

3823

3824

3825

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

3826

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.

3827

3828

3829

3830

3831

#### 4.3.5.2 eWiseAdd: Matrix variant

3832

Perform element-wise (general) addition on the elements of two matrices, producing a third matrix as result.

3833

## 3834 C Syntax

```

3835         GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3836                               const GrB_Matrix Mask,
3837                               const GrB_BinaryOp accum,
3838                               const GrB_Semiring op,
3839                               const GrB_Matrix A,
3840                               const GrB_Matrix B,
3841                               const GrB_Descriptor desc);
3842
3843         GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3844                               const GrB_Matrix Mask,
3845                               const GrB_BinaryOp accum,
3846                               const GrB_Monoid op,
3847                               const GrB_Matrix A,
3848                               const GrB_Matrix B,
3849                               const GrB_Descriptor desc);
3850
3851         GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3852                               const GrB_Matrix Mask,
3853                               const GrB_BinaryOp accum,
3854                               const GrB_BinaryOp op,
3855                               const GrB_Matrix A,
3856                               const GrB_Matrix B,
3857                               const GrB_Descriptor desc);

```

## 3858 Parameters

3859     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3860     that may be accumulated with the result of the element-wise operation. On output,  
3861     the matrix holds the results of the operation.

3862     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3863     stored into the output matrix C. The mask dimensions must match those of the  
3864     matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3865     of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types  
3866     in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
3867     dimensions of C), `GrB_NULL` should be specified.

3868     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3869     entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3870     specified.

3871     **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3872     operation. Depending on which type is passed, the following defines the binary  
3873     operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3874 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
 3875 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-  
 3876 nored.  
 3877 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-  
 3878 nary op and additive identity are ignored.

3879 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3880 operation.

3881 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 3882 operation.

3883 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 3884 should be specified. Non-default field/value pairs are listed as follows:  
 3885

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.

## 3887 Return Values

3888 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 3889 blocking mode, this indicates that the compatibility tests on di-  
 3890 mensions and domains for the input arguments passed successfully.  
 3891 Either way, output matrix C is ready to be used in the next method  
 3892 of the sequence.

3893 GrB\_PANIC Unknown internal error.

3894 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 3895 GraphBLAS objects (input or output) is in an invalid state caused  
 3896 by a previous execution error. Call GrB\_error() to access any error  
 3897 messages generated by the implementation.

3898 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3899 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3900 a call to new (or Matrix\_dup for matrix parameters).

3901 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

3902 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3903 corresponding domains of the binary operator ( $\text{op}$ ) or accumulation  
 3904 operator, or the mask's domain is not compatible with `bool` (in the  
 3905 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3906 Description

3907 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:  
 3908  $C = A \oplus B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \oplus B)$ .  
 3909 Logically, this operation occurs in three steps:

3910 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3911 and dimensions are tested for compatibility.

3912 **Compute** The indicated computations are carried out.

3913 **Output** The result is written into the output matrix, possibly under control of a mask.

3914 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3915 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3916 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)
- 3917 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3918 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3919 The argument matrices, the “sum” operator ( $\text{op}$ ), and the accumulation operator (if provided) are  
 3920 tested for domain compatibility as follows:

- 3921 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3922 must be from one of the pre-defined types of Table 3.2.
- 3923 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3924 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3925 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3926 5.  $\mathbf{D}(A)$  and  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3927 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3928 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of  $\text{op}$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3929 the accumulation operator.

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

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

1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - (b) If `Mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
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.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

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

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

We are now ready to carry out the element-wise “sum” and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\tilde{\mathbf{T}}$ : The matrix holding the element-wise sum of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.



3961 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$   
 3962 is created. The value of each of its elements is computed by

$$\begin{aligned}
 3963 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\
 3964 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\
 3965 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})))
 \end{aligned}$$

3968 where the difference operator in the previous expressions refers to set difference.

3969 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3970 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3971 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3972 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3973 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 3974 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned}
 3975 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 3976 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 3977 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 3978 \quad & \\
 3979 \quad &
 \end{aligned}$$

3980 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3981 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3982 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3983 mask which acts as a “write mask”.

- 3984 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3985 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3986 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3987 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3988 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3989 mask are unchanged:

$$3990 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3991 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 3992 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 3993 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 3994 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3995 sequence.

### 3996 4.3.6 extract: Selecting sub-graphs

3997 Extract a subset of a matrix or vector.

#### 3998 4.3.6.1 extract: Standard vector variant

3999 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector  
4000 whose size is equal to the number of indices.

### 4001 C Syntax

```
4002         GrB_Info GrB_extract(GrB_Vector          w,  
4003                             const GrB_Vector    mask,  
4004                             const GrB_BinaryOp   accum,  
4005                             const GrB_Vector    u,  
4006                             const GrB_Index     *indices,  
4007                             GrB_Index           nindices,  
4008                             const GrB_Descriptor desc);
```

### 4009 Parameters

4010 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4011 that may be accumulated with the result of the extract operation. On output, this  
4012 vector holds the results of the operation.

4013 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4014 stored into the output vector **w**. The mask dimensions must match those of the  
4015 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4016 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4017 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4018 dimensions of **w**), **GrB\_NULL** should be specified.

4019 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4020 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4021 specified.

4022 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4023 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of  
4024 elements from **u** that are extracted. If all elements of **u** are to be extracted in order  
4025 from 0 to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution  
4026 mode and return value, this array may be manipulated by the caller after this  
4027 operation returns without affecting any deferred computations for this operation.

4028 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4029 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4030 should be specified. Non-default field/value pairs are listed as follows:

4031

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <b>w</b> 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 <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .

4032

## 4033 Return Values

4034 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 4035 blocking mode, this indicates that the compatibility tests on  
 4036 dimensions and domains for the input arguments passed suc-  
 4037 cessfully. Either way, output vector **w** is ready to be used in the  
 4038 next method of the sequence.

4039 GrB\_PANIC Unknown internal error.

4040 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 4041 opaque GraphBLAS objects (input or output) is in an invalid  
 4042 state caused by a previous execution error. Call GrB\_error() to  
 4043 access any error messages generated by the implementation.

4044 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

4045 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
 4046 by a call to **new** (or **dup** for vector parameters).

4047 GrB\_INDEX\_OUT\_OF\_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-  
 4048 blocking mode, this error can be deferred.

4049 GrB\_DIMENSION\_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices**  $\neq$  **size(w)**.

4050 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with each  
 4051 other or the corresponding domains of the accumulation oper-  
 4052 ator, or the mask's domain is not compatible with **bool** (in the  
 4053 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

4054 GrB\_NULL\_POINTER Argument **row\_indices** is a NULL pointer.

## 4055 Description

4056 This variant of GrB\_extract computes the result of extracting a subset of locations from a Graph-  
 4057 BLAS vector in a specific order: **w** = **u(indices)**; or, if an optional binary accumulation operator

4058  $(\odot)$  is provided,  $w = w \odot u(\text{indices})$ . More explicitly:

$$4059 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4060 Logically, this operation occurs in three steps:

4061     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4062             and dimensions are tested for compatibility.

4063     **Compute** The indicated computations are carried out.

4064     **Output** The result is written into the output vector, possibly under control of a mask.

4065 Up to three argument vectors are used in this GrB\_extract operation:

- 4066     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4067     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4068     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4069 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4070 bility as follows:

- 4071     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4072         must be from one of the pre-defined types of Table 3.2.
- 4073     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(u)$ .
- 4074     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4075         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4076         mulation operator.

4077 Two domains are compatible with each other if values from one domain can be cast to values in  
4078 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4079 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4080 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4081 error listed above is returned.

4082 From the arguments, the internal vectors, mask, and index array used in the computation are  
4083 formed ( $\leftarrow$  denotes copy):

- 4084     1. Vector  $\tilde{w} \leftarrow w$ .
- 4085     2. One-dimensional mask,  $\tilde{m}$ , is computed from argument `mask` as follows:  
4086         (a) If `mask = GrB_NULL`, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .

- 4087 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
 4088 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
 4089 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
 4090 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
 4091 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .  
 4092 4. The internal index array,  $\widetilde{\mathbf{I}}$ , is computed from argument indices as follows:  
 4093 (a) If  $\text{indices} = \text{GrB\_ALL}$ , then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .  
 4094 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4095 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 4096 must hold:

- 4097 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$   
 4098 2.  $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$ .

4099 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4100 match error listed above is returned.

4101 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4102 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4103 We are now ready to carry out the extract and any additional associated operations. We describe  
 4104 this in terms of two intermediate vectors:

- 4105 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from  $\widetilde{\mathbf{u}}$  in their destination locations relative to  $\widetilde{\mathbf{w}}$ .
- 4106 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4107 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$4108 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4109 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  
 4110 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`  
 4111 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
 4112 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4113 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 4114 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 4115 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$4116 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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

## 4183 Return Values

4184	<b>GrB_SUCCESS</b>	In blocking mode, the operation completed successfully. In non-
4185		blocking mode, this indicates that the compatibility tests on
4186		dimensions and domains for the input arguments passed suc-
4187		cessfully. Either way, output matrix C is ready to be used in the
4188		next method of the sequence.
4189	<b>GrB_PANIC</b>	Unknown internal error.
4190	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
4191		opaque GraphBLAS objects (input or output) is in an invalid
4192		state caused by a previous execution error. Call <code>GrB_error()</code> to
4193		access any error messages generated by the implementation.
4194	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
4195	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
4196		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4197	<b>GrB_INDEX_OUT_OF_BOUNDS</b>	A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or
4198		a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In
4199		non-blocking mode, this error can be deferred.
4200	<b>GrB_DIMENSION_MISMATCH</b>	Mask and C dimensions are incompatible, <code>nrows</code> $\neq$ <code>nrows(C)</code> , or
4201		<code>ncols</code> $\neq$ <code>ncols(C)</code> .
4202	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various matrices are incompatible with each
4203		other or the corresponding domains of the accumulation oper-
4204		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4205		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4206	<b>GrB_NULL_POINTER</b>	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4207		is a NULL pointer, or both.

## 4208 Description

4209 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified  
 4210 rows and columns of a GraphBLAS matrix in a specific order:  $C = A(\text{row\_indices}, \text{col\_indices})$ ; or,  
 4211 if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A(\text{row\_indices}, \text{col\_indices})$ .  
 4212 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4214 Logically, this operation occurs in three steps:

4215 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 4216 and dimensions are tested for compatibility.



4217 **Compute** The indicated computations are carried out.

4218 **Output** The result is written into the output matrix, possibly under control of a mask.

4219 Up to three argument matrices are used in the `GrB_extract` operation:

- 4220 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4221 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)
- 4222 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4223 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
4224 ibility as follows:

- 4225 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
4226 must be from one of the pre-defined types of Table 3.2.
- 4227 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4228 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4229 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4230 mulation operator.

4231 Two domains are compatible with each other if values from one domain can be cast to values in  
4232 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4233 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4234 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4235 error listed above is returned.

4236 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are  
4237 formed ( $\leftarrow$  denotes copy):

- 4238 1. Matrix  $\tilde{C} \leftarrow C$ .
- 4239 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:
  - 4240 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
4241  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 4242 (b) If `Mask`  $\neq$  `GrB_NULL`,
    - 4243 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
4244  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 4245 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
4246  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 4247 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .
- 4248 3. Matrix  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$ .

- 4249 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4250 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .
- 4251 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \text{nrows}$ .
- 4252 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4253 (a) If `col_indices` = `GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .
- 4254 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

4255 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4256 must hold:

- 4257 1.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$ .
- 4258 2.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$ .
- 4259 3.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$ .
- 4260 4.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$ .

4261 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4262 match error listed above is returned.

4263 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4264 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4265 We are now ready to carry out the extract and any additional associated operations. We describe  
4266 this in terms of two intermediate matrices:

- 4267 •  $\tilde{\mathbf{T}}$ : The matrix holding the extraction from  $\tilde{\mathbf{A}}$ .
- 4268 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4269 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$4270 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \\ \{(i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4271 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  $\tilde{\mathbf{J}}$   
4272 array is not in the range  $[0, \text{ncols}(\tilde{\mathbf{A}}))$ , the execution of `GrB_extract` ends and the index out-of-  
4273 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
4274 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
4275 this point forward in the sequence.

4276 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4277 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

4278 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4279 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4280 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
4281 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4282 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4283 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4284 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4287 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4288 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
4289 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
4290 mask which acts as a “write mask”.

4291 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
4292 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$4293 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4294 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
4295 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
4296 mask are unchanged:

$$4297 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4298 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4299 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4300 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
4301 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
4302 sequence.

#### 4303 4.3.6.3 extract: Column (and row) variant

4304 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the  
4305 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as  
4306 well.

## 4307 C Syntax

```

4308      GrB_Info GrB_extract(GrB_Vector      w,
4309                          const GrB_Vector  mask,
4310                          const GrB_BinaryOp accum,
4311                          const GrB_Matrix  A,
4312                          const GrB_Index   *row_indices,
4313                          GrB_Index        nrows,
4314                          GrB_Index        col_index,
4315                          const GrB_Descriptor desc);

```

## 4316 Parameters

4317     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4318     that may be accumulated with the result of the extract operation. On output, this  
4319     vector holds the results of the operation.

4320     **mask** (IN) An optional “write” mask that controls which results from this operation are  
4321     stored into the output vector **w**. The mask dimensions must match those of the  
4322     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4323     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4324     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4325     dimensions of **w**), **GrB\_NULL** should be specified.

4326     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4327     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4328     specified.

4329     **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4330     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations  
4331     within the specified column of **A** from which elements are extracted. If elements in  
4332     all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless  
4333     of execution mode and return value, this array may be manipulated by the caller  
4334     after this operation returns without affecting any deferred computations for this  
4335     operation.

4336     **nrows** (IN) The number of indices in the **row\_indices** array. Must be equal to **size(w)**.

4337     **col\_index** (IN) The index of the column of **A** from which to extract values. It must be in the  
4338     range  $[0, \mathbf{ncols}(A))$ .

4339     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4340     should be specified. Non-default field/value pairs are listed as follows:

4341

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <b>w</b> 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 <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .
A	GrB_INP0	GrB_TRAN	Use transpose of <b>A</b> for the operation.

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output 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

4369 an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot A(:, \text{col\_index})(\text{row\_indices})$ .  
 4370 More explicitly:

$$4371 \quad \begin{aligned} w(i) &= A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4372 Logically, this operation occurs in three steps:

4373     **Setup** The internal matrices, vectors, and mask used in the computation are formed and their  
 4374 domains and dimensions are tested for compatibility.

4375     **Compute** The indicated computations are carried out.

4376     **Output** The result is written into the output vector, possibly under control of a mask.

4377 Up to three argument vectors and matrices are used in this GrB\_extract operation:

- 4378 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4379 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4380 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4381 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain  
 4382 compatibility as follows:

- 4383 1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
 4384 must be from one of the pre-defined types of Table 3.2.
- 4385 2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(A)$ .
- 4386 3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4387 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4388 mulation operator.

4389 Two domains are compatible with each other if values from one domain can be cast to values in  
 4390 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4391 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4392 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
 4393 error listed above is returned.

4394 From the arguments, the internal vector, matrix, mask, and index array used in the computation  
 4395 are formed ( $\leftarrow$  denotes copy):

- 4396 1. Vector  $\tilde{w} \leftarrow w$ .
- 4397 2. One-dimensional mask,  $\tilde{m}$ , is computed from argument **mask** as follows:  
 4398 (a) If **mask** = GrB\_NULL, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .

4399 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
 4400 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
 4401 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
 4402 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
 4403 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .  
 4404 4. The internal row index array,  $\widetilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:  
 4405 (a) If `indices = GrB_ALL`, then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .  
 4406 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$ .

4407 The internal vector, `mask`, and index array are checked for dimension compatibility. The following  
 4408 conditions must hold:

- 4409 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4410 2.  $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$ .

4411 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4412 match error listed above is returned.

4413 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4414 1.  $0 \leq \text{col\_index} < \text{ncols}(\mathbf{A})$

4415 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above  
 4416 is returned.

4417 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4418 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4419 We are now ready to carry out the extract and any additional associated operations. We describe  
 4420 this in terms of two intermediate vectors:

- 4421 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from a column of  $\widetilde{\mathbf{A}}$ .
- 4422 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4423 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$4424 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col\_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col\_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4425 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`  
 4426 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,  
 4427 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result  
 4428 vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4429 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

4430 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

4431 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4432 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4433 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 4434 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4435 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

$$4436 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4437 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4440 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4441 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 4442 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 4443 mask which acts as a “write mask”.

4444 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
 4445 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4446 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4447 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 4448 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 4449 mask are unchanged:

$$4450 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4451 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 4452 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 4453 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
 4454 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 4455 sequence.

### 4456 4.3.7 assign: Modifying sub-graphs

4457 Assign the contents of a subset of a matrix or vector.

#### 4458 4.3.7.1 assign: Standard vector variant

4459 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.  
 4460 The size of the input vector is the same size as the index array provided.



## 4461 C Syntax

```

4462         GrB_Info GrB_assign(GrB_Vector      w,
4463                             const GrB_Vector mask,
4464                             const GrB_BinaryOp accum,
4465                             const GrB_Vector u,
4466                             const GrB_Index *indices,
4467                             GrB_Index      nindices,
4468                             const GrB_Descriptor desc);

```

## 4469 Parameters

4470        **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4471        that may be accumulated with the result of the assign operation. On output, this  
4472        vector holds the results of the operation.

4473        **mask** (IN) An optional “write” mask that controls which results from this operation are  
4474        stored into the output vector **w**. The mask dimensions must match those of the  
4475        vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4476        of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4477        in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4478        dimensions of **w**), **GrB\_NULL** should be specified.

4479        **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4480        entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4481        specified.

4482        **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4483        **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4484        **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0  
4485        to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4486        and return value, this array may be manipulated by the caller after this operation  
4487        returns without affecting any deferred computations for this operation. If this  
4488        array contains duplicate values, it implies in assignment of more than one value to  
4489        the same location which leads to undefined results.

4490        **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4491        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4492        should be specified. Non-default field/value pairs are listed as follows:

4493

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

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

**GrB\_PANIC** Unknown internal error.

**GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call **GrB\_error()** to access any error messages generated by the implementation.

**GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

**GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector parameters).

**GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indices** is greater than or equal to **size(w)**. In non-blocking mode, this can be reported as an execution error.

**GrB\_DIMENSION\_MISMATCH** mask and w dimensions are incompatible, or **nindices**  $\neq$  **size(u)**.

**GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

**GrB\_NULL\_POINTER** Argument **indices** is a NULL pointer.

## Description

This variant of **GrB\_assign** computes the result of assigning elements from a source GraphBLAS vector to a destination GraphBLAS vector in a specific order:  $w(\text{indices}) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w(\text{indices}) = w(\text{indices}) \odot u$ . More explicitly:

$$w(\text{indices}[i]) = u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or}$$

$$w(\text{indices}[i]) = w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.$$

4522 Logically, this operation occurs in three steps:

4523     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4524             and dimensions are tested for compatibility.

4525     **Compute** The indicated computations are carried out.

4526     **Output** The result is written into the output vector, possibly under control of a mask.

4527 Up to three argument vectors are used in the `GrB_assign` operation:

- 4528     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4529     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4530     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4531 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4532 bility as follows:

- 4533     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
4534         must be from one of the pre-defined types of Table 3.2.
- 4535     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4536     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
4537         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
4538         mulation operator.

4539 Two domains are compatible with each other if values from one domain can be cast to values in  
4540 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4541 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4542 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
4543 error listed above is returned.

4544 From the arguments, the internal vectors, mask and index array used in the computation are formed  
4545 ( $\leftarrow$  denotes copy):

- 4546     1. Vector  $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 4547     2. One-dimensional mask,  $\widetilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 4548         (a) If `mask` = `GrB_NULL`, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 4549         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 4550             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 4551             ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 4552         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .

4553 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4554 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument indices as follows:

4555 (a) If `indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i$ ,  $\forall i : 0 \leq i < \text{nindices}$ .

4556 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{indices}[i]$ ,  $\forall i : 0 \leq i < \text{nindices}$ .

4557 The internal vector and mask are checked for dimension compatibility. The following conditions

4558 must hold:

4559 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4560 2.  $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$ .

4561 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-

4562 match error listed above is returned.

4563 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

4564 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4565 We are now ready to carry out the assign and any additional associated operations. We describe

4566 this in terms of two intermediate vectors:

4567 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{w}}$ .

4568 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4569 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

4570 
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4571 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{w}}$ , computation

4572 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`

4573 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the

4574 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4575 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

4576 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

4577 
$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4578 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure

4579 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

4580 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4581 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of

4582 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

4583 
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4584 
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4585 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,
```

```

4620         const GrB_Index      *row_indices,
4621         GrB_Index             nrows,
4622         const GrB_Index      *col_indices,
4623         GrB_Index             ncols,
4624         const GrB_Descriptor desc);

```

## 4625 Parameters

4626     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
4627     that may be accumulated with the result of the assign operation. On output, the  
4628     matrix holds the results of the operation.

4629     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
4630     stored into the output matrix **C**. The mask dimensions must match those of the  
4631     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4632     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
4633     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4634     dimensions of **C**), **GrB\_NULL** should be specified.

4635     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4636     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4637     specified.

4638     **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4639     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
4640     that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,  
4641     then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
4642     this array may be manipulated by the caller after this operation returns without  
4643     affecting any deferred computations for this operation. If this array contains du-  
4644     plicate values, it implies assignment of more than one value to the same location  
4645     which leads to undefined results.

4646     **nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(A)**  
4647     if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4648     **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns  
4649     of **C** that are assigned. If all columns of **C** are to be assigned in order from 0  
4650     to **ncols** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4651     and return value, this array may be manipulated by the caller after this operation  
4652     returns without affecting any deferred computations for this operation. If this  
4653     array contains duplicate values, it implies assignment of more than one value to  
4654     the same location which leads to undefined results.

4655     **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **ncols(A)** if **A** is  
4656     not transposed, or equal to **nrows(A)** if **A** is transposed.

4657  
4658  
4659

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:

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.

## 4661 Return Values

4662  
4663  
4664  
4665  
4666

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.

4667

GrB\_PANIC Unknown internal error.

4668  
4669  
4670  
4671

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.

4672

GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

4673  
4674

GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix\_dup for matrix parameters).

4675  
4676  
4677

GrB\_INDEX\_OUT\_OF\_BOUNDS A value in row\_indices is greater than or equal to nrows(C), or a value in col\_indices is greater than or equal to ncols(C). In non-blocking mode, this can be reported as an execution error.

4678  
4679

GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, nrow  $\neq$  nrow(A), or ncols  $\neq$  ncols(A).

4680  
4681  
4682  
4683

GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

4684  
4685

GrB\_NULL\_POINTER Either argument row\_indices is a NULL pointer, argument col\_indices is a NULL pointer, or both.

## 4686 Description

4687 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows  
 4688 and columns in `C` in a specified order:  $C(\text{row\_indices}, \text{col\_indices}) = A$ ; or, if an optional binary  
 4689 accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot$   
 4690 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4691 & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot A(i, j), \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4692 Logically, this operation occurs in three steps:

4693     **Setup** The internal matrices and mask used in the computation are formed and their domains  
 4694     and dimensions are tested for compatibility.

4695     **Compute** The indicated computations are carried out.

4696     **Output** The result is written into the output matrix, possibly under control of a mask.

4697 Up to three argument matrices are used in the `GrB_assign` operation:

- 4698     1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4699     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)
- 4700     3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4701 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
 4702 ibility as follows:

- 4703     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 4704     must be from one of the pre-defined types of Table 3.2.
- 4705     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4706     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4707     of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4708     mulation operator.

4709 Two domains are compatible with each other if values from one domain can be cast to values in  
 4710 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4711 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4712 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 4713 error listed above is returned.

4714 From the arguments, the internal matrices, mask, and index arrays used in the computation are  
 4715 formed ( $\leftarrow$  denotes copy):



- 4716 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 4717 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
- 4718 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
4719  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 4720 (b) If `Mask  $\neq$  GrB_NULL`,
- 4721 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
4722  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
- 4723 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
4724  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
- 4725 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4726 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4727 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4728 (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4729 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4730 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4731 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
- 4732 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

4733 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4734 must hold:

- 4735 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 4736 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 4737 3.  $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$ .
- 4738 4.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$ .

4739 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4740 match error listed above is returned.

4741 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4742 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4743 We are now ready to carry out the assign and any additional associated operations. We describe  
4744 this in terms of two intermediate vectors:

- 4745 •  $\tilde{\mathbf{T}}$ : The matrix holding the contents from  $\tilde{\mathbf{A}}$  in their destination locations relative to  $\tilde{\mathbf{C}}$ .
- 4746 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4747 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$4748 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{( \tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j) ) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \} \}.$$

4749 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  
 4750  $\tilde{\mathbf{J}}$  array is not in the range  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-  
 4751 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
 4752 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
 4753 this point forward in the sequence.

4754 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 4755 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$4756 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4757 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})) \} \}.$$

4758 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 4759 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  
 4760 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

4761 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4762 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4763 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4764 \\ 4765 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4766 where the difference operator refers to set difference.

- 4767 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4768 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}}) \} \}.$$

4769 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4770 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4771 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4772 \\ 4773 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4774 \\ 4775 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4776 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4777 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 4778 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 4779 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

4813 bool or any of the predefined “built-in” types in Table 3.2. If the default mask  
 4814 is desired (i.e., a mask that is all true with the dimensions of a column of C),  
 4815 GrB\_NULL should be specified.

4816 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
 4817 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
 4818 specified.

4819 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column  
 4820 of C.

4821 **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
 4822 the specified column of C that are to be assigned. If all elements of the column  
 4823 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB\_ALL should  
 4824 be specified. Regardless of execution mode and return value, this array may be  
 4825 manipulated by the caller after this operation returns without affecting any de-  
 4826 ferred computations for this operation. If this array contains duplicate values, it  
 4827 implies in assignment of more than one value to the same location which leads to  
 4828 undefined results.

4829 **nrows** (IN) The number of values in **row\_indices** array. Must be equal to **size(u)**.

4830 **col\_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4831 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4832 should be specified. Non-default field/value pairs are listed as follows:

4833

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 <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

4834

## 4835 Return Values

4836 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4837 blocking mode, this indicates that the compatibility tests on  
 4838 dimensions and domains for the input arguments passed suc-  
 4839 cessfully. Either way, output matrix C is ready to be used in the  
 4840 next method of the sequence.

4841 **GrB\_PANIC** Unknown internal error.

4842           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
4843   opaque GraphBLAS objects (input or output) is in an invalid  
4844   state caused by a previous execution error. Call `GrB_error()` to  
4845   access any error messages generated by the implementation.

4846           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

4847           GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
4848   by a call to `new` (or `dup` for vector or matrix parameters).

4849           GrB\_INVALID\_INDEX `col_index` is outside the allowable range (i.e., greater than `ncols(C)`).

4850           GrB\_INDEX\_OUT\_OF\_BOUNDS A value in `row_indices` is greater than or equal to `nrows(C)`. In  
4851   non-blocking mode, this can be reported as an execution error.

4852           GrB\_DIMENSION\_MISMATCH `mask` size and number of rows in `C` are not the same, or `nrows`  $\neq$   
4853   `size(u)`.

4854           GrB\_DOMAIN\_MISMATCH The domains of the matrix and vector are incompatible with  
4855   each other or the corresponding domains of the accumulation  
4856   operator, or the mask's domain is not compatible with `bool` (in  
4857   the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4858           GrB\_NULL\_POINTER Argument `row_indices` is a NULL pointer.

## 4859 Description

4860 This variant of `GrB_assign` computes the result of assigning a subset of locations in a column of a  
4861 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:  
4862  $C(:, col\_index) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(:, col\_index) =$   
4863  $C(:, col\_index) \odot u$ . Taking order of `row_indices` into account, it is more explicitly written as:

$$4864 \quad C(row\_indices[i], col\_index) = u(i), \forall i : 0 \leq i < nrows, \text{ or}$$

$$C(row\_indices[i], col\_index) = C(row\_indices[i], col\_index) \odot u(i), \forall i : 0 \leq i < nrows.$$

4865 Logically, this operation occurs in three steps:

4866       **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
4867                   domains and dimensions are tested for compatibility.

4868       **Compute** The indicated computations are carried out.

4869       **Output** The result is written into the output matrix, possibly under control of a mask.

4870 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 4871 1.  $C = \langle D(C), nrows(C), ncols(C), L(C) = \{(i, j, C_{ij})\} \rangle$
- 4872 2.  $mask = \langle D(mask), size(mask), L(mask) = \{(i, m_i)\} \rangle$  (optional)

4873 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4874 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
4875 compatibility as follows:

4876 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4877 must be from one of the pre-defined types of Table 3.2.

4878 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .

4879 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4880 of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4881 mulation operator.

4882 Two domains are compatible with each other if values from one domain can be cast to values in  
4883 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4884 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4885 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
4886 error listed above is returned.

4887 The `col_index` parameter is checked for a valid value. The following condition must hold:

4888 1.  $0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{C})$

4889 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
4890 is returned.

4891 From the arguments, the internal vectors, `mask`, and index array used in the computation are  
4892 formed ( $\leftarrow$  denotes copy):

4893 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a column of  $\mathbf{C}$  as follows:

4894 
$$\tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4895 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

4896 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$ .

4897 (b) If `mask`  $\neq$  `GrB_NULL`,

4898 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

4899 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

4900 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

4901 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4902 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:

4903 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .

4904 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i]$ ,  $\forall i : 0 \leq i < \text{nrows}$ .

4905 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
4906 conditions must hold:

- 4907 1.  $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4908 2.  $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$ .

4909 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4910 match error listed above is returned.

4911 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4912 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4913 We are now ready to carry out the assign and any additional associated operations. We describe  
4914 this in terms of two intermediate vectors:

- 4915 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 4916 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4917 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4918 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4919 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
4920 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
4921 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4922 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

4923 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 4924 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$4925 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4926 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
4927 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  
4928 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4929 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4930 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$4931 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4932 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4934 where the difference operator refers to set difference.

4935 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4936 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4937 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4938 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4939 \quad z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

4940

$$4941 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

4942

$$4943 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

4944 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4945 Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final  
4946 result matrix,  $\mathbf{C}(:, \text{col\_index})$ . This is carried out under control of the mask which acts as a “write  
4947 mask”.

4948 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(:, \text{col\_index})$  on input to this  
4949 operation are deleted and the new contents of the column is given by:

$$4950 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4951 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
4952 copied into the column of the final result matrix,  $\mathbf{C}(:, \text{col\_index})$ , and elements of this column  
4953 that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} 4954 \quad \mathbf{L}(\mathbf{C}) &= \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \\ 4955 &\quad \{(i, \text{col\_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ 4956 &\quad \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

4957 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4958 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4959 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may  
4960 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 4961 4.3.7.4 assign: Row variant

4962 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the  
4963 output cannot be transposed, a different variant of `assign` is provided to assign to a column of a  
4964 matrix.



## 4965 C Syntax

```
4966         GrB_Info GrB_assign(GrB_Matrix      C,  
4967                             const GrB_Vector mask,  
4968                             const GrB_BinaryOp accum,  
4969                             const GrB_Vector u,  
4970                             GrB_Index      row_index,  
4971                             const GrB_Index *col_indices,  
4972                             GrB_Index      ncols,  
4973                             const GrB_Descriptor desc);
```

## 4974 Parameters

4975 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values  
4976 that may be accumulated with the result of the assign operation. On output, this  
4977 matrix holds the results of the operation.

4978 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4979 stored into the specified row of the output matrix **C**. The mask dimensions must  
4980 match those of a single row of the matrix **C**. If the **GrB\_STRUCTURE** descriptor  
4981 is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or  
4982 any of the predefined “built-in” types in Table 3.2. If the default mask is desired  
4983 (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB\_NULL** should  
4984 be specified.

4985 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4986 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4987 specified.

4988 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of  
4989 **C**.

4990 **row\_index** (IN) The index of the row in **C** to assign. Must be in the range  $[0, \mathbf{nrows}(\mathbf{C})]$ .

4991 **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4992 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to  
4993 be assigned in order from index 0 to  $\mathbf{ncols} - 1$ , then **GrB\_ALL** should be specified.  
4994 Regardless of execution mode and return value, this array may be manipulated by  
4995 the caller after this operation returns without affecting any deferred computations  
4996 for this operation. If this array contains duplicate values, it implies in assignment  
4997 of more than one value to the same location which leads to undefined results.

4998 **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **size(u)**.

4999 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5000 should be specified. Non-default field/value pairs are listed as follows:  
5001

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 <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .

## 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 <b>GrB_error()</b> 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 <b>new</b> (or <b>dup</b> for vector or matrix parameters).
GrB_INVALID_INDEX	<b>row_index</b> is outside the allowable range (i.e., greater than <b>nrows(C)</b> ).
GrB_INDEX_OUT_OF_BOUNDS	A value in <b>col_indices</b> is greater than or equal to <b>ncols(C)</b> . In non-blocking mode, this can be reported as an execution error.
GrB_DIMENSION_MISMATCH	<b>mask</b> size and number of columns in C are not the same, or <b>ncols</b> $\neq$ <b>size(u)</b> .
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 <b>bool</b> (in the case where <b>desc[GrB_MASK].GrB_STRUCTURE</b> is not set).
GrB_NULL_POINTER	Argument <b>col_indices</b> 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:

5030  $C(\text{row\_index}, :) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_index}, :$   
 5031  $) = C(\text{row\_index}, :) \odot u$ . Taking order of `col_indices` into account it is more explicitly written as:

5032  $C(\text{row\_index}, \text{col\_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$   
 5033  $C(\text{row\_index}, \text{col\_indices}[j]) = C(\text{row\_index}, \text{col\_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$

5033 Logically, this operation occurs in three steps:

5034     **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
 5035               domains and dimensions are tested for compatibility.

5036     **Compute** The indicated computations are carried out.

5037     **Output** The result is written into the output matrix, possibly under control of a mask.

5038 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5039     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$   
 5040     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)  
 5041     3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5042 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
 5043 compatibility as follows:

- 5044     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 5045         must be from one of the pre-defined types of Table 3.2.  
 5046     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(u)$ .  
 5047     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5048         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 5049         mulation operator.

5050 Two domains are compatible with each other if values from one domain can be cast to values in  
 5051 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5052 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5053 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 5054 error listed above is returned.

5055 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5056     1.  $0 \leq \text{row\_index} < \mathbf{nrows}(C)$

5057 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
 5058 is returned.

5059 From the arguments, the internal vectors, mask, and index array used in the computation are  
 5060 formed ( $\leftarrow$  denotes copy):

5061 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a row of  $\mathbf{C}$  as follows:

$$5062 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5063 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

5064 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$ .

5065 (b) If `mask  $\neq$  GrB_NULL`,

5066 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

5067 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

5068 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

5069 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5070 4. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:

5071 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .

5072 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

5073 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
5074 conditions must hold:

5075 1.  $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5076 2.  $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$ .

5077 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
5078 match error listed above is returned.

5079 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5080 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5081 We are now ready to carry out the assign and any additional associated operations. We describe  
5082 this in terms of two intermediate vectors:

- 5083 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 5084 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5085 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$5086 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5087 At this point, if any value of  $\tilde{\mathbf{J}}[j]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
5088 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
5089 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
5090 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

5091 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

5092 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}}$  is defined as

$$5093 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5094 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
5095 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  
5096 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5097 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5098 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$5099 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5100 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5101 where the difference operator refers to set difference.

5102 • If  $\text{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5103 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5104 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5105 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5106 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5107 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5108 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5109 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5110 Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final  
5111 result matrix,  $\mathbf{C}(\text{row\_index}, :)$ . This is carried out under control of the mask which acts as a “write  
5112 mask”.

5113 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is set, then any values in  $\mathbf{C}(\text{row\_index}, :)$  on input to this  
5114 operation are deleted and the new contents of the column is given by:

$$5115 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5116 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
5117 copied into the column of the final result matrix,  $\mathbf{C}(\text{row\_index}, :)$ , and elements of this column  
5118 that fall outside the set indicated by the mask are unchanged:

$$5119 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup$$

$$5120 \quad \{(\text{row\_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5121 \quad \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5122 In  $\text{GrB\_BLOCKING}$  mode, the method exits with return value  $\text{GrB\_SUCCESS}$  and the new content  
5123 of vector  $\mathbf{w}$  is as defined above and fully computed. In  $\text{GrB\_NONBLOCKING}$  mode, the method  
5124 exits with return value  $\text{GrB\_SUCCESS}$  and the new content of vector  $\mathbf{w}$  is as defined above but may  
5125 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5129 **4.3.7.5 assign: Constant vector variant**[Scott: NEW CONTENT]

5130 Assign the same value to a specified subset of vector elements. With the use of GrB\_ALL, the entire  
5131 destination vector can be filled with the constant.

5132 **C Syntax**

```
5133     GrB_Info GrB_assign(GrB_Vector      w,  
5134                        const GrB_Vector mask,  
5135                        const GrB_BinaryOp accum,  
5136                        <type>          val,  
5137                        const GrB_Index  *indices,  
5138                        GrB_Index        nindices,  
5139                        const GrB_Descriptor desc);
```

```
5140     GrB_Info GrB_assign(GrB_Vector      w,  
5141                        const GrB_Vector mask,  
5142                        const GrB_BinaryOp accum,  
5143                        const GrB_Scalar  s,  
5144                        const GrB_Index  *indices,  
5145                        GrB_Index        nindices,  
5146                        const GrB_Descriptor desc);
```

5147 **Parameters**

5148 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5149 that may be accumulated with the result of the assign operation. On output, this  
5150 vector holds the results of the operation.

5151 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5152 stored into the output vector w. The mask dimensions must match those of the  
5153 vector w. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
5154 of the mask vector must be of type bool or any of the predefined “built-in” types  
5155 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
5156 dimensions of w), GrB\_NULL should be specified.

5157 **accum** (IN) An optional binary operator used for accumulating entries into existing w  
5158 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
5159 specified.

5160 **val** (IN) Scalar value to assign to (a subset of) w.

5161 **s** (IN) Scalar value to assign to (a subset of) w.

5162 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5163 w that are to be assigned. If all elements of w are to be assigned in order from 0

5164 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode  
 5165 and return value, this array may be manipulated by the caller after this operation  
 5166 returns without affecting any deferred computations for this operation. In this  
 5167 variant, the specific order of the values in the array has no effect on the result.  
 5168 Unlike other variants, if there are duplicated values in this array the result is still  
 5169 defined.

5170 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If  
 5171 `nindices` is zero, the operation becomes a NO-OP.

5172 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
 5173 should be specified. Non-default field/value pairs are listed as follows:

5174

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> .

5175

## 5176 Return Values

5177 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 5178 blocking mode, this indicates that the compatibility tests on  
 5179 dimensions and domains for the input arguments passed suc-  
 5180 cessfully. Either way, output vector `w` is ready to be used in the  
 5181 next method of the sequence.

5182 **GrB\_PANIC** Unknown internal error.

5183 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 5184 opaque GraphBLAS objects (input or output) is in an invalid  
 5185 state caused by a previous execution error. Call `GrB_error()` to  
 5186 access any error messages generated by the implementation.

5187 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5188 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
 5189 by a call to `new` (or `dup` for vector parameters).

5190 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-  
 5191 blocking mode, this can be reported as an execution error.

5192 **GrB\_DIMENSION\_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less  
 5193 than `size(w)`.





5224 4. If **accum** is not **GrB\_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the  
 5225 method, must be compatible with **D<sub>in2</sub>(accum)** of the accumulation operator.

5226 Two domains are compatible with each other if values from one domain can be cast to values in  
 5227 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5228 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5229 any compatibility rule above is violated, execution of **GrB\_assign** ends and the domain mismatch  
 5230 error listed above is returned.

5231 From the arguments, the internal vectors, mask and index array used in the computation are formed  
 5232 ( $\leftarrow$  denotes copy):

- 5233 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5234 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:
  - 5235 (a) If **mask** = **GrB\_NULL**, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 5236 (b) If **mask**  $\neq$  **GrB\_NULL**,
    - 5237 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 5238 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 5239 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5240 3. Scalar  $\tilde{s} \leftarrow s$  (**GrB\_Scalar** version only).
- 5241 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument **indices** as follows:
  - 5242 (a) If **indices** = **GrB\_ALL**, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$ .
  - 5243 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$ .

5244 The internal vector and mask are checked for dimension compatibility. The following conditions  
 5245 must hold:

- 5246 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5247 2.  $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$ .

5248 If any compatibility rule above is violated, execution of **GrB\_assign** ends and the dimension mis-  
 5249 match error listed above is returned.

5250 From this point forward, in **GrB\_NONBLOCKING** mode, the method can optionally exit with  
 5251 **GrB\_SUCCESS** return code and defer any computation and/or execution error codes.

5252 We are now ready to carry out the assign and any additional associated operations. We describe  
 5253 this in terms of two intermediate vectors:

- 5254 •  $\tilde{\mathbf{t}}$ : The vector holding the copies of the scalar, either **val** or  $\tilde{s}$ , in their destination locations  
 5255 relative to  $\tilde{\mathbf{w}}$ .

5256 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5257 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows. If a non-opaque scalar  $\mathbf{val}$  is provided:

$$5258 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5259 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$5260 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5261 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$5262 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5263 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  
 5264 is not in the range  $[0, \mathbf{size}(\tilde{\mathbf{w}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds  
 5265 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a  
 5266 sequence-terminating `GrB_wait()` is called. Regardless, the result vector,  $\mathbf{w}$ , is invalid from this  
 5267 point forward in the sequence.

5268 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

5269 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5270 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5271 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
 5272 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  
 5273 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5274 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5275 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5276 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5277 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5279 where the difference operator refers to set difference. We note that in this case of assigning  
 5280 a constant,  $\{\tilde{\mathbf{I}}[k], \forall k\}$  and  $\mathbf{ind}(\tilde{\mathbf{t}})$  are identical.

5281 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5282 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5283 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5284 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5285 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$5286 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5287 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5290 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5291 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 5292 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 5293 mask which acts as a “write mask”.

- 5294 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 5295 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5296 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5297 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 5298 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 5299 mask are unchanged:

$$5300 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5301 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 5302 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 5303 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 5304 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 5305 sequence.

#### 5306 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5307 Assign the same value to a specified subset of matrix elements. With the use of GrB\_ALL, the  
 5308 entire destination matrix can be filled with the constant.

### 5309 C Syntax

```
5310      GrB_Info GrB_assign(GrB_Matrix      C,
5311                        const GrB_Matrix  Mask,
5312                        const GrB_BinaryOp accum,
5313                        <type>            val,
5314                        const GrB_Index    *row_indices,
5315                        GrB_Index          nrows,
5316                        const GrB_Index    *col_indices,
5317                        GrB_Index          ncols,
5318                        const GrB_Descriptor desc);
```

```
5319      GrB_Info GrB_assign(GrB_Matrix      C,
5320                        const GrB_Matrix  Mask,
5321                        const GrB_BinaryOp accum,
5322                        const GrB_Scalar   s,
5323                        const GrB_Index    *row_indices,
5324                        GrB_Index          nrows,
```

```

5325         const GrB_Index      *col_indices,
5326         GrB_Index             ncols,
5327         const GrB_Descriptor   desc);

```

## 5328 Parameters

5329     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
5330     that may be accumulated with the result of the assign operation. On output, the  
5331     matrix holds the results of the operation.

5332     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
5333     stored into the output matrix **C**. The mask dimensions must match those of the  
5334     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5335     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
5336     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5337     dimensions of **C**), **GrB\_NULL** should be specified.

5338     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
5339     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5340     specified.

5341     **val** (IN) Scalar value to assign to (a subset of) **C**.

5342     **s** (IN) Scalar value to assign to (a subset of) **C**.

5343     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
5344     that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** − 1,  
5345     then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
5346     this array may be manipulated by the caller after this operation returns without  
5347     affecting any deferred computations for this operation. Unlike other variants, if  
5348     there are duplicated values in this array the result is still defined.

5349     **nrows** (IN) The number of values in **row\_indices** array. Must be in the range: [0, **nrows**(**C**)].  
5350     If **nrows** is zero, the operation becomes a NO-OP.

5351     **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**  
5352     that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** − 1,  
5353     then **GrB\_ALL** should be specified. Regardless of execution mode and return value,  
5354     this array may be manipulated by the caller after this operation returns without  
5355     affecting any deferred computations for this operation. Unlike other variants, if  
5356     there are duplicated values in this array the result is still defined.

5357     **ncols** (IN) The number of values in **col\_indices** array. Must be in the range: [0, **ncols**(**C**)].  
5358     If **ncols** is zero, the operation becomes a NO-OP.

5359     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5360     should be specified. Non-default field/value pairs are listed as follows:

5361

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`

5391 or  $C(\text{row\_indices}, \text{col\_indices}) = s$  is performed. If an optional binary accumulation operator  
 5392  $(\odot)$  is provided, then either  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot \text{val}$  or  
 5393  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot s$  is performed. More explicitly, if a  
 5394 non-opaque value  $\text{val}$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = \text{val}, \text{ or} \\ 5395 & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5396 Correspondingly, if a `GrB_Scalar`  $s$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = s, \text{ or} \\ 5397 & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5398 Logically, this operation occurs in three steps:

5399     Setup The internal vectors and mask used in the computation are formed and their domains  
 5400     and dimensions are tested for compatibility.

5401     Compute The indicated computations are carried out.

5402     Output The result is written into the output matrix, possibly under control of a mask.

5403 Up to two argument matrices are used in the `GrB_assign` operation:

- 5404     1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5405     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)

5406 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain  
 5407 compatibility as follows:

- 5408     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 5409     must be from one of the pre-defined types of Table 3.2.
- 5410     2.  $\mathbf{D}(C)$  must be compatible with either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5411     method.
- 5412     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5413     of the accumulation operator.
- 5414     4. If `accum` is not `GrB_NULL`, then either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5415     method, must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

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

From the arguments, the internal matrices, index arrays, and mask used in the computation are formed ( $\leftarrow$  denotes copy):

1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
  - (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - (b) If `Mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
3. Scalar  $\tilde{s} \leftarrow s$  (`GrB_Scalar` version only).
4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
  - (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
  - (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
  - (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

The internal matrix and mask are checked for dimension compatibility. The following conditions must hold:

1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
3.  $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$ .
4.  $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$ .

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

5448 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 5449 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5450 We are now ready to carry out the assign and any additional associated operations. We describe  
 5451 this in terms of two intermediate matrices:

- 5452 •  $\tilde{\mathbf{T}}$ : The matrix holding the copies of the scalar, either `val` or  $\tilde{s}$ , in their destination locations  
 5453 relative to  $\tilde{\mathbf{C}}$ .
- 5454 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5455 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows. If a non-opaque scalar `val` is provided:

$$5456 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5457 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e., `size`( $\tilde{s}$ ) = 1):

$$5458 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5459 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e., `size`( $\tilde{s}$ ) = 0):

$$5460 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5461 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  
 5462 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  
 5463  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds error listed above is  
 5464 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating  
 5465 `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from this point forward in the  
 5466 sequence.

5467 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 5468 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$5469 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5470 \quad \{(i, j, Z_{ij}) \mid (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}}))\} \rangle.$$

5471 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 5472 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  
 5473 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

5474 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 5475 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5476 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5477 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \\ 5478$$

5479 where the difference operator refers to set difference. We note that, in this particular case of  
 5480 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.



5481 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5482 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5483 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5484 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5485 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5486 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5487 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5488  
5489  
5490 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5491 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5492 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5493 mask which acts as a “write mask”.

5494 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5495 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5496 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5497 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
5498 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
5499 mask are unchanged:

$$5500 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5501 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5502 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5503 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
5504 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5505 sequence.

### 5506 4.3.8 apply: Apply a function to the elements of an object

5507 Computes the transformation of the values of the elements of a vector or a matrix using a unary  
5508 function, or a binary function where one argument is bound to a scalar.

#### 5509 4.3.8.1 apply: Vector variant

5510 Computes the transformation of the values of the elements of a vector using a unary function.

## 5511 C Syntax

```

5512      GrB_Info GrB_apply(GrB_Vector      w,
5513                        const GrB_Vector  mask,
5514                        const GrB_BinaryOp accum,
5515                        const GrB_UnaryOp  op,
5516                        const GrB_Vector  u,
5517                        const GrB_Descriptor desc);

```

## 5518 Parameters

5519     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5520     that may be accumulated with the result of the apply operation. On output, this  
5521     vector holds the results of the operation.

5522     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5523     stored into the output vector **w**. The mask dimensions must match those of the  
5524     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5525     of the mask vector must be of type **bool** or any of the predefined “built-in” types  
5526     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5527     dimensions of **w**), **GrB\_NULL** should be specified.

5528     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5529     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5530     specified.

5531     **op** (IN) A unary operator applied to each element of input vector **u**.

5532     **u** (IN) The GraphBLAS vector to which the unary function is applied.

5533     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5534     should be specified. Non-default field/value pairs are listed as follows:

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

## 5537 Return Values

5538     **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5539     blocking mode, this indicates that the compatibility tests on di-  
5540     mensions and domains for the input arguments passed successfully.

5541                                Either way, output vector  $w$  is ready to be used in the next method  
5542                                of the sequence.

5543                                **GrB\_PANIC** Unknown internal error.

5544                                **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
5545                                GraphBLAS objects (input or output) is in an invalid state caused  
5546                                by a previous execution error. Call **GrB\_error()** to access any error  
5547                                messages generated by the implementation.

5548                                **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5549 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
5550                                a call to **new** (or **dup** for vector parameters).

5551 **GrB\_DIMENSION\_MISMATCH**  $mask$ ,  $w$  and/or  $u$  dimensions are incompatible.

5552                                **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the corre-  
5553                                sponding domains of the accumulation operator or unary function,  
5554                                or the mask's domain is not compatible with **bool** (in the case where  
5555                                 $desc[GrB\_MASK].GrB\_STRUCTURE$  is not set).

## 5556 **Description**

5557 This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a  
5558 GraphBLAS vector:  $w = f(u)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
5559  $w = w \odot f(u)$ .

5560 Logically, this operation occurs in three steps:

5561                                **Setup** The internal vectors and mask used in the computation are formed and their domains  
5562                                and dimensions are tested for compatibility.

5563 **Compute** The indicated computations are carried out.

5564 **Output** The result is written into the output vector, possibly under control of a mask.

5565 Up to three argument vectors are used in this **GrB\_apply** operation:

- 5566        1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5567        2.  $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$  (optional)
- 5568        3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5569 The argument vectors, unary operator and the accumulation operator (if provided) are tested for  
5570 domain compatibility as follows:

- 5571 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
5572 must be from one of the pre-defined types of Table 3.2.
- 5573 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5574 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5575 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5576  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5577 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in}(\text{op})$ .

5578 Two domains are compatible with each other if values from one domain can be cast to values in  
5579 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5580 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5581 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
5582 error listed above is returned.

5583 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
5584 denotes copy):

- 5585 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5586 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 5587 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 5588 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 5589 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 5590 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 5591 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5592 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5593 The internal vectors and masks are checked for dimension compatibility. The following conditions  
5594 must hold:

- 5595 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5596 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

5597 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5598 error listed above is returned.

5599 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5600 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5601 We are now ready to carry out the apply and any additional associated operations. We describe  
5602 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 <b>Mask</b> matrix. The stored values are not examined.
Mask	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <b>Mask</b> .
A	<code>GrB_INP0</code>	<code>GrB_TRAN</code>	Use transpose of A for the operation.

## 5663 Return Values

5664	<b>GrB_SUCCESS</b>	In blocking mode, the operation completed successfully. In non-
5665		blocking mode, this indicates that the compatibility tests on
5666		dimensions and domains for the input arguments passed suc-
5667		cessfully. Either way, output matrix C is ready to be used in the
5668		next method of the sequence.
5669	<b>GrB_PANIC</b>	Unknown internal error.
5670	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
5671		opaque GraphBLAS objects (input or output) is in an invalid
5672		state caused by a previous execution error. Call <b>GrB_error()</b> to
5673		access any error messages generated by the implementation.
5674	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
5675	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
5676		by a call to <b>new</b> (or <b>Matrix_dup</b> for matrix parameters).
5677	<b>GrB_DIMENSION_MISMATCH</b>	Mask and C dimensions are incompatible, <b>nrows</b> $\neq$ <b>nrows</b> (C), or
5678		<b>ncols</b> $\neq$ <b>ncols</b> (C).
5679	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various matrices are incompatible with the
5680		corresponding domains of the accumulation operator or unary
5681		function, or the mask's domain is not compatible with <b>bool</b> (in
5682		the case where <b>desc</b> [ <b>GrB_MASK</b> ]. <b>GrB_STRUCTURE</b> is not set).

## 5683 Description

5684 This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a  
 5685 GraphBLAS matrix:  $C = f(A)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
 5686  $C = C \odot f(A)$ .

5687 Logically, this operation occurs in three steps:

5688     **Setup** The internal matrices and mask used in the computation are formed and their domains  
 5689     and dimensions are tested for compatibility.

5690     **Compute** The indicated computations are carried out.

5691     **Output** The result is written into the output matrix, possibly under control of a mask.

5692 Up to three argument matrices are used in the **GrB\_apply** operation:

- 5693 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5694 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)

5695 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5696 The argument matrices, unary operator and the accumulation operator (if provided) are tested for  
5697 domain compatibility as follows:

- 5698 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
5699 must be from one of the pre-defined types of Table 3.2.
- 5700 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5701 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5702 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5703  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5704 4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in}(\text{op})$  of the unary operator.

5705 Two domains are compatible with each other if values from one domain can be cast to values in  
5706 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5707 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5708 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
5709 error listed above is returned.

5710 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
5711 are formed ( $\leftarrow$  denotes copy):

- 5712 1. Matrix  $\tilde{C} \leftarrow C$ .
- 5713 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument **Mask** as follows:
  - 5714 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
5715  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 5716 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 5717 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
5718  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 5719 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
5720  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 5721 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .
- 5722 3. Matrix  $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$ .

5723 The internal matrices and mask are checked for dimension compatibility. The following conditions  
5724 must hold:

- 5725 1.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$ .
- 5726 2.  $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$ .
- 5727 3.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$ .



5728 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

5729 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5730 error listed above is returned.

5731 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5732 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5733 We are now ready to carry out the apply and any additional associated operations. We describe  
5734 this in terms of two intermediate matrices:

- 5735 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the unary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 5736 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5737 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$5738 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5739 where  $f = \mathbf{f}(\text{op})$ .

5740 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 5741 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 5742 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5743 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5744 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5745 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5746 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5747 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5748 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5751 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5752 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5753 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5754 mask which acts as a “write mask”.

- 5755 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5756 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5757 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the mask are unchanged:

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

In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

#### 4.3.8.3 apply: Vector-BinaryOp variants[Scott: NEW CONTENT]

Computes the transformation of the values of the stored elements of a vector using a binary operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the first argument to the binary operator and stored elements of the vector are passed as the second argument. In the *bind-second* variant, the elements of the vector are passed as the first argument and the specified scalar value is passed as the second argument. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### C Syntax

```
// bind-first + scalar value
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
                   const GrB_BinaryOp   accum,
                   const GrB_BinaryOp   op,
                   <type>               val,
                   const GrB_Vector     u,
                   const GrB_Descriptor desc);
```

```
// bind-first + GraphBLAS scalar
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
                   const GrB_BinaryOp   accum,
                   const GrB_BinaryOp   op,
                   const GrB_Scalar     s,
                   const GrB_Vector     u,
                   const GrB_Descriptor desc);
```

```
// bind-second + scalar value
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
```

```

5794         const GrB_BinaryOp      accum,
5795         const GrB_BinaryOp      op,
5796         const GrB_Vector        u,
5797         <type>                  val,
5798         const GrB_Descriptor    desc);

5799 // bind-second + GraphBLAS scalar
5800 GrB_Info GrB_apply(GrB_Vector      w,
5801                   const GrB_Vector mask,
5802                   const GrB_BinaryOp accum,
5803                   const GrB_BinaryOp op,
5804                   const GrB_Vector u,
5805                   const GrB_Scalar s,
5806                   const GrB_Descriptor desc);

```

## 5807 Parameters

5808     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5809     that may be accumulated with the result of the apply operation. On output, this  
5810     vector holds the results of the operation.

5811     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5812     stored into the output vector **w**. The mask dimensions must match those of the  
5813     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5814     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5815     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5816     dimensions of **w**), **GrB\_NULL** should be specified.

5817     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5818     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5819     specified.

5820     **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar  
5821     value, **val**.

5822     **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as  
5823     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
5824     argument in the *bind-second* variant.

5825     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
5826     argument in the *bind-first* variant, or the right-hand (second) argument in the  
5827     *bind-second* variant.

5828     **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand  
5829     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
5830     the *bind-second* variant. It must not be empty.

5831 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
5832 should be specified. Non-default field/value pairs are listed as follows:

5833

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <b>w</b> 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 <b>mask</b> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <b>mask</b> .

5834

## 5835 Return Values

5836 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
5837 blocking mode, this indicates that the compatibility tests on di-  
5838 mensions and domains for the input arguments passed successfully.  
5839 Either way, output vector **w** is ready to be used in the next method  
5840 of the sequence.

5841 GrB\_PANIC Unknown internal error.

5842 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
5843 GraphBLAS objects (input or output) is in an invalid state caused  
5844 by a previous execution error. Call GrB\_error() to access any error  
5845 messages generated by the implementation.

5846 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

5847 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
5848 a call to new (or dup for vector parameters).

5849 GrB\_DIMENSION\_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

5850 GrB\_DOMAIN\_MISMATCH The domains of the various vectors and scalar are incompatible with  
5851 the corresponding domains of the binary operator or accumulation  
5852 operator, or the **mask**'s domain is not compatible with **bool** (in the  
5853 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

5854 GrB\_EMPTY\_OBJECT The GrB\_Scalar **s** used in the call is empty (**nvals(s) = 0**) and  
5855 therefore a value cannot be passed to the binary operator.

## 5856 Description

5857 This variant of GrB\_apply computes the result of applying a binary operator to the elements of a  
5858 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5859                   bind-first:      $w = f(\text{val}, u)$  or  $w = f(s, u)$

5860                   bind-second:     $w = f(u, \text{val})$  or  $w = f(u, s)$ ,

5861 or if an optional binary accumulation operator ( $\odot$ ) is provided:

5862                   bind-first:      $w = w \odot f(\text{val}, u)$  or  $w = w \odot f(s, u)$

5863                   bind-second:     $w = w \odot f(u, \text{val})$  or  $w = w \odot f(u, s)$ .

5864 Logically, this operation occurs in three steps:

5865     **Setup** The internal vectors and mask used in the computation are formed and their domains  
5866             and dimensions are tested for compatibility.

5867     **Compute** The indicated computations are carried out.

5868     **Output** The result is written into the output vector, possibly under control of a mask.

5869 Up to three argument vectors are used in this GrB\_apply operation:

5870     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

5871     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)

5872     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5873 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are  
5874 tested for domain compatibility as follows:

5875     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
5876         must be from one of the pre-defined types of Table 3.2.

5877     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.

5878     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5879         of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
5880          $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

5881     4.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

5882     5. If bind-first:

5883         (a)  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

5884         (b) If the non-opaque scalar **val** is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
5885             of the binary operator.

5886         (c) If the GrB\_Scalar **s** is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
5887             binary operator.

- 5888 6. If bind-second:
- 5889 (a)  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the binary operator.
- 5890 (b) If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$
- 5891 of the binary operator.
- 5892 (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
- 5893 binary operator.

5894 Two domains are compatible with each other if values from one domain can be cast to values in

5895 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5896 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5897 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5898 error listed above is returned.

5899 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$

5900 denotes copy):

- 5901 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5902 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 5903 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 5904 (b) If `mask  $\neq$  GrB_NULL`,
- 5905 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 5906 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 5907 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5908 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 5909 4. Scalar  $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$  (GraphBLAS scalar case).

5910 The internal vectors and masks are checked for dimension compatibility. The following conditions

5911 must hold:

- 5912 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5913 2.  $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$ .

5914 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5915 error listed above is returned.

5916 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5917 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5918 If an empty `GrB_Scalar`  $\tilde{\mathbf{s}}$  is provided ( $\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.

5919 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

5920 `val` with the same domain as  $\tilde{\mathbf{s}}$  and set `val = val( $\tilde{\mathbf{s}}$ )`.

5921 We are now ready to carry out the apply and any additional associated operations. We describe

5922 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.

#### 5956 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

5957 Computes the transformation of the values of the stored elements of a matrix using a binary  
5958 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the  
5959 first argument to the binary operator and stored elements of the matrix are passed as the second  
5960 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument  
5961 and the specified scalar value is passed as the second argument. The scalar can be passed either as  
5962 a non-opaque variable or as a GrB\_Scalar object.

#### 5963 C Syntax

```
5964 // bind-first + scalar value
5965 GrB_Info GrB_apply(GrB_Matrix      C,
5966                   const GrB_Matrix Mask,
5967                   const GrB_BinaryOp accum,
5968                   const GrB_BinaryOp op,
5969                   <type>           val,
5970                   const GrB_Matrix A,
5971                   const GrB_Descriptor desc);
```

```
5972 // bind-first + GraphBLAS scalar
5973 GrB_Info GrB_apply(GrB_Matrix      C,
5974                   const GrB_Matrix Mask,
5975                   const GrB_BinaryOp accum,
5976                   const GrB_BinaryOp op,
5977                   const GrB_Scalar s,
5978                   const GrB_Matrix A,
5979                   const GrB_Descriptor desc);
```

```
5980 // bind-second + scalar value
5981 GrB_Info GrB_apply(GrB_Matrix      C,
5982                   const GrB_Matrix Mask,
5983                   const GrB_BinaryOp accum,
5984                   const GrB_BinaryOp op,
5985                   const GrB_Matrix A,
5986                   <type>           val,
5987                   const GrB_Descriptor desc);
```

```
5988 // bind-second + GraphBLAS scalar
5989 GrB_Info GrB_apply(GrB_Matrix      C,
5990                   const GrB_Matrix Mask,
5991                   const GrB_BinaryOp accum,
5992                   const GrB_BinaryOp op,
5993                   const GrB_Matrix A,
```



```

5994         const GrB_Scalar      s,
5995         const GrB_Descriptor desc);

```

## 5996 Parameters

5997     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
5998     that may be accumulated with the result of the apply operation. On output, the  
5999     matrix holds the results of the operation.

6000     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6001     stored into the output matrix C. The mask dimensions must match those of the  
6002     matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
6003     of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
6004     in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
6005     dimensions of C), `GrB_NULL` should be specified.

6006     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
6007     entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
6008     specified.

6009     **op** (IN) A binary operator applied to each element of input matrix, A, with the element  
6010     of the input matrix used as the left-hand argument, and the scalar value, `val`, used  
6011     as the right-hand argument.

6012     **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as  
6013     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
6014     argument in the *bind-second* variant.

6015     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
6016     argument in the *bind-first* variant, or the right-hand (second) argument in the  
6017     *bind-second* variant.

6018     **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand  
6019     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
6020     the *bind-second* variant. It must not be empty.

6021     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6022     should be specified. Non-default field/value pairs are listed as follows:  
6023

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.

## 6051 Description

6052 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a  
 6053 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6054                   bind-first:      $C = f(\text{val}, A)$  or  $C = f(s, A)$

6055                   bind-second:     $C = f(A, \text{val})$  or  $C = f(A, s)$ ,

6056 or if an optional binary accumulation operator ( $\odot$ ) is provided:

6057                   bind-first:      $C = C \odot f(\text{val}, A)$  or  $C = C \odot f(s, A)$

6058                   bind-second:     $C = C \odot f(A, \text{val})$  or  $C = C \odot f(A, s)$ .

6059 Logically, this operation occurs in three steps:

6060       **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6061               and dimensions are tested for compatibility.

6062       **Compute** The indicated computations are carried out.

6063       **Output** The result is written into the output matrix, possibly under control of a mask.

6064 Up to three argument matrices are used in the `GrB_apply` operation:

- 6065     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6066     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)
- 6067     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6068 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are  
 6069 tested for domain compatibility as follows:

- 6070     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 6071       must be from one of the pre-defined types of Table 3.2.
- 6072     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.
- 6073     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 6074       of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
 6075        $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6076     4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.
- 6077     5. If bind-first:  
 6078       (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

6079 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
 6080 of the binary operator.

6081 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
 6082 binary operator.

6083 6. If `bind-second`:

6084 (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

6085 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
 6086 of the binary operator.

6087 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the  
 6088 binary operator.

6089 Two domains are compatible with each other if values from one domain can be cast to values in  
 6090 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6091 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6092 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
 6093 error listed above is returned.

6094 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
 6095 are formed ( $\leftarrow$  denotes copy):

6096 1. Matrix  $\tilde{C} \leftarrow C$ .

6097 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:

6098 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
 6099  $j < \mathbf{ncols}(C)\} \rangle$ .

6100 (b) If `Mask`  $\neq$  `GrB_NULL`,

6101 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
 6102  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,

6103 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
 6104  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .

6105 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .

6106 3. Matrix  $\tilde{A}$  is computed from argument `A` as follows:

6107 `bind-first`:  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? A^T : A$

6108 `bind-second`:  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$

6109 4. Scalar  $\tilde{s} \leftarrow s$  (`GraphBLAS` scalar case).

6110 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 6111 must hold:

6112 1.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$ .

6113 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .

6114 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6115 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6116 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6117 error listed above is returned.

6118 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6119 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6120 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6121 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6122  $\mathbf{val}$  with the same domain as  $\tilde{s}$  and set  $\mathbf{val} = \mathbf{val}(\tilde{s})$ .

6123 We are now ready to carry out the apply and any additional associated operations. We describe  
6124 this in terms of two intermediate matrices:

- 6125 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the binary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 6126 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6127 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as one of the following:

6128 bind-first:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6129 bind-second:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6130 where  $f = \mathbf{f}(\mathbf{op})$ .

6131 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6132 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6133 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6134 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6135 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6136 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6137 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6138 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6139 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6140 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6143 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 6144 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 6145 mask which acts as a “write mask”.

- 6146 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 6147 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6148 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6149 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 6150 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 6151 mask are unchanged:

$$6152 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6153 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 6154 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 6155 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 6156 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 6157 sequence.

#### 6158 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6159 Computes the transformation of the values of the stored elements of a vector using an index unary  
 6160 operator that is a function of the stored value, its location indices, and an user provided scalar  
 6161 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6162 C Syntax

```
6163      GrB_Info GrB_apply(GrB_Vector      w,
6164                        const GrB_Vector  mask,
6165                        const GrB_BinaryOp accum,
6166                        const GrB_IndexUnaryOp op,
6167                        const GrB_Vector  u,
6168                        <type>            val,
6169                        const GrB_Descriptor desc);
```

```
6170      GrB_Info GrB_apply(GrB_Vector      w,
6171                        const GrB_Vector  mask,
6172                        const GrB_BinaryOp accum,
6173                        const GrB_IndexUnaryOp op,
6174                        const GrB_Vector  u,
6175                        const GrB_Scalar  s,
6176                        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
<b>w</b>	<b>GrB_OUTP</b>	<b>GrB_REPLACE</b>	Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.
<b>mask</b>	<b>GrB_MASK</b>	<b>GrB_STRUCTURE</b>	The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined.
<b>mask</b>	<b>GrB_MASK</b>	<b>GrB_COMP</b>	Use the complement of <b>mask</b> .

## 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.

6208                   GrB\_PANIC   Unknown internal error.

6209           GrB\_INVALID\_OBJECT   This is returned in any execution mode whenever one of the  
6210                                   opaque GraphBLAS objects (input or output) is in an invalid  
6211                                   state caused by a previous execution error. Call GrB\_error() to  
6212                                   access any error messages generated by the implementation.

6213           GrB\_OUT\_OF\_MEMORY   Not enough memory available for operation.

6214   GrB\_UNINITIALIZED\_OBJECT   One or more of the GraphBLAS objects has not been initialized  
6215                                   by a call to new (or another constructor).

6216   GrB\_DIMENSION\_MISMATCH   mask, w and/or u dimensions are incompatible.

6217   GrB\_DOMAIN\_MISMATCH   The domains of the various vectors are incompatible with the cor-  
6218                                   responding domains of the accumulation operator or index unary  
6219                                   operator, or the mask's domain is not compatible with bool (in  
6220                                   the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

6221           GrB\_EMPTY\_OBJECT   The GrB\_Scalar s used in the call is empty ( $\mathbf{nvals}(s) = 0$ ) and  
6222                                   therefore a value cannot be passed to the index unary operator.

## 6223   Description

6224   This variant of GrB\_apply computes the result of applying an index unary operator to the elements  
6225   of a GraphBLAS vector each composed with the element's index and a scalar constant, val or s:

$$6226 \quad \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \text{val}) \text{ or } \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}),$$

6227   or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6228 \quad \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \text{val}) \text{ or } \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}).$$

6229   Logically, this operation occurs in three steps:

6230       **Setup**   The internal vectors and mask used in the computation are formed and their domains  
6231                   and dimensions are tested for compatibility.

6232       **Compute**   The indicated computations are carried out.

6233       **Output**   The result is written into the output vector, possibly under control of a mask.

6234   Up to three argument vectors are used in this GrB\_apply operation:

- 6235   1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6236   2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)



6237 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6238 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6239 are tested for domain compatibility as follows:

- 6240 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
6241 must be from one of the pre-defined types of Table 3.2.
- 6242 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the index unary operator.
- 6243 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6244 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be compatible  
6245 with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6246 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6247 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  
6248 the index unary operator.
- 6249 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  
6250 unary operator.

6251 Two domains are compatible with each other if values from one domain can be cast to values in  
6252 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6253 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6254 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
6255 error listed above is returned.

6256 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6257 denotes copy):

- 6258 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6259 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 6260 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 6261 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 6262 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
    - 6263 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 6264 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6265 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6266 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6267 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6268 must hold:

6269 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6270 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

6271 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6272 error listed above is returned.

6273 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6274 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6275 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6276 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided ( $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable `val`  
6277 with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6278 We are now ready to carry out the apply and any additional associated operations. We describe  
6279 this in terms of two intermediate vectors:

- 6280 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6281  $\tilde{\mathbf{u}}$ .
- 6282 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6283 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6284 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6285 where  $f_i = \mathbf{f}(\text{op})$ .

6286 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6287 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 6288 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6289 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6290 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6291 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6292 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6293 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6294 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

6297 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6298 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6299 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6300 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.

#### 6313 4.3.8.6 apply: Matrix index unary operator variant[Scott: NEW CONTENT]

6314 Computes the transformation of the values of the stored elements of a matrix using an index unary  
6315 operator that is a function of the stored value, its location indices, and an user provided scalar  
6316 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6317 C Syntax

```
6318     GrB_Info GrB_apply(GrB_Matrix      C,
6319                       const GrB_Matrix Mask,
6320                       const GrB_BinaryOp accum,
6321                       const GrB_IndexUnaryOp op,
6322                       const GrB_Matrix A,
6323                       <type>          val,
6324                       const GrB_Descriptor desc);
```

```
6325     GrB_Info GrB_apply(GrB_Matrix      C,
6326                       const GrB_Matrix Mask,
6327                       const GrB_BinaryOp accum,
6328                       const GrB_IndexUnaryOp op,
6329                       const GrB_Matrix A,
6330                       const GrB_Scalar s,
6331                       const GrB_Descriptor desc);
```

#### 6332 Parameters

6333 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6334 that may be accumulated with the result of the apply operation. On output, the  
6335 matrix holds the results of the operation.

6336 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6337 stored into the output matrix **C**. The mask dimensions must match those of the  
6338 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6339 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6340 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6341 dimensions of **C**), **GrB\_NULL** should be specified.

6342 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6343 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6344 specified.

6345 **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  
6346 to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6347 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6348 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6349 ator.

6350 **val** (IN) An additional scalar value that is passed to the index unary operator.

6351 **s** (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6352 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6353 should be specified. Non-default field/value pairs are listed as follows:  
6354

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

## 6356 Return Values

6357 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
6358 blocking mode, this indicates that the compatibility tests on di-  
6359 mensions and domains for the input arguments passed successfully.  
6360 Either way, output matrix **C** is ready to be used in the next method  
6361 of the sequence.

6362 **GrB\_PANIC** Unknown internal error.

6363 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
6364 GraphBLAS objects (input or output) is in an invalid state caused

6365 by a previous execution error. Call `GrB_error()` to access any error  
 6366 messages generated by the implementation.

6367 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6368 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 6369 a call to `new` (or another constructor).

6370 **GrB\_DIMENSION\_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6371 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the  
 6372 corresponding domains of the accumulation operator or index unary  
 6373 operator, or the mask's domain is not compatible with `bool` (in the  
 6374 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6375 **GrB\_EMPTY\_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
 6376 therefore a value cannot be passed to the index unary operator.

## 6377 Description

6378 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements  
 6379 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar  
 6380 constant, `val` or `s`:

$$6381 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6382 or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6383 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6384 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
 6385 indices, respectively.

6386 Logically, this operation occurs in three steps:

6387 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6388 and dimensions are tested for compatibility.

6389 **Compute** The indicated computations are carried out.

6390 **Output** The result is written into the output matrix, possibly under control of a mask.

6391 Up to three argument matrices are used in the `GrB_apply` operation:

- 6392 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6393 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)

6394 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6395 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6396 are tested for domain compatibility as follows:

- 6397 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{Mask})$   
6398 must be from one of the pre-defined types of Table 3.2.
- 6399 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator.
- 6400 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6401 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be compatible  
6402 with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 6403 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6404 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  
6405 the index unary operator.
- 6406 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  
6407 unary operator.

6408 Two domains are compatible with each other if values from one domain can be cast to values in  
6409 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6410 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6411 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
6412 error listed above is returned.

6413 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation  
6414 are formed ( $\leftarrow$  denotes copy):

- 6415 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6416 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument **Mask** as follows:
  - 6417 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
6418  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 6419 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 6420 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
6421  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 6422 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
6423  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 6424 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6425 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument **A** as follows:
  - 6426  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6427 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6428 The internal matrices and mask are checked for dimension compatibility. The following conditions  
6429 must hold:

- 6430 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6431 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6432 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6433 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6434 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6435 error listed above is returned.

6436 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6437 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6438 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6439 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6440 `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6441 We are now ready to carry out the apply and any additional associated operations. We describe  
6442 this in terms of two intermediate matrices:

- 6443 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6444  $\tilde{\mathbf{A}}$ .
- 6445 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6446 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6447 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6448 where  $f_i = \mathbf{f}(\mathbf{op})$ .

6449 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6450 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6451 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6452 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6453 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6454 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 6455 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6456 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6457 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6458 \end{aligned}$$

6459 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6461 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 6462 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 6463 mask which acts as a “write mask”.

- 6464 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 6465 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6466 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6467 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 6468 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 6469 mask are unchanged:

$$6470 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6471 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 6472 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 6473 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 6474 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 6475 sequence.

#### 6476 4.3.9 select:

6477 Apply a select operator to the stored elements of an object to determine whether or not to keep  
 6478 them.

##### 6479 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6480 Apply a select operator (an index unary operator) to the elements of a vector.

#### 6481 C Syntax

```
6482 // scalar value variant
6483 GrB_Info GrB_select(GrB_Vector          w,
6484                    const GrB_Vector     mask,
6485                    const GrB_BinaryOp   accum,
6486                    const GrB_IndexUnaryOp op,
6487                    const GrB_Vector     u,
6488                    <type>               val,
6489                    const GrB_Descriptor desc);
6490
6491 // GraphBLAS scalar variant
6492 GrB_Info GrB_select(GrB_Vector          w,
6493                    const GrB_Vector     mask,
```



```

6494         const GrB_BinaryOp      accum,
6495         const GrB_IndexUnaryOp  op,
6496         const GrB_Vector        u,
6497         const GrB_Scalar        s,
6498         const GrB_Descriptor    desc);
6499

```

## 6500 Parameters

6501     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6502     that may be accumulated with the result of the select operation. On output, this  
6503     vector holds the results of the operation.

6504     **mask** (IN) An optional “write” mask that controls which results from this operation are  
6505     stored into the output vector **w**. The mask dimensions must match those of the  
6506     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6507     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6508     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6509     dimensions of **w**), **GrB\_NULL** should be specified.

6510     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6511     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6512     specified.

6513     **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  
6514     to each element stored in the input vector, **u**. It is a function of the stored element’s  
6515     value, its location index, and a user supplied scalar value (either **s** or **val**).

6516     **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-  
6517     ator.

6518     **val** (IN) An additional scalar value that is passed to the index unary operator.

6519     **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6520     not be empty.

6521     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6522     should be specified. Non-default field/value pairs are listed as follows:

6523

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

6524

## 6525 Return Values

6526           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 6527                           blocking mode, this indicates that the compatibility tests on di-  
 6528                           mensions and domains for the input arguments passed success-  
 6529                           fully. Either way, output vector **w** is ready to be used in the next  
 6530                           method of the sequence.

6531           **GrB\_PANIC** Unknown internal error.

6532           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 6533                           opaque GraphBLAS objects (input or output) is in an invalid  
 6534                           state caused by a previous execution error. Call **GrB\_error()** to  
 6535                           access any error messages generated by the implementation.

6536           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6537           **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
 6538                           by a call to one of its constructors.

6539           **GrB\_DIMENSION\_MISMATCH** **mask**, **w** and/or **u** dimensions are incompatible.

6540           **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the cor-  
 6541                           responding domains of the accumulation operator or index unary  
 6542                           operator, or the **mask**'s domain is not compatible with **bool** (in  
 6543                           the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

6544           **GrB\_EMPTY\_OBJECT** The **GrB\_Scalar s** used in the call is empty (**nvals(s) = 0**) and  
 6545                           therefore a value cannot be passed to the index unary operator.

## 6546 Description

6547 This variant of **GrB\_select** computes the result of applying a index unary operator to select the  
 6548 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored  
 6549 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding  
 6550 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.  
 6551 This acts like a functional mask on the input vector as follows:

$$6552 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6553 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6554 Correspondingly, if a **GrB\_Scalar s**, is provided:

$$6555 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6556 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6557 Logically, this operation occurs in three steps:

6558     **Setup** The internal vectors and mask used in the computation are formed and their domains  
6559             and dimensions are tested for compatibility.

6560     **Compute** The indicated computations are carried out.

6561     **Output** The result is written into the output vector, possibly under control of a mask.

6562 Up to three argument vectors are used in this `GrB_select` operation:

- 6563     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$   
6564     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)  
6565     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6566 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6567 are tested for domain compatibility as follows:

- 6568     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
6569         must be from one of the pre-defined types of Table 3.2.  
6570     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .  
6571     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6572         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
6573         mulation operator.  
6574     4.  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6575         i.e., castable to `bool`.  
6576     5.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.  
6577     6.  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(\mathbf{s})$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$   
6578         of the index unary operator.

6579 Two domains are compatible with each other if values from one domain can be cast to values in  
6580 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6581 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6582 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch  
6583 error listed above is returned.

6584 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6585 denotes copy):

- 6586     1. Vector  $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$ .  
6587     2. One-dimensional mask,  $\widetilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

- 6588 (a) If  $\text{mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
- 6589 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,
- 6590 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
- 6591 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
- 6592 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 6593 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6594 4. Scalar  $\widetilde{s} \leftarrow s$  (GrB\_Scalar version only).

6595 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6596 must hold:

- 6597 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6598 2.  $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$ .

6599 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch  
6600 error listed above is returned.

6601 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6602 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6603 If an empty `GrB_Scalar`  $\widetilde{s}$  is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6604 If a non-empty `GrB_Scalar`,  $\widetilde{s}$ , is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 1$ ), we then create an internal variable  
6605 `val` with the same domain as  $\widetilde{s}$  and set  $\text{val} = \text{val}(\widetilde{s})$ .

6606 We are now ready to carry out the `select` and any additional associated operations. We describe  
6607 this in terms of two intermediate vectors:

- 6608 •  $\widetilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6609  $\widetilde{\mathbf{u}}$ .
- 6610 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6611 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$6612 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6613 where  $f_i = \mathbf{f}(\text{op})$ .

6614 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6615 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 6616 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$6617 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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);
```

```

6653 // GraphBLAS scalar variant
6654 GrB_Info GrB_select(GrB_Matrix          C,
6655                    const GrB_Matrix     Mask,
6656                    const GrB_BinaryOp    accum,
6657                    const GrB_IndexUnaryOp op,
6658                    const GrB_Matrix      A,
6659                    const GrB_Scalar      s,
6660                    const GrB_Descriptor   desc);

```

## 6661 Parameters

6662     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6663     that may be accumulated with the result of the select operation. On output, the  
6664     matrix holds the results of the operation.

6665     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6666     stored into the output matrix **C**. The mask dimensions must match those of the  
6667     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6668     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6669     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6670     dimensions of **C**), **GrB\_NULL** should be specified.

6671     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6672     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6673     specified.

6674     **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  
6675     to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6676     value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6677     **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6678     ator.

6679     **val** (IN) An additional scalar value that is passed to the index unary operator.

6680     **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6681     not be empty.

6682     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6683     should be specified. Non-default field/value pairs are listed as follows:  
6684

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:

6714  $C = A \langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \rangle$ , or  
6715  $C = C \odot A \langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \rangle$ .

6716 Correspondingly, if a GrB\_Scalar,  $s$ , is provided:

6717  $C = A \langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s) \rangle$ , or  
6718  $C = C \odot A \langle f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s) \rangle$ .

6719 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
6720 indices, respectively.

6721 Logically, this operation occurs in three steps:

6722     **Setup** The internal matrices and mask used in the computation are formed and their domains  
6723             and dimensions are tested for compatibility.

6724     **Compute** The indicated computations are carried out.

6725     **Output** The result is written into the output matrix, possibly under control of a mask.

6726 Up to three argument matrices are used in the GrB\_select operation:

- 6727 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6728 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)
- 6729 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6730 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6731 are tested for domain compatibility as follows:

- 6732 1. If **Mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
6733     must be from one of the pre-defined types of Table 3.2.
- 6734 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 6735 3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6736     of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
6737     mulation operator.
- 6738 4.  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6739     i.e., castable to **bool**.
- 6740 5.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6741 6.  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
6742     of the index unary operator.



6743 Two domains are compatible with each other if values from one domain can be cast to values in  
 6744 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6745 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6746 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch  
 6747 error listed above is returned.

6748 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
 6749 are formed ( $\leftarrow$  denotes copy):

- 6750 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6751 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 6752 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 6753  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 6754 (b) If `Mask  $\neq$  GrB_NULL`,
    - 6755 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 6756  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 6757 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 6758  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 6759 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6760 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument `A` as follows:  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6761 4. Scalar  $\tilde{s} \leftarrow s$  (`GrB_Scalar` version only).

6762 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 6763 must hold:

- 6764 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6765 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6766 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6767 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6768 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch  
 6769 error listed above is returned.

6770 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 6771 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6772 If an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
 6773 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
 6774 `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6775 We are now ready to carry out the `select` and any additional associated operations. We describe  
 6776 this in terms of two intermediate matrices:

- 6777 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6778  $\tilde{\mathbf{A}}$ .
- 6779 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6780 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6781 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6782 where  $f_i = \mathbf{f}(\text{op})$ .

6783 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6784 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6785 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6786 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6787 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6788 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6789 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6790 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6791 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6792 \quad 6793$$

6794 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6795 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6796 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6797 mask which acts as a “write mask”.

- 6798 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
6799 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6800 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6801 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6802 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6803 mask are unchanged:

$$6804 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6805 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
6806 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
6807 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
6808 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6809 sequence.

### 6810 4.3.10 reduce: Perform a reduction across the elements of an object

6811 Computes the reduction of the values of the elements of a vector or matrix.

#### 6812 4.3.10.1 reduce: Standard matrix to vector variant

6813 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns  
6814 is desired, the input matrix should be transposed using the descriptor.

### 6815 C Syntax

```
6816         GrB_Info GrB_reduce(GrB_Vector          w,  
6817                             const GrB_Vector    mask,  
6818                             const GrB_BinaryOp   accum,  
6819                             const GrB_Monoid     op,  
6820                             const GrB_Matrix     A,  
6821                             const GrB_Descriptor desc);  
6822  
6823         GrB_Info GrB_reduce(GrB_Vector          w,  
6824                             const GrB_Vector    mask,  
6825                             const GrB_BinaryOp   accum,  
6826                             const GrB_BinaryOp   op,  
6827                             const GrB_Matrix     A,  
6828                             const GrB_Descriptor desc);
```

### 6829 Parameters

6830 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6831 that may be accumulated with the result of the reduction operation. On output,  
6832 this vector holds the results of the operation.

6833 **mask** (IN) An optional “write” mask that controls which results from this operation are  
6834 stored into the output vector **w**. The mask dimensions must match those of the  
6835 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6836 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6837 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6838 dimensions of **w**), **GrB\_NULL** should be specified.

6839 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6840 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6841 specified.

6842 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.  
6843 Depending on which type is passed, the following defines the binary operator with  
6844 one domain,  $F_b = \langle D, D, D, \oplus \rangle$ , that is used:

6845 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
6846 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  
6847 monoid is ignored.

6848 If `op` is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in  
6849 both cases  $\odot(\text{op})$  must be commutative and associative. Otherwise, the outcome  
6850 of the operation is undefined.

6851 `A` (IN) The GraphBLAS matrix on which reduction will be performed.

6852 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6853 should be specified. Non-default field/value pairs are listed as follows:  
6854

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.

## 6856 Return Values

6857 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
6858 blocking mode, this indicates that the compatibility tests on di-  
6859 mensions and domains for the input arguments passed successfully.  
6860 Either way, output vector `w` is ready to be used in the next method  
6861 of the sequence.

6862 `GrB_PANIC` Unknown internal error.

6863 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
6864 GraphBLAS objects (input or output) is in an invalid state caused  
6865 by a previous execution error. Call `GrB_error()` to access any error  
6866 messages generated by the implementation.

6867 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

6868 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by  
6869 a call to `new` (or `dup` for vector parameters).

6870 `GrB_DIMENSION_MISMATCH` `mask`, `w` and/or `u` dimensions are incompatible.

6871 `GrB_DOMAIN_MISMATCH` Either the domains of the various vectors and matrices are incom-  
6872 patible with the corresponding domains of the accumulation oper-  
6873 ator or reduce function, or the domains of the GraphBLAS binary

6874 operator `op` are not all the same, or the mask's domain is not com-  
6875 patible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE`  
6876 is not set).

## 6877 Description

6878 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows  
6879 of an input matrix:  $w(i) = \bigoplus A(i, :) \forall i$ ; or, if an optional binary accumulation operator is provided,  
6880  $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$ , where  $\bigoplus = \odot(F_b)$  and  $\odot = \odot(\text{accum})$ .

6881 Logically, this operation occurs in three steps:

6882     **Setup** The internal vector, matrix and mask used in the computation are formed and their  
6883 domains and dimensions are tested for compatibility.

6884 **Compute** The indicated computations are carried out.

6885 **Output** The result is written into the output vector, possibly under control of a mask.

6886 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6887 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6888 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 6889 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6890 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested  
6891 for domain compatibility as follows:

- 6892 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
6893 must be from one of the pre-defined types of Table 3.2.
- 6894 2.  $\mathbf{D}(w)$  must be compatible with the domain of the reduction binary operator,  $\mathbf{D}(F_b)$ .
- 6895 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6896 of the accumulation operator and  $\mathbf{D}(F_b)$ , must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
6897 mulation operator.
- 6898 4.  $\mathbf{D}(A)$  must be compatible with the domain of the binary reduction operator,  $\mathbf{D}(F_b)$ .

6899 Two domains are compatible with each other if values from one domain can be cast to values in  
6900 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6901 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6902 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
6903 error listed above is returned.

6904 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6905 denotes copy):

- 6906 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6907 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 6908 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 6909 (b) If `mask  $\neq$  GrB_NULL`,
- 6910 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 6911 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 6912 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6913 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

6914 The internal vectors and masks are checked for dimension compatibility. The following conditions  
 6915 must hold:

- 6916 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6917 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6918 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-  
 6919 match error listed above is returned.

6920 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 6921 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6922 We carry out the reduce and any additional associated operations. We describe this in terms of  
 6923 two intermediate vectors:

- 6924 •  $\tilde{\mathbf{t}}$ : The vector holding the result from reducing along the rows of input matrix  $\tilde{\mathbf{A}}$ .
- 6925 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6926 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6927 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6928 The value of each of its elements is computed by

$$6929 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6930 where  $\bigoplus = \odot(F_b)$ .

6931 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6932 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

6933 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6934 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6935 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6936 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6937 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 6938 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6939 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6940 \quad & \\ 6941 \end{aligned}$$

6942 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6943 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6944 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6945 mask which acts as a “write mask”.

6946 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
6947 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$6948 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6949 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
6950 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
6951 mask are unchanged:

$$6952 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6953 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
6954 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
6955 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
6956 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6957 sequence.

#### 6958 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

6959 Reduce all stored values into a single scalar.

#### 6960 C Syntax

```
6961 // scalar value + monoid (only)
6962 GrB_Info GrB_reduce(<type>          *val,
6963                    const GrB_BinaryOp accum,
6964                    const GrB_Monoid  op,
6965                    const GrB_Vector  u,
```

```

6966             const GrB_Descriptor desc);
6967
6968 // GraphBLAS Scalar + monoid
6969 GrB_Info GrB_reduce(GrB_Scalar      s,
6970                   const GrB_BinaryOp accum,
6971                   const GrB_Monoid  op,
6972                   const GrB_Vector  u,
6973                   const GrB_Descriptor desc);
6974
6975 // GraphBLAS Scalar + binary operator
6976 GrB_Info GrB_reduce(GrB_Scalar      s,
6977                   const GrB_BinaryOp accum,
6978                   const GrB_BinaryOp op,
6979                   const GrB_Vector  u,
6980                   const GrB_Descriptor desc);

```

## 6981 Parameters

6982 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
6983 a value that may be accumulated (optionally) with the result of the reduction  
6984 operation. On output, this scalar holds the results of the operation.

6985 **accum** (IN) An optional binary operator used for accumulating entries into an exist-  
6986 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,  
6987 **GrB\_NULL** should be specified.

6988 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
6989 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
6990 otherwise, the outcome of the operation is undefined.

6991 **u** (IN) The GraphBLAS vector on which reduction will be performed.

6992 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6993 should be specified. Non-default field/value pairs are listed as follows:

6995 Param	Field	Value	Description
------------	-------	-------	-------------

6996 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
6997 tions. There are currently no non-default field/value pairs that can be set for this  
6998 operation.

## 6999 Return Values

7000 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
7001 cessfully, and the output scalar (**s** or **val**) is ready to be used in the  
7002 next method of the sequence.



7003	<b>GrB_PANIC</b>	Unknown internal error.
7004	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the opaque
7005		GraphBLAS objects (input or output) is in an invalid state caused
7006		by a previous execution error. Call <b>GrB_error()</b> to access any error
7007		messages generated by the implementation.
7008	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
7009	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized by
7010		a call to a respective constructor.
7011	<b>GrB_NULL_POINTER</b>	val pointer is NULL.
7012	<b>GrB_DOMAIN_MISMATCH</b>	The domains of input and output arguments are incompatible with
7013		the corresponding domains of the accumulation operator, or reduce
7014		operator.

## 7015 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):

$$\text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7016 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7017 Logically, this operation occurs in three steps:

7018 **Setup** The internal vector used in the computation is formed and its domain is tested for  
7019 compatibility.

7020 **Compute** The indicated computations are carried out.

7021 **Output** The result is written into the output scalar.

7022 One vector argument is used in this **GrB\_reduce** operation:

- 7023 1.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7024 The output scalar, argument vector, reduction operator and accumulation operator (if provided)  
7025 are tested for domain compatibility as follows:

- 7026 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  
7027  $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

- 7028 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  
 7029  $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
 7030 be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 7031 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  $F_b$ ).

7032 Two domains are compatible with each other if values from one domain can be cast to values in  
 7033 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7034 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7035 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
 7036 error listed above is returned.

7037 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one  
 7038 of the following occurs depending on the output variant:

$$7039 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7040 or

$$7041 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7042 where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7043 For all other cases, the internal vector and scalar used in the computation is formed ( $\leftarrow$  denotes  
 7044 copy):

- 7045 1. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 7046 2. Scalar  $\tilde{s} \leftarrow \text{s}$  (GraphBLAS scalar case).

7047 We are now ready to carry out the reduction and any additional associated operations. An inter-  
 7048 mediate scalar result  $t$  is computed as follows:

$$7049 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

7050 where  $\oplus = \odot(\text{op})$ .

7051 The final reduction value is computed as follows:

$$7052 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7053 or

$$7054 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB\_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7055 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
7056 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7057 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7058 Reduce all stored values into a single scalar.

### 7059 C Syntax

```
7060 // scalar value + monoid (only)
7061 GrB_Info GrB_reduce(<type>          *val,
7062                   const GrB_BinaryOp accum,
7063                   const GrB_Monoid   op,
7064                   const GrB_Matrix   A,
7065                   const GrB_Descriptor desc);
7066
7067 // GraphBLAS Scalar + monoid
7068 GrB_Info GrB_reduce(GrB_Scalar      s,
7069                   const GrB_BinaryOp accum,
7070                   const GrB_Monoid   op,
7071                   const GrB_Matrix   A,
7072                   const GrB_Descriptor desc);
7073
7074 // GraphBLAS Scalar + binary operator
7075 GrB_Info GrB_reduce(GrB_Scalar      s,
7076                   const GrB_BinaryOp accum,
7077                   const GrB_BinaryOp op,
7078                   const GrB_Matrix   A,
7079                   const GrB_Descriptor desc);
```

### 7080 Parameters

7081 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7082 a value that may be accumulated (optionally) with the result of the reduction  
7083 operation. On output, this scalar holds the results of the operation.

7084 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or  
7085 **val**) value. If assignment rather than accumulation is desired, GrB\_NULL should  
7086 be specified.

7087 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7088 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7089 otherwise, the outcome of the operation is undefined.

7090 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

7091 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7092 should be specified. Non-default field/value pairs are listed as follows:

7093

7094 

Param	Field	Value	Description
-------	-------	-------	-------------

7095 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7096 tions. There are currently no non-default field/value pairs that can be set for this  
7097 operation.

## 7098 Return Values

7099 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
7100 cessfully, and the output scalar (s or val) is ready to be used in the  
7101 next method of the sequence.

7102 GrB\_PANIC Unknown internal error.

7103 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7104 GraphBLAS objects (input or output) is in an invalid state caused  
7105 by a previous execution error. Call GrB\_error() to access any error  
7106 messages generated by the implementation.

7107 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7108 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7109 a call to a respective constructor.

7110 GrB\_NULL\_POINTER val pointer is NULL.

7111 GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
7112 the corresponding domains of the accumulation operator, or reduce  
7113 operator.

## 7114 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):

$$\text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7115 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7116 Logically, this operation occurs in three steps:

7117       **Setup** The internal matrix used in the computation is formed and its domain is tested for  
 7118       compatibility.

7119       **Compute** The indicated computations are carried out.

7120       **Output** The result is written into the output scalar.

7121   One matrix argument is used in this GrB\_reduce operation:

7122       1.  $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7123   The output scalar, argument matrix, reduction operator and accumulation operator (if provided)  
 7124   are tested for domain compatibility as follows:

7125       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  
 7126        $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

7127       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  
 7128        $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
 7129       be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

7130       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  $F_b$ ).

7131   Two domains are compatible with each other if values from one domain can be cast to values in  
 7132   the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7133   compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7134   any compatibility rule above is violated, execution of GrB\_reduce ends and the domain mismatch  
 7135   error listed above is returned.

7136   The number of values stored in the input,  $A$ , is checked. If there are no stored values in  $A$ , then  
 7137   one of the following occurs depending on the output variant:

$$7138 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB\_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7139   or

$$7140 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB\_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7141   where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with GrB\_SUCCESS.

7142   For all other cases, the internal matrix and scalar used in the computation is formed ( $\leftarrow$  denotes  
 7143   copy):

7144       1. Matrix  $\tilde{A} \leftarrow A$ .

7145       2. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

7146 We are now ready to carry out the reduce and any additional associated operations. An intermediate  
 7147 scalar result  $t$  is computed as follows:

$$7148 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7149 where  $\oplus = \odot(\text{op})$ .

7150 The final reduction value is computed as follows:

$$7151 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7152 or

$$7153 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB\_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7154 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
 7155 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7156 4.3.11 transpose: Transpose rows and columns of a matrix

7157 This version computes a new matrix that is the transpose of the source matrix.

#### 7158 C Syntax

```
7159         GrB_Info GrB_transpose(GrB_Matrix      C,
7160                               const GrB_Matrix Mask,
7161                               const GrB_BinaryOp accum,
7162                               const GrB_Matrix A,
7163                               const GrB_Descriptor desc);
```

#### 7164 Parameters

7165 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 7166 that may be accumulated with the result of the transpose operation. On output,  
 7167 the matrix holds the results of the operation.

7168 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 7169 stored into the output matrix C. The mask dimensions must match those of the  
 7170 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
 7171 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
 7172 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
 7173 dimensions of C), GrB\_NULL should be specified.

7174        **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
7175                entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
7176                specified.

7177        **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7178        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7179                should be specified. Non-default field/value pairs are listed as follows:

7180

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

7181

## 7182    **Return Values**

7183        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
7184                blocking mode, this indicates that the compatibility tests on di-  
7185                mensions and domains for the input arguments passed successfully.  
7186                Either way, output matrix **C** is ready to be used in the next method  
7187                of the sequence.

7188        **GrB\_PANIC** Unknown internal error.

7189        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
7190                GraphBLAS objects (input or output) is in an invalid state caused  
7191                by a previous execution error. Call **GrB\_error()** to access any error  
7192                messages generated by the implementation.

7193        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

7194        **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
7195                a call to **new** (or **Matrix\_dup** for matrix parameters).

7196        **GrB\_DIMENSION\_MISMATCH** **mask**, **C** and/or **A** dimensions are incompatible.

7197        **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the cor-  
7198                responding domains of the accumulation operator, or the mask's do-  
7199                main is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCT**  
7200                is not set).

## 7201 Description

7202 GrB\_transpose computes the result of performing a transpose of the input matrix:  $C = A^T$ ; or, if an  
 7203 optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A^T$ . We note that the input matrix  
 7204 A can itself be optionally transposed before the operation, which would cause either an assignment  
 7205 from A to C or an accumulation of A into C.

7206 Logically, this operation occurs in three steps:

7207     **Setup** The internal matrix and mask used in the computation are formed and their domains  
 7208             and dimensions are tested for compatibility.

7209     **Compute** The indicated computations are carried out.

7210     **Output** The result is written into the output matrix, possibly under control of a mask.

7211 Up to three matrix arguments are used in this GrB\_transpose operation:

- 7212     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7213     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)
- 7214     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7215 The argument matrices and accumulation operator (if provided) are tested for domain compatibility  
 7216 as follows:

- 7217     1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 7218         must be from one of the pre-defined types of Table 3.2.
- 7219     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$  of the input matrix.
- 7220     3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 7221         of the accumulation operator and  $\mathbf{D}(A)$  of the input matrix must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 7222         of the accumulation operator.

7223 Two domains are compatible with each other if values from one domain can be cast to values in  
 7224 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7225 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7226 any compatibility rule above is violated, execution of GrB\_transpose ends and the domain mismatch  
 7227 error listed above is returned.

7228 From the argument matrices, the internal matrices and mask used in the computation are formed  
 7229 ( $\leftarrow$  denotes copy):

- 7230     1. Matrix  $\tilde{C} \leftarrow C$ .
- 7231     2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument Mask as follows:



- 7232 (a) If  $\text{Mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
7233  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 7234 (b) If  $\text{Mask} \neq \text{GrB\_NULL}$ ,
- 7235 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
7236  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
- 7237 ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
7238  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
- 7239 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$ .
- 7240 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

7241 The internal matrices and masks are checked for dimension compatibility. The following conditions  
7242 must hold:

- 7243 1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$ .
- 7244 2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$ .
- 7245 3.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$ .
- 7246 4.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$ .

7247 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension  
7248 mismatch error listed above is returned.

7249 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
7250 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7251 We are now ready to carry out the matrix transposition and any additional associated operations.  
7252 We describe this in terms of two intermediate matrices:

- 7253 •  $\widetilde{\mathbf{T}}$ : The matrix holding the transpose of  $\widetilde{\mathbf{A}}$ .
- 7254 •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

7255 The intermediate matrix

$$7256 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7257 is created.

7258 The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7259 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 7260 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$7261 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7262 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 7263 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned}
 7264 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 7265 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7266 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7267 \quad & \\
 7268 \quad &
 \end{aligned}$$

7269 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7270 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 7271 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 7272 mask which acts as a “write mask”.

- 7273 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 7274 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7275 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7276 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 7277 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 7278 mask are unchanged:

$$7279 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7280 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 7281 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 7282 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 7283 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 7284 sequence.

#### 7285 4.3.12 kronecker: Kronecker product of two matrices

7286 Computes the Kronecker product of two matrices. The result is a matrix.

#### 7287 C Syntax

```

7288      GrB_Info GrB_kronecker(GrB_Matrix      C,
7289                           const GrB_Matrix  Mask,
7290                           const GrB_BinaryOp accum,
7291                           const GrB_Semiring op,
7292                           const GrB_Matrix  A,
7293                           const GrB_Matrix  B,
7294                           const GrB_Descriptor desc);
7295  
```

```

7296     GrB_Info GrB_kronecker(GrB_Matrix      C,
7297                             const GrB_Matrix Mask,
7298                             const GrB_BinaryOp accum,
7299                             const GrB_Monoid op,
7300                             const GrB_Matrix A,
7301                             const GrB_Matrix B,
7302                             const GrB_Descriptor desc);
7303
7304     GrB_Info GrB_kronecker(GrB_Matrix      C,
7305                             const GrB_Matrix Mask,
7306                             const GrB_BinaryOp accum,
7307                             const GrB_BinaryOp op,
7308                             const GrB_Matrix A,
7309                             const GrB_Matrix B,
7310                             const GrB_Descriptor desc);

```

## 7311 Parameters

7312 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
7313 that may be accumulated with the result of the Kronecker product. On output,  
7314 the matrix holds the results of the operation.

7315 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
7316 stored into the output matrix C. The mask dimensions must match those of the  
7317 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
7318 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
7319 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
7320 dimensions of C), GrB\_NULL should be specified.

7321 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7322 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
7323 specified.

7324 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
7325 operation. Depending on which type is passed, the following defines the binary  
7326 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

7327 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

7328 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-  
7329 nored.

7330 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  
7331 is ignored.

7332 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
7333 product.

7334 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
7335 product.

7336 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7337 should be specified. Non-default field/value pairs are listed as follows:  
7338

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.

## 7340 Return Values

7341 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
7342 blocking mode, this indicates that the compatibility tests on di-  
7343 mensions and domains for the input arguments passed successfully.  
7344 Either way, output matrix C is ready to be used in the next method  
7345 of the sequence.

7346 GrB\_PANIC Unknown internal error.

7347 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7348 GraphBLAS objects (input or output) is in an invalid state caused  
7349 by a previous execution error. Call GrB\_error() to access any error  
7350 messages generated by the implementation.

7351 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7352 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7353 a call to new (or Matrix\_dup for matrix parameters).

7354 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

7355 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
7356 corresponding domains of the binary operator (op) or accumulation  
7357 operator, or the mask's domain is not compatible with bool (in the  
7358 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 7359 Description

7360 GrB\_kronecker computes the Kronecker product  $C = A \otimes B$  or, if an optional binary accumulation  
7361 operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$  (where matrices A and B can be optionally transposed).

7362 The Kronecker product is defined as follows:

7363

$$7364 \quad \mathbf{C} = \mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{0,0} \otimes \mathbf{B} & A_{0,1} \otimes \mathbf{B} & \dots & A_{0,n_A-1} \otimes \mathbf{B} \\ A_{1,0} \otimes \mathbf{B} & A_{1,1} \otimes \mathbf{B} & \dots & A_{1,n_A-1} \otimes \mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes \mathbf{B} & A_{m_A-1,1} \otimes \mathbf{B} & \dots & A_{m_A-1,n_A-1} \otimes \mathbf{B} \end{bmatrix}$$

7365 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  
7366 Kronecker product are defined as

$$7367 \quad \mathbf{C}(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7368 where  $\otimes$  is the multiplicative operator specified by the `op` parameter.

7369 Logically, this operation occurs in three steps:

7370 **Setup** The internal matrices and mask used in the computation are formed and their domains  
7371 and dimensions are tested for compatibility.

7372 **Compute** The indicated computations are carried out.

7373 **Output** The result is written into the output matrix, possibly under control of a mask.

7374 Up to four argument matrices are used in the `GrB_kronecker` operation:

- 7375 1.  $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 7376 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)
- 7377 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 7378 4.  $\mathbf{B} = \langle \mathbf{D}(\mathbf{B}), \mathbf{nrows}(\mathbf{B}), \mathbf{ncols}(\mathbf{B}), \mathbf{L}(\mathbf{B}) = \{(i, j, B_{ij})\} \rangle$

7379 The argument matrices, the "product" operator (`op`), and the accumulation operator (if provided)  
7380 are tested for domain compatibility as follows:

- 7381 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{Mask})$   
7382 must be from one of the pre-defined types of Table 3.2.
- 7383 2.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 7384 3.  $\mathbf{D}(\mathbf{B})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 7385 4.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 7386 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
7387 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
7388 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.

7422 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$   
7423  $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$  is created. The value of  
7424 each of its elements is computed by

$$7425 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7426 where  $\otimes$  is the multiplicative operator specified by the `op` parameter.

7427 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7428 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 7429 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$7430 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7431 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
7432 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$7433 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7434 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7435 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7436 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7439 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
7440 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
7441 mask which acts as a “write mask”.

- 7442 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
7443 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7444 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7445 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
7446 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
7447 mask are unchanged:

$$7448 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7449 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
7450 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
7451 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
7452 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7453 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
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>	<code>GrB_Vector_apply_IndexOp_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>	<code>GrB_Vector_apply_IndexOp_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>	<code>GrB_Vector_apply_IndexOp_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>	<code>GrB_Vector_apply_IndexOp_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>	<code>GrB_Vector_apply_IndexOp_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>	<code>GrB_Vector_apply_IndexOp_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>	<code>GrB_Vector_apply_IndexOp_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>	<code>GrB_Vector_apply_IndexOp_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>	<code>GrB_Vector_apply_IndexOp_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>	<code>GrB_Vector_apply_IndexOp_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>	<code>GrB_Vector_apply_IndexOp_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>	<code>GrB_Vector_apply_IndexOp_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,<i>other</i>,...)</code>	<code>GrB_Vector_apply_IndexOp_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>	<code>GrB_Matrix_apply_IndexOp_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>	<code>GrB_Matrix_apply_IndexOp_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>	<code>GrB_Matrix_apply_IndexOp_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>	<code>GrB_Matrix_apply_IndexOp_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>	<code>GrB_Matrix_apply_IndexOp_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,<i>other</i>,...)</code>	<code>GrB_Matrix_apply_IndexOp_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)</code>

Table 5.10: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)	GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)	GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)	GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)	GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)	GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)	GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)	GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)
GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)	GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)	GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)	GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)	GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)
GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)	GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)

Table 5.11: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_reduce(GrB_Vector,...,GrB_Monoid,...)	GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Vector,...)	GrB_Vector_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Vector,...)	GrB_Vector_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(bool*,...,GrB_Vector,...)	GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)
GrB_reduce(int8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)
GrB_reduce(uint8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)
GrB_reduce(int16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)
GrB_reduce(uint16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)
GrB_reduce(int32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)
GrB_reduce(uint32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)
GrB_reduce(int64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)
GrB_reduce(uint64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)
GrB_reduce(float*,...,GrB_Vector,...)	GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)
GrB_reduce(double*,...,GrB_Vector,...)	GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)
GrB_reduce( <i>other</i> *,...,GrB_Vector,...)	GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)	GrB_Matrix_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)	GrB_Matrix_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)
GrB_reduce(bool*,...,GrB_Matrix,...)	GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)
GrB_reduce(int8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)
GrB_reduce(uint8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)
GrB_reduce(int16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)
GrB_reduce(uint16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)
GrB_reduce(int32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)
GrB_reduce(uint32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)
GrB_reduce(int64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)
GrB_reduce(uint64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)
GrB_reduce(float*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)
GrB_reduce(double*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)
GrB_reduce( <i>other</i> *,...,GrB_Matrix,...)	GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)
GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)



# 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.

- 7484 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified  
7485 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats  
7486 have been deferred.
- 7487 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7488 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return  
7489 error if duplicate locations encountered.
- 7490 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7491 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7492 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type T (not to  
7493 be confused with the proposed unary operator).
- 7494 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added  
7495 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to  
7496 a method that is not supported by the implementation.
- 7497 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque  
7498 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7499 • (Issue BB-45) Removed language about annihilators.
- 7500 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7501 • Updated a number algorithms in the appendix to use new operations and methods.
- 7502 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the  
7503 specification.
- 7504 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and  
7505 `GRB_SUBVERSION`.
- 7506 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7507 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection  
7508 fixed to read `eWiseAdd`/set union.
- 7509 • Typographical error (`NEQ`  $\rightarrow$  `NE`) in Description of Table 3.8.

7510 Changes in 1.3.0 (Released: 25 September 2019):

- 7511 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque  
7512 GraphBLAS object as an argument.
- 7513 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7514 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a  
7515 scalar.

7516  
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

- (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.

- 7549 • Fixed miscellaneous typographical errors (such as  $\otimes$ ,  $\oplus$ ).
- 7550 Changes in 1.2.0:
- 7551 • Removed "provisional" clause.
- 7552 Changes in 1.1.0:
- 7553 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
  - 7554 `assign` operations.
  - 7555 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
  - 7556 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
  - 7557 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
  - 7558 from the domains of the binary operator provided.
  - 7559 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
  - 7560 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
  - 7561 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
  - 7562 which is returned when the capacities of the output arrays are insufficient to hold all of the
  - 7563 tuples.
  - 7564 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
  - 7565 face.
  - 7566 • Added replace flag (`z`) notation to Table 4.1.
  - 7567 • Updated the “Mathematical Description” of the `assign` operation in Table 4.1.
  - 7568 • Added triangle counting example.
  - 7569 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
  - 7570 sections of GraphBLAS operations when the respective text was the “standard” text (i.e.,
  - 7571 identical in a majority of the operations).
  - 7572 • Fixed typographical errors.
- 7573 Changes in 1.0.2:
- 7574 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
  - 7575 matrices and avoid casting issues in certain implementations.
  - 7576 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
  - 7577 erased outside the assigned area.
  - 7578 • Changes non-polymorphic interface:
    - 7579 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.



- 7580           – Renamed GrB\_Vector\_reduce\_BinaryOp to GrB\_Matrix\_reduce\_BinaryOp.
- 7581           – Renamed GrB\_Vector\_reduce\_Monoid to GrB\_Matrix\_reduce\_Monoid.
- 7582       • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7583       • 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.

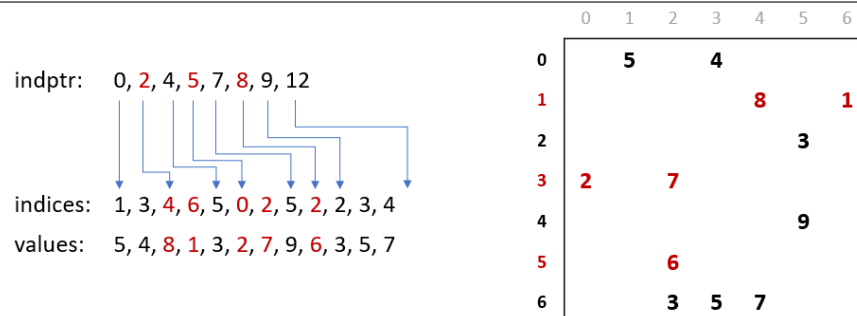


Figure B.1: Data layout for CSR format.

**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.

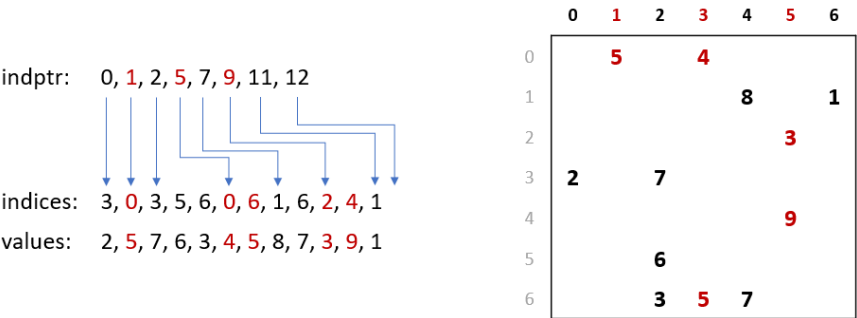


Figure B.2: Data layout for CSC format.

**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.

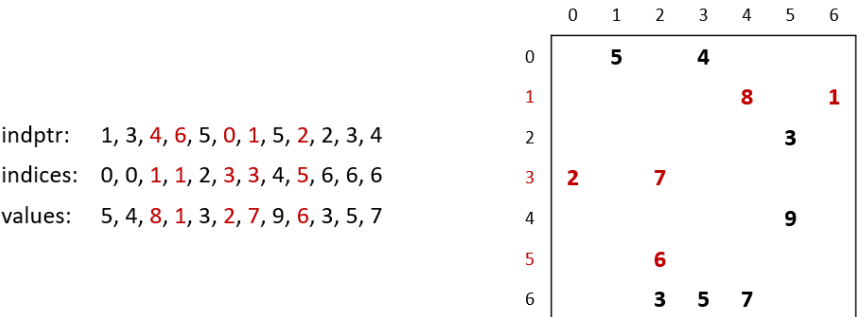


Figure B.3: Data layout for COO format.

7616 **Appendix C**

7617 **Examples**

## C.1 Example: Level breadth-first search (BFS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i] = 0$ . (Vector  $v$  should be empty on input.)
11  */
12  GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
13  {
14      GrB_Index n;
15      GrB_Matrix_nrows(&n,A);                //  $n = \#$  of rows of  $A$ 
16
17      GrB_Vector_new(v,GrB_INT32,n);          // Vector<int32_t>  $v(n)$ 
18
19      GrB_Vector q;                            // vertices visited in each level
20      GrB_Vector_new(&q,GrB_BOOL,n);          // Vector<bool>  $q(n)$ 
21      GrB_Vector_setElement(q,(bool)true,s);  //  $q[s] = \text{true}$ , false everywhere else
22
23      /*
24       * BFS traversal and label the vertices.
25       */
26      int32_t d = 0;                            //  $d = \text{level in BFS traversal}$ 
27      bool succ = false;                        //  $\text{succ} == \text{true}$  when some successor found
28      do {
29          ++d;                                    // next level (start with 1)
30          GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL); //  $v[q] = d$ 
31          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
32                q,A,GrB_DESC_RC);                //  $q[!v] = q \parallel A$ ; finds all the
33                                                  // unvisited successors from current  $q$ 
34          GrB_reduce(&succ,GrB_NULL,GrB_LOR_MONOID_BOOL,
35                q,GrB_NULL);                    //  $\text{succ} = \parallel(q)$ 
36      } while (succ);                            // if there is no successor in  $q$ , we are done.
37
38      GrB_free(&q);                                //  $q$  vector no longer needed
39
40      return GrB_SUCCESS;
41  }

```

## C.2 Example: Level BFS in GraphBLAS using apply

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i]$  does not have a stored element.
11  * Vector  $v$  should be uninitialized on input.
12  */
13  GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index n;
16      GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
17
18      GrB_Vector_new(v,GrB_INT32,n);     // Vector<int32_t>  $v(n) = 0$ 
19
20      GrB_Vector q;                      // vertices visited in each level
21      GrB_Vector_new(&q,GrB_BOOL,n);     // Vector<bool>  $q(n) = \text{false}$ 
22      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
23
24      /*
25       * BFS traversal and label the vertices.
26       */
27      int32_t level = 0;                  // level = depth in BFS traversal
28      GrB_Index nvals;
29      do {
30          ++level;                        // next level (start with 1)
31          GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,
32                  GrB_SECOND_INT32,q,level,GrB_NULL); //  $v[q] = \text{level}$ 
33          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
34                  q,A,GrB_DESC_RC);      //  $q[!v] = q \ || \ \&\& \ A$ ; finds all the
35                                          // unvisited successors from current  $q$ 
36          GrB_Vector_nvals(&nvals, q);
37      } while (nvals);                    // if there is no successor in  $q$ , we are done.
38
39      GrB_free(&q);                       //  $q$  vector no longer needed
40
41      return GrB_SUCCESS;
42  }

```

## C.3 Example: Parent BFS in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a binary  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS
9   * traversal of the graph and sets  $parents[i]$  to the index of vertex  $i$ 's parent.
10  * The parent of the root vertex,  $s$ , will be set to itself ( $parents[s] = s$ ). If
11  * vertex  $i$  is not reachable from  $s$ ,  $parents[i]$  will not contain a stored value.
12  */
13  GrB_Info BFS(GrB_Vector *parents, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index N;
16      GrB_Matrix_nrows(&N, A);           //  $N = \#$  vertices
17
18      GrB_Vector_new(parents, GrB_UINT64, N);
19      GrB_Vector_setElement(*parents, s, s);    //  $parents[s] = s$ 
20
21      GrB_Vector wavefront;
22      GrB_Vector_new(&wavefront, GrB_UINT64, N);
23      GrB_Vector_setElement(wavefront, 1UL, s);    //  $wavefront[s] = 1$ 
24
25      /*
26       * BFS traversal and label the vertices.
27       */
28      GrB_Index nvals;
29      GrB_Vector_nvals(&nvals, wavefront);
30
31      while (nvals > 0)
32      {
33          // convert all stored values in wavefront to their 0-based index
34          GrB_apply(wavefront, GrB_NULL, GrB_NULL, GrB_ROWINDEX_INT64,
35                  wavefront, 0UL, GrB_NULL);
36
37          // "FIRST" because left-multiplying wavefront rows. Masking out the parent
38          // list ensures wavefront values do not overwrite parents already stored.
39          GrB_vxm(wavefront, *parents, GrB_NULL, GrB_MIN_FIRST_SEMIRING_UINT64,
40                  wavefront, A, GrB_DESC_RSC);
41
42          // Don't need to mask here since we did it in vxm. Merges new parents in
43          // current wavefront with existing parents:  $parents += wavefront$ 
44          GrB_apply(*parents, GrB_NULL, GrB_PLUS_UINT64,
45                  GrB_IDENTITY_UINT64, wavefront, GrB_NULL);
46
47          GrB_Vector_nvals(&nvals, wavefront);
48      }
49
50      GrB_free(&wavefront);
51
52      return GrB_SUCCESS;
53  }

```



## C.4 Example: Betweenness centrality (BC) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ ,
9   * compute the BC-metric vector  $\delta$ , which should be empty on input.
10  */
11 GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n,A);                //  $n = \#$  of vertices in graph
15
16     GrB_Vector_new(delta, GrB_FP32, n);      // Vector<float>  $\delta(n)$ 
17
18     GrB_Matrix sigma;
19     GrB_Matrix_new(&sigma, GrB_INT32, n, n); //  $\text{Matrix}<\text{int32}_t> \text{sigma}(n,n)$ 
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     * ( $t1, t2, t3, t4$ ) are temporary vectors
52     */
53     GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
54     GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
55     GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
56     GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
57
58     for (int i=d-1; i>0; i--)
59     {
60         GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t1 = 1 + \delta$ 
61         GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, t1, *delta, GrB_NULL);
62         GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t2 = \text{sigma}[i,:]$ 
63         GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t2 = (1 + \delta) / \text{sigma}[i,:]$ 
64         GrB_mvx(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
65                 // add contributions made by

```

```

63         A, t2, GrB_NULL);
64     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, GrB_DESC_T0); // t4 = sigma[i-1,:]
65     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, GrB_TIMES_FP32, t4, t3, GrB_NULL); // t4 = sigma[i-1,:]*t3
66     GrB_eWiseAdd(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, *delta, t4, GrB_NULL); // accumulate into delta
67 }
68
69 GrB_free(&sigma);
70 GrB_free(&q); GrB_free(&p);
71 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
72
73 return GrB_SUCCESS;
74 }

```

## C.5 Example: Batched BC in GraphBLAS

```

1  #include <stdlib.h>
2  #include "GraphBLAS.h" // in addition to other required C headers
3
4  // Compute partial BC metric for a subset of source vertices, s, in graph A
5  GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
6  {
7      GrB_Index n;
8      GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
9      GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
10
11     // index and value arrays needed to build numsp
12     GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
13     int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
14     for(int i=0; i<nsver; ++i) {
15         i_nsver[i] = i;
16         ones[i] = 1;
17     }
18
19     // numsp: structure holds the number of shortest paths for each node and starting vertex
20     // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
21     GrB_Matrix numsp;
22     GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
23     GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
24     free(i_nsver); free(ones);
25
26     // frontier: Holds the current frontier where values are path counts.
27     // Initialized to out vertices of each source node in s.
28     GrB_Matrix frontier;
29     GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
30     GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, GrB_DESC_RCT0);
31
32     // sigma: stores frontier information for each level of BFS phase. The memory
33     // for an entry in sigmas is only allocated within the do-while loop if needed.
34     // n is an upper bound on diameter.
35     GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n);
36
37     int32_t d = 0; // BFS level number
38     GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
39
40     // ----- The BFS phase (forward sweep) -----
41     do {
42         // sigmas[d](:,s) = d^th 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 }
```