

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

Version 2.1

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

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

Generated on 2023/06/21 at 14:13:30 EDT

<sup>†</sup>Based on *GraphBLAS Mathematics* by Jeremy Kepner

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

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

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

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

# Contents

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

49	3.4	Algebraic objects, operators and associated functions . . . . .	35
50	3.4.1	Operators . . . . .	36
51	3.4.2	Monoids . . . . .	41
52	3.4.3	Semirings . . . . .	41
53	3.5	Collections . . . . .	45
54	3.5.1	Scalars . . . . .	45
55	3.5.2	Vectors . . . . .	45
56	3.5.3	Matrices . . . . .	46
57	3.5.3.1	External matrix formats . . . . .	46
58	3.5.4	Masks . . . . .	46
59	3.6	Descriptors . . . . .	47
60	3.7	Fields . . . . .	48
61	3.7.1	Input Types . . . . .	51
62	3.7.1.1	INT32 Handling . . . . .	51
63	3.7.1.2	GrB_Scalar Handling . . . . .	51
64	3.7.1.3	String (char*) Handling . . . . .	51
65	3.7.1.4	void* Handling . . . . .	51
66	3.7.2	Hints . . . . .	51
67	3.7.3	GrB_NAME . . . . .	52
68	3.8	GrB_Info return values . . . . .	54
69	<b>4</b>	<b>Methods</b>	<b>57</b>
70	4.1	Context methods . . . . .	57
71	4.1.1	init: Initialize a GraphBLAS context . . . . .	57
72	4.1.2	finalize: Finalize a GraphBLAS context . . . . .	58
73	4.1.3	getVersion: Get the version number of the standard. . . . .	59
74	4.2	Object methods . . . . .	59
75	4.2.1	Get and Set methods . . . . .	60
76	4.2.1.1	get: Query the value of an object . . . . .	60
77	4.2.1.2	set: Set field of an object . . . . .	60
78	4.2.2	Algebra methods . . . . .	61

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

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

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

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



# List of Tables

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

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

# 229 List of Figures

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

## Acknowledgments

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

Those who served as C API Subcommittee members for GraphBLAS 2.1 are (in alphabetical order):

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

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

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

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

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

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

- NSF Graduate Research Fellowship under Grant No. DGE 1752814 and by the NSF under Award No. 1823034 with the University of California, Berkeley
- The Department of Energy Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231

- Intel Corporation
- Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute [DM-0003727, DM19-0929, DM21-0090]
- International Business Machines Corporation

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



# Chapter 1

## Introduction

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

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

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

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

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

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

is not possible to express the signature of that method.

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

The remainder of this document is organized as follows:

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



## Chapter 2

# Basic concepts

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

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

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

## 2.1 Glossary

### 2.1.1 GraphBLAS API basic definitions

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

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

### 2.1.2 GraphBLAS objects and their structure

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

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

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

### 2.1.3 Algebraic structures used in the GraphBLAS

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

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

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

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

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

#### 2.1.4 The execution of an application using the GraphBLAS C API

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

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

### 2.1.5 GraphBLAS methods: behaviors and error conditions

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

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

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

## 2.2 Notation

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

## 2.3 Mathematical foundations

Graphs can be represented in terms of matrices. The values stored in these matrices correspond to attributes (often weights) of edges in the graph.<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 3.16. The return values fall into three groups: informational, API errors, and execution errors. While API and execution errors take on negative values, informational return values listed in Table 3.16(a) are non-negative and include `GrB_SUCCESS` (a value of 0) and `GrB_NO_VALUE`.

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

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

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

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

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

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

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





## Chapter 3

# Objects

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

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

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

### 3.2 Indices, index arrays, and scalar arrays

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

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

```
typedef uint64_t GrB_Index;
```

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

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

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

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

### 806 3.3 Types (domains)

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

---

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

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

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

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

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

---

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

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

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

### 3.4 Algebraic objects, operators and associated functions

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

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

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

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

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

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

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

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

### 3.4.1 Operators

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

---

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

(a) Properties of algebraic objects.

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

(b) Recipes for algebraic objects.

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

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

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

---

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

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

857 A GraphBLAS *index unary operator*  $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB\_Index}), D_{in_2}, f_i \rangle$  is defined by three  
858 domains,  $D_{out}$ ,  $D_{in_1}$ ,  $D_{in_2}$ , the domain of GraphBLAS indices, and an operation  $f_i : D_{in_1} \times I_{U64}^2 \times$   
859  $D_{in_2} \rightarrow D_{out}$  (where  $I_{U64}$  corresponds to the domain of a `GrB_Index`). For a given GraphBLAS  
860 index operator  $F_i$ , we define  $\mathbf{D}_{out}(F_i) = D_{out}$ ,  $\mathbf{D}_{in_1}(F_i) = D_{in_1}$ ,  $\mathbf{D}_{in_2}(F_i) = D_{in_2}$ , and  $\mathbf{f}(F_i) = f_i$ .

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

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

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

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

Operator type Type	GraphBLAS identifier	Domains (– is don’t care) $A, u$ $i, j$ $s$ result				Description
GrB_IndexUnaryOp	GrB_ROWINDEX_ $I_{32/64}$	–	$I_{U64}$	$I_{32/64}$	$I_{32/64}$	$f(A_{ij}, i, j, s) = (i + s)$ , replace with its row index (+ s)
		–	$I_{U64}$	$I_{32/64}$	$I_{32/64}$	$f(u_i, i, 0, s) = (i + s)$
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 $x$ INT $x$ FP $x$	0 0 0	addition
GrB_TIMES_MONOID_ $T$	UINT $x$ INT $x$ FP $x$	1 1 1	multiplication
GrB_MIN_MONOID_ $T$	UINT $x$ INT $x$ FP $x$	UINT $x$ _MAX INT $x$ _MAX INFINITY	minimum
GrB_MAX_MONOID_ $T$	UINT $x$ INT $x$ FP $x$	0 INT $x$ _MIN -INFINITY	maximum
GrB_LOR_MONOID_BOOL	BOOL	false	logical OR
GrB_LAND_MONOID_BOOL	BOOL	true	logical AND
GrB_LXOR_MONOID_BOOL	BOOL	false	logical XOR (not equal)
GrB_LXNOR_MONOID_BOOL	BOOL	true	logical XNOR (equal)

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

GraphBLAS identifier	Domains, $T$ ( $T \times T \rightarrow T$ )	+ identity $\times$ annihilator	Description
<code>GrB_PLUS_TIMES_SEMIRING_T</code>	<code>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
	$\text{INT}x$	$\text{INT}x\_MIN$	
	$\text{FP}x$	$-INFINITY$	

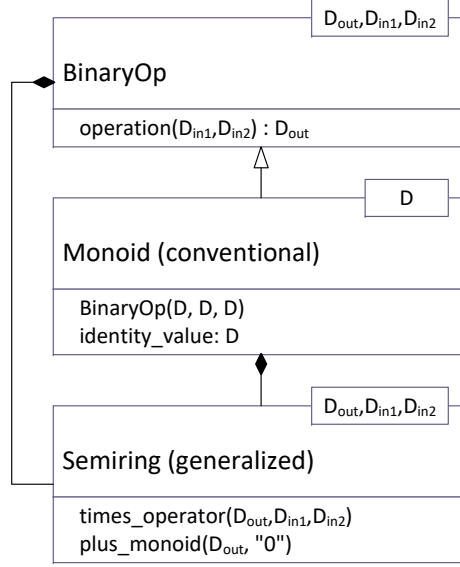


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

## 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 Descriptors

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

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

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

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

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

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

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

## 3.7 Fields

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

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

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

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



---

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

(a) Types used with GraphBLAS descriptors.

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

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

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

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

Value Name	Value	Description
<code>GrB_DEFAULT</code>	0	The default (unset) value for each field.
<code>GrB_REPLACE</code>	1	Clear the output object before assigning computed values.
<code>GrB_COMP</code>	2	Use the complement of the associated object. When combined with <code>GrB_STRUCTURE</code> , the complement of the structure of the associated object is used without evaluating the values stored.
<code>GrB_TRAN</code>	3	Use the transpose of the associated object.
<code>GrB_STRUCTURE</code>	4	The write mask is constructed from the structure (pattern of stored values) of the associated object. The stored values are not examined.
<code>GrB_COMP_STRUCTURE</code>	6	Shorthand for both <code>GrB_COMP</code> and <code>GrB_STRUCTURE</code> .

---

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

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

### 1019 3.7.1 Input Types

1020 Allowable types used in `GrB_get` and `GrB_set` are `INT32`, `GrB_Scalar`, `char*`, and `void*`. Each  
1021 `GrB_Field` is associated with exactly one of these types as defined in Table 3.13. Implementations  
1022 that add additional `GrB_Fields` must document the type associated with each `GrB_Field`.

#### 1023 3.7.1.1 INT32 Handling

1024 `INT32` types use a 32-bit signed integer type. This can be used both for numeric values as well as  
1025 enumerated C types. Enumerated types must specify the numeric value for each enum, and the  
1026 value specified must fit within the allowable 32-bit signed integer range.

#### 1027 3.7.1.2 GrB\_Scalar Handling

1028 When calling `GrB_get`, the user must provide an already initialized `GrB_Scalar` object to which  
1029 the implementation will write a value of the correct element type. When calling `GrB_set`, the  
1030 `GrB_Scalar` must not be empty, otherwise a `GrB_EMPTY_OBJECT` error is raised.

#### 1031 3.7.1.3 String (char\*) Handling

1032 When the input to `GrB_set` is a `char*` the input array is null terminated. The GraphBLAS imple-  
1033 mentation must copy this array into internal data structures. Using `GrB_get` for strings requires  
1034 two calls. First, call `GrB_get` with the field and object, but pass `size_t*` as the value argument.  
1035 The implementation will return the size of the string buffer that the user must create. Second, call  
1036 `GrB_get` with the field and object, this time passing a pointer to the newly created string buffer.  
1037 The GraphBLAS implementation will write to this buffer, including a trailing null terminator. The  
1038 size returned in the first call will include enough bytes for the null terminator.

#### 1039 3.7.1.4 void\* Handling

1040 When the input to `GrB_set` is a `void*`, an extra `size_t` argument is passed to indicate the size of the  
1041 buffer. The GraphBLAS implementation must copy this many bytes from the buffer into internal  
1042 data structures. Similar to reading strings, `GrB_get` must be called twice for `void*`. The first call  
1043 passes `size_t*` to find the required buffer size. The user must create a buffer and then pass the  
1044 pointer to `GrB_get`. The implementation will write to this buffer. No standard specification or  
1045 protocol is required for the contents of `void*`. It is meant to be a mechanism to allow full freedom  
1046 for GraphBLAS implementations with needs that cannot be handled using `INT32`, `GrB_Scalar`, or  
1047 Strings.

### 1048 3.7.2 Hints

1049 Several fields are *hints* (marked H in Table 3.13). Hints are used to represent intended use cases  
1050 or best guesses, but do not determine strict behavior. When `GrB_set` is called with a hint, the

1051 GraphBLAS implementation should return `GrB_SUCCESS`, but is free to use or ignore the hint.  
1052 When `GrB_get` is called, the implementation should return a best guess on the correct answer. If  
1053 there is no clear answer, the implementation should return `GrB_UNKNOWN`.

### 1054 3.7.3 `GrB_NAME`

1055 The `GrB_NAME` field is a special case regarding writability. All user-defined objects have a  
1056 `GrB_NAME` field which defaults to an empty string. Collections and `GrB_Descriptors` may have  
1057 their `GrB_NAME` set at any time. User-defined algebraic objects and `GrB_Types` may only have  
1058 their `GrB_NAME` set once to a globally unique value. Attempting to set this field after it has  
1059 already been set will return a `GrB_ALREADY_SET` error code.

1060 Built-in algebraic objects and `GrB_Types` have names which can be read, but not written to. The  
1061 name returned will be the string form of the `GrB_Type` listed in Table 3.2 or the GraphBLAS  
1062 identifier listed in Tables 3.5, 3.6, 3.7, 3.8, and 3.9. For example, the name of `GrB_BOOL` type  
1063 is "`GrB_BOOL`" (8 characters) and the name of `GrB_MIN_FP64` binary op is "`GrB_MIN_FP64`" (12  
1064 characters).

1065 The `GrB_NAME` of the global context is read-only and returns the name of the library implemen-  
1066 tation.

Table 3.13: Field values of type GrB\_Field enumeration, corresponding types, and the objects which must implement that GrB\_Field. Collection refers to GrB\_Matrix, GrB\_Vector, and GrB\_Scalar, Algebraic refers to Operators, Monoids, and Semirings, Type refers to GrB\_Type, and Global refers to the GrB\_Global context. All fields may be read, some may be written (denoted by W), and some are hints (denoted by H) which may be ignored by the implementation. For \* see 3.7

Field Name	W	H	Value	Implementing Objects	Type
GrB_OUTP_FIELD	W	—	0	GrB_Descriptor	INT32 (GrB_Desc_Value)
GrB_MASK_FIELD	W	—	1	GrB_Descriptor	INT32 (GrB_Desc_Value)
GrB_INP0_FIELD	W	—	2	GrB_Descriptor	INT32 (GrB_Desc_Value)
GrB_INP1_FIELD	W	—	3	GrB_Descriptor	INT32 (GrB_Desc_Value)
GrB_NAME	*		10	Global, Collection, Algebraic, Type	Null terminated char*
GrB_LIBRARY_VER_MAJOR	—	—	11	Global	INT32
GrB_LIBRARY_VER_MINOR	—	—	12	Global	INT32
GrB_LIBRARY_VER_PATCH	—	—	13	Global	INT32
GrB_API_VER_MAJOR	—	—	14	Global	INT32
GrB_API_VER_MINOR	—	—	15	Global	INT32
GrB_API_VER_PATCH	—	—	16	Global	INT32
GrB_BLOCKING_MODE	—	—	17	Global	INT32 (GrB_Mode)
GrB_STORAGE_ORIENTATION_HINT	W	H	100	Global, Collection	INT32 (GrB_Orientation)
GrB_ELTYPE_CODE	—	—	102	Collection, Type	INT32 (GrB_Type_Code)
GrB_INPUT1TYPE_CODE	—	—	103	Algebraic	INT32 (GrB_Type_Code)
GrB_INPUT2TYPE_CODE	—	—	104	Algebraic	INT32 (GrB_Type_Code)
GrB_OUTPUTTYPE_CODE	—	—	105	Algebraic	INT32 (GrB_Type_Code)
GrB_ELTYPE_STRING	—	—	106	Collection, Type	Null terminated char*
GrB_INPUT1TYPE_STRING	—	—	107	Algebraic	Null terminated char*
GrB_INPUT2TYPE_STRING	—	—	108	Algebraic	Null terminated char*
GrB_OUTPUTTYPE_STRING	—	—	109	Algebraic	Null terminated char*
GrB_SIZE	—	—	110	Type	INT32

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

Field Name	Description
GrB_NAME	The name of any GraphBLAS object, or the name of the library implementation.
GrB_BLOCKING_MODE	The blocking mode as set by GrB_init
GrB_STORAGE_ORIENTATION_HINT	Hint to the library that a collection is best stored in a row (lexicographic) or column (colexicographic) major format.
GrB_ELTYPE_(CODE/STRING)	The element type of a collection.
GrB_INPUT1TYPE_(CODE/STRING)	The type of the first argument to an operator. Returns GrB_NO_VALUE for Semirings and IndexUnaryOps which depend only on the index.
GrB_INPUT2TYPE_(CODE/STRING)	The type of the second argument to an operator. Returns GrB_NO_VALUE for Semirings, UnaryOps, and IndexUnaryOps which depend only on the index.
GrB_OUTPUTTYPE_(CODE/STRING)	The type of the output of an operator.
GrB_SIZE	The size of the GrB_Type.

### 3.8 GrB\_Info return values

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

---

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

(a) Field values of type GrB\_Orientation.

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

---

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

(a) Informational return values

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

(b) API errors

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

(c) Execution errors

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



## Chapter 4

# Methods

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

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

### 4.1 Context methods

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

#### 4.1.1 `init`: Initialize a GraphBLAS context

Creates and initializes a GraphBLAS C API context.

#### C Syntax

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

#### Parameters

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

## 1089 **Return Values**

1090                   GrB\_SUCCESS operation completed successfully.

1091                   GrB\_PANIC unknown internal error.

1092                   GrB\_INVALID\_VALUE invalid mode specified, or method called multiple times.

## 1093 **Description**

1094 The init method creates and initializes a GraphBLAS C API context. The argument to GrB\_init  
1095 defines the mode for the context. The two available modes are:

- 1096       • GrB\_BLOCKING: In this mode, each method in a sequence returns after its computations have  
1097        completed and output arguments are available to subsequent statements in an application.  
1098        When executing in GrB\_BLOCKING mode, the methods execute in program order.
- 1099       • GrB\_NONBLOCKING: In this mode, methods in a sequence may return after arguments in  
1100        the method have been tested for dimension and domain compatibility within the method  
1101        but potentially before their computations complete. Output arguments are available to sub-  
1102        sequent GraphBLAS methods in an application. When executing in GrB\_NONBLOCKING  
1103        mode, the methods in a sequence may execute in any order that preserves the mathematical  
1104        result defined by the sequence.

1105 An application can only create one context per execution instance. An application may only call  
1106 GrB\_Init once. Calling GrB\_Init more than once results in undefined behavior.

## 1107 **4.1.2 finalize: Finalize a GraphBLAS context**

1108 Terminates and frees any internal resources created to support the GraphBLAS C API context.

## 1109 **C Syntax**

1110                   GrB\_Info GrB\_finalize();

## 1111 **Return Values**

1112                   GrB\_SUCCESS operation completed successfully.

1113                   GrB\_PANIC unknown internal error.

## 1114 **Description**

1115 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS  
1116 C API context. `GrB_finalize` may only be called after a context has been initialized by calling  
1117 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-  
1118 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined  
1119 behavior.

## 1120 **4.1.3 getVersion: Get the version number of the standard.**

1121 Query the library for the version number of the standard that this library implements.

## 1122 **C Syntax**

```
1123         GrB_Info GrB_getVersion(unsigned int *version,  
1124                                unsigned int *subversion);
```

## 1125 **Parameters**

1126 version (OUT) On successful return will hold the value of the major version number.

1127 version (OUT) On successful return will hold the value of the subversion number.

## 1128 **Return Values**

1129 GrB\_SUCCESS operation completed successfully.

1130 GrB\_PANIC unknown internal error.

## 1131 **Description**

1132 The `getVersion` method is used to query the major and minor version number of the GraphBLAS  
1133 C API specification that the library implements at runtime. To support compile time queries the  
1134 following two macros shall also be defined by the library.

```
1135         #define GRB_VERSION      2  
1136         #define GRB_SUBVERSION  0
```

## 1137 **4.2 Object methods**

1138 This section describes methods that setup and operate on GraphBLAS opaque objects but are not  
1139 part of the the GraphBLAS math specification.

## 1140 4.2.1 Get and Set methods

1141 The methods in this section query and, optionally, set internal fields of GraphBLAS objects.

### 1142 4.2.1.1 get: Query the value of an object

#### 1143 C Syntax

```
1144 GrB_Info GrB_get(GrB_<OBJ> o, <type> value, GrB_Field field);
```

#### 1145 Parameters

1146 OBJ (IN) An existing, valid GraphBLAS object (collection, operation, type) which is  
1147 being queried. To indicate the global context, the constant `GrB_Global` is used.

1148 value (OUT) A pointer to or `GrB_Scalar` containing a value whose type is dependent on  
1149 field which will be filled with the current value of the field. type may be `int32_t*`,  
1150 `size_t*`, `GrB_Scalar`, `char*` or `void*`.

1151 field (IN) The field being queried.

#### 1152 Return Value

1153 `GrB_SUCCESS` The method completed successfully.

1154 `GrB_PANIC` unknown internal error.

1155 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1156 `GrB_UNINITIALIZED_OBJECT` the value parameter is `GrB_Scalar` and has not been initialized by  
1157 a call to `new`.

1158 `GrB_INVALID_VALUE` invalid value type provided for the field or invalid field.

#### 1159 Description

1160 Queries a field of an existing GraphBLAS object. The type of the argument is uniquely determined  
1161 by field. For the case of `char*` and `void*`, the value can be replaced with `size_t*` to get the required  
1162 buffer size to hold the response. Fields marked as hints in Table 3.13 will return a hint on how  
1163 best to use the object.

### 1164 4.2.1.2 set: Set field of an object

1165 Set the content for a field for an existing GraphBLAS object.

## 1166 C Syntax

```
1167     GrB_Info GrB_set(GrB_<OBJ> o, <type> value, GrB_Field field);
1168     GrB_Info GrB_set(GrB_<OBJ> o, void *value, GrB_Field field, size_t voidSize);
```

## 1169 Parameters

1170 OBJ (IN) The GraphBLAS object which is having field set. To indicate  
1171 the global context, the constant `GrB_Global` is used.

1172 value (IN) A value whose type is dependent on field. type may be a  
1173 `int32_t`, `GrB_Scalar`, `char*` or `void*`.

1174 field (IN) The field being set.

1175 voidSize (IN) The size of the `void*` buffer. Note that a size is not needed for  
1176 `char*` because the string is assumed null-terminated.

## 1177 Return Values

1178 `GrB_SUCCESS` The method completed successfully.

1179 `GrB_PANIC` unknown internal error.

1180 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1181 `GrB_UNINITIALIZED_OBJECT` the `GrB_Scalar` parameter has not been initialized by a call to `new`.

1182 `GrB_INVALID_VALUE` invalid value set on the field, invalid field, or field is read-only.

1183 `GrB_ALREADY_SET` this field has already been set, and may only be set once.

## 1184 Description

1185 Set a field of OBJ or the Global context to a new value.

## 1186 4.2.2 Algebra methods

### 1187 4.2.2.1 Type\_new: Construct a new GraphBLAS (user-defined) type

1188 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,  
1189 monoids, semirings, vectors and matrices.

## 1190 C Syntax

```
1191         GrB_Info GrB_Type_new(GrB_Type  *utype,  
1192                               size_t     sizeof(ctype));
```

## 1193 Parameters

1194 utype (INOUT) On successful return, contains a handle to the newly created user-defined  
1195 GraphBLAS type object.

1196 ctype (IN) A C type that defines the new GraphBLAS user-defined type.

## 1197 Return Values

1198 GrB\_SUCCESS operation completed successfully.

1199 GrB\_PANIC unknown internal error.

1200 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1201 GrB\_NULL\_POINTER utype pointer is NULL.

## 1202 Description

1203 Given a C type ctype, the Type\_new method returns in utype a handle to a new GraphBLAS type  
1204 that is equivalent to the C type. Variables of this ctype must be a struct, union, or fixed-size array.  
1205 In particular, given two variables, src and dst, of type ctype, the following operation must be a  
1206 valid way to copy the contents of src to dst:

```
1207         memcpy(&dst, &src, sizeof(ctype))
```

1208 A new, user-defined type utype should be destroyed with a call to GrB\_free(utype) when no longer  
1209 needed.

1210 It is not an error to call this method more than once on the same variable; however, the handle to  
1211 the previously created object will be overwritten.

### 1212 4.2.2.2 UnaryOp\_new: Construct a new GraphBLAS unary operator

1213 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types  
1214 (domains).

## 1215 C Syntax

```
1216      GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1217                              void          (*unary_func)(void*, const void*),  
1218                              GrB_Type      d_out,  
1219                              GrB_Type      d_in);
```

## 1220 Parameters

1221     **unary\_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1222     unary operator object.

1223     **unary\_func** (IN) a pointer to a user-defined function that takes one input parameter of **d\_in**'s  
1224     type and returns a value of **d\_out**'s type, both passed as **void** pointers. Specifically  
1225     the signature of the function is expected to be of the form:

```
1226           void func(void *out, const void *in);
```

1228     **d\_out** (IN) The **GrB\_Type** of the return value of the unary operator being created. Should  
1229     be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1230     BLAS type.

1231     **d\_in** (IN) The **GrB\_Type** of the input argument of the unary operator being created.  
1232     Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1233     GraphBLAS type.

## 1234 Return Values

1235     **GrB\_SUCCESS** operation completed successfully.

1236     **GrB\_PANIC** unknown internal error.

1237     **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1238     **GrB\_UNINITIALIZED\_OBJECT** any **GrB\_Type** parameter (for user-defined types) has not been ini-  
1239     tialized by a call to **GrB\_Type\_new**.

1240     **GrB\_NULL\_POINTER** **unary\_op** or **unary\_func** pointers are **NULL**.

## 1241 Description

1242     The **UnaryOp\_new** method creates a new GraphBLAS unary operator

1243      $f_u = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in}), \text{unary\_func} \rangle$

1244 and returns a handle to it in `unary_op`.

1245 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments  
1246 are aliased. In other words, for all invocations of the function:

```
1247     unary_func(out,in);
```

1248 the value of `out` must be the same as if the following code was executed:

```
1249     D(d_in) *tmp = malloc(sizeof(D(d_in)));
1250     memcpy(tmp,in,sizeof(D(d_in)));
1251     unary_func(out,tmp);
1252     free(tmp);
```

1253 It is not an error to call this method more than once on the same variable; however, the handle to  
1254 the previously created object will be overwritten.

#### 1255 4.2.2.3 BinaryOp\_new: Construct a new GraphBLAS binary operator

1256 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types  
1257 (domains).

### 1258 C Syntax

```
1259     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,
1260                               void          (*binary_func)(void*,
1261                               const void*,
1262                               const void*),
1263                               GrB_Type      d_out,
1264                               GrB_Type      d_in1,
1265                               GrB_Type      d_in2);
```

### 1266 Parameters

1267 `binary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1268 binary operator object.

1269 `binary_func` (IN) A pointer to a user-defined function that takes two input parameters of types  
1270 `d_in1` and `d_in2` and returns a value of type `d_out`, all passed as void pointers.  
1271 Specifically the signature of the function is expected to be of the form:

```
1272         void func(void *out, const void *in1, const void *in2);
1273
```



1274 **d\_out** (IN) The **GrB\_Type** of the return value of the binary operator being created. Should  
1275 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1276 BLAS type.

1277 **d\_in1** (IN) The **GrB\_Type** of the left hand argument of the binary operator being created.  
1278 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1279 GraphBLAS type.

1280 **d\_in2** (IN) The **GrB\_Type** of the right hand argument of the binary operator being cre-  
1281 ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-  
1282 defined GraphBLAS type.

## 1283 **Return Values**

1284 **GrB\_SUCCESS** operation completed successfully.

1285 **GrB\_PANIC** unknown internal error.

1286 **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1287 **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_Type** (for user-defined types) has not been initialized by a  
1288 call to **GrB\_Type\_new**.

1289 **GrB\_NULL\_POINTER** **binary\_op** or **binary\_func** pointer is **NULL**.

## 1290 **Description**

1291 The **BinaryOp\_new** methods creates a new GraphBLAS binary operator

1292  $f_b = \langle \mathbf{D}(d\_out), \mathbf{D}(d\_in1), \mathbf{D}(d\_in2), \text{binary\_func} \rangle$

1293 and returns a handle to it in **binary\_op**.

1294 The implementation of **binary\_func** must be such that it works even if any of the **d\_out**, **d\_in1**, and  
1295 **d\_in2** arguments are aliased to each other. In other words, for all invocations of the function:

1296 **binary\_func(out, in1, in2);**

1297 the value of **out** must be the same as if the following code was executed:

```
1298 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1299 D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1300 memcpy(tmp1, in1, sizeof(D(d_in1)));
1301 memcpy(tmp2, in2, sizeof(D(d_in2)));
1302 binary_func(out, tmp1, tmp2);
1303 free(tmp2);
1304 free(tmp1);
```

1305 It is not an error to call this method more than once on the same variable; however, the handle to  
1306 the previously created object will be overwritten.

#### 1307 4.2.2.4 Monoid\_new: Construct a new GraphBLAS monoid

1308 Creates a new monoid with specified binary operator and identity value.

#### 1309 C Syntax

```
1310         GrB_Info GrB_Monoid_new(GrB_Monoid    *monoid,  
1311                                GrB_BinaryOp    binary_op,  
1312                                <type>          identity);
```

#### 1313 Parameters

1314 monoid (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1315 monoid object.

1316 binary\_op (IN) An existing GraphBLAS associative binary operator whose input and output  
1317 types are the same.

1318 identity (IN) The value of the identity element of the monoid. Must be the same type as  
1319 the type used by the **binary\_op** operator.

#### 1320 Return Values

1321 GrB\_SUCCESS operation completed successfully.

1322 GrB\_PANIC unknown internal error.

1323 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1324 GrB\_UNINITIALIZED\_OBJECT the GrB\_BinaryOp (for user-defined operators) has not been initial-  
1325 ized by a call to GrB\_BinaryOp\_new.

1326 GrB\_NULL\_POINTER monoid pointer is NULL.

1327 GrB\_DOMAIN\_MISMATCH all three argument types of the binary operator and the type of the  
1328 identity value are not the same.

#### 1329 Description

1330 The Monoid\_new method creates a new monoid  $M = \langle \mathbf{D}(\text{binary\_op}), \text{binary\_op}, \text{identity} \rangle$  and re-  
1331 turns a handle to it in **monoid**.

1332 If `binary_op` is not associative, the results of GraphBLAS operations that require associativity of  
1333 this monoid will be undefined.

1334 It is not an error to call this method more than once on the same variable; however, the handle to  
1335 the previously created object will be overwritten.

#### 1336 4.2.2.5 Semiring\_new: Construct a new GraphBLAS semiring

1337 Creates a new semiring with specified domain, operators, and elements.

### 1338 C Syntax

```
1339         GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1340                                 GrB_Monoid    add_op,  
1341                                 GrB_BinaryOp   mul_op);
```

### 1342 Parameters

1343 `semiring` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1344 `semiring`.

1345 `add_op` (IN) An existing GraphBLAS commutative monoid that specifies the addition op-  
1346 erator and its identity.

1347 `mul_op` (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-  
1348 plication operator. In addition, `mul_op`'s output domain,  $\mathbf{D}_{out}(\text{mul\_op})$ , must be  
1349 the same as the `add_op`'s domain  $\mathbf{D}(\text{add\_op})$ .

### 1350 Return Values

1351 `GrB_SUCCESS` operation completed successfully.

1352 `GrB_PANIC` unknown internal error.

1353 `GrB_OUT_OF_MEMORY` not enough memory available for this method to complete.

1354 `GrB_UNINITIALIZED_OBJECT` the `add_op` (for user-define monoids) object has not been initialized  
1355 with a call to `GrB_Monoid_new` or the `mul_op` (for user-defined  
1356 operators) object has not been not been initialized by a call to  
1357 `GrB_BinaryOp_new`.

1358 `GrB_NULL_POINTER` `semiring` pointer is NULL.

1359 `GrB_DOMAIN_MISMATCH` the output domain of `mul_op` does not match the domain of the  
1360 `add_op` monoid.

## 1361 Description

1362 The `Semiring_new` method creates a new semiring:

$$1363 \quad S = \langle \mathbf{D}_{out}(\text{mul\_op}), \mathbf{D}_{in_1}(\text{mul\_op}), \mathbf{D}_{in_2}(\text{mul\_op}), \text{add\_op}, \text{mul\_op}, \mathbf{0}(\text{add\_op}) \rangle$$

1364 and returns a handle to it in `semiring`. Note that  $\mathbf{D}_{out}(\text{mul\_op})$  must be the same as  $\mathbf{D}(\text{add\_op})$ .

1365 If `add_op` is not commutative, then GraphBLAS operations using this semiring will be undefined.

1366 It is not an error to call this method more than once on the same variable; however, the handle to  
1367 the previously created object will be overwritten.

### 1368 4.2.2.6 IndexUnaryOp\_new: Construct a new GraphBLAS index unary operator [Scott: 1369 NEW CONTENT]

1370 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its  
1371 types (domains).

## 1372 C Syntax

```
1373 GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp *index_unary_op,  
1374                               void (*index_unary_func)(void*,  
1375                                                         const void*,  
1376                                                         GrB_Index,  
1377                                                         GrB_Index,  
1378                                                         const void*),  
1379                               GrB_Type d_out,  
1380                               GrB_Type d_in1,  
1381                               GrB_Type d_in2);
```

## 1382 Parameters

1383 `index_unary_op` (INOUT) On successful return, contains a handle to the newly created Graph-  
1384 BLAS index unary operator object.

1385 `index_unary_func` (IN) A pointer to a user-defined function that takes input parameters of types  
1386 `d_in1`, `GrB_Index`, `GrB_Index` and `d_in2` and returns a value of type `d_out`. Ex-  
1387 cept for the `GrB_Index` parameters, all are passed as void pointers. Specifically  
1388 the signature of the function is expected to be of the form:

```
1389 void func(void *out,  
1390           const void *in1,  
1391           GrB_Index row_index,  
1392           GrB_Index col_index,
```

1393 `const void *in2);`

1394

1395 **d\_out** (IN) The `GrB_Type` of the return value of the index unary operator being created.  
1396 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1397 GraphBLAS type.

1398 **d\_in1** (IN) The `GrB_Type` of the first input argument of the index unary operator being  
1399 created and corresponds to the stored values of the `GrB_Vector` or `GrB_Matrix`  
1400 being operated on. Should be one of the predefined GraphBLAS types in Ta-  
1401 ble 3.2, or a user-defined GraphBLAS type.

1402 **d\_in2** (IN) The `GrB_Type` of the last input argument of the index unary operator be-  
1403 ing created and corresponds to a scalar provided by the GraphBLAS operation  
1404 that uses this operator. Should be one of the predefined GraphBLAS types in  
1405 Table 3.2, or a user-defined GraphBLAS type.

## 1406 Return Values

1407 `GrB_SUCCESS` operation completed successfully.

1408 `GrB_PANIC` unknown internal error.

1409 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1410 `GrB_UNINITIALIZED_OBJECT` the `GrB_Type` (for user-defined types) has not been initialized by a  
1411 call to `GrB_Type_new`.

1412 `GrB_NULL_POINTER` `index_unary_op` or `index_unary_func` pointer is `NULL`.

## 1413 Description

1414 The `IndexUnaryOp_new` methods creates a new GraphBLAS index unary operator

1415 
$$f_i = \langle \mathbf{D}(d\_out), \mathbf{D}(d\_in1), \mathbf{D}(GrB\_Index), \mathbf{D}(GrB\_Index), \mathbf{D}(d\_in2), index\_unary\_func \rangle$$

1416 and returns a handle to it in `index_unary_op`.

1417 The implementation of `index_unary_func` must be such that it works even if any of the `d_out`,  
1418 `d_in1`, and `d_in2` arguments are aliased to each other. In other words, for all invocations of the  
1419 function:

1420 `index_unary_func(out, in1, row_index, col_index, n, in2);`

1421 the value of `out` must be the same as if the following code was executed (shown here for matrices):

```

1422     GrB_Index row_index = ...;
1423     GrB_Index col_index = ...;
1424     D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
1425     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1426     memcpy(tmp1,in1,sizeof(D(d_in1)));
1427     memcpy(tmp2,in2,sizeof(D(d_in2)));
1428     index_unary_func(out,tmp1,row_index,col_index,tmp2);
1429     free(tmp2);
1430     free(tmp1);

```

1431 It is not an error to call this method more than once on the same variable; however, the handle to  
1432 the previously created object will be overwritten.

### 1433 4.2.3 Scalar methods

#### 1434 4.2.3.1 Scalar\_new: Construct a new scalar

1435 Creates a new empty scalar with specified domain.

#### 1436 C Syntax

```

1437     GrB_Info GrB_Scalar_new(GrB_Scalar *s,
1438                             GrB_Type    d);

```

#### 1439 Parameters

1440 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1441 scalar.

1442 **d** (IN) The type corresponding to the domain of the scalar being created. Can be  
1443 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1444 GraphBLAS type.

#### 1445 Return Values

1446 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1447 blocking mode, this indicates that the API checks for the input  
1448 arguments passed successfully. Either way, output scalar **s** is ready  
1449 to be used in the next method of the sequence.

1450 **GrB\_PANIC** Unknown internal error.

1451 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1452 GraphBLAS objects (input or output) is in an invalid state caused

1453 by a previous execution error. Call `GrB_error()` to access any error  
1454 messages generated by the implementation.

1455 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1456 **GrB\_UNINITIALIZED\_OBJECT** The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`  
1457 (needed for user-defined types).

1458 **GrB\_NULL\_POINTER** The `s` pointer is `NULL`.

## 1459 Description

1460 Creates a new GraphBLAS scalar  $s$  of domain  $\mathbf{D}(d)$  and empty  $\mathbf{L}(s)$ . The method returns a handle  
1461 to the new scalar in `s`.

1462 It is not an error to call this method more than once on the same variable; however, the handle to  
1463 the previously created object will be overwritten.

### 1464 4.2.3.2 Scalar\_dup: Construct a copy of a GraphBLAS scalar

1465 Creates a new scalar with the same domain and contents as another scalar.

## 1466 C Syntax

```
1467 GrB_Info GrB_Scalar_dup(GrB_Scalar      *t,  
1468                          const GrB_Scalar s);
```

## 1469 Parameters

1470 `t` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1471 scalar.

1472 `s` (IN) The GraphBLAS scalar to be duplicated.

## 1473 Return Values

1474 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1475 blocking mode, this indicates that the API checks for the input  
1476 arguments passed successfully. Either way, output scalar `t` is ready  
1477 to be used in the next method of the sequence.

1478 **GrB\_PANIC** Unknown internal error.

1479 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1480 GraphBLAS objects (input or output) is in an invalid state caused

1481 by a previous execution error. Call `GrB_error()` to access any error  
1482 messages generated by the implementation.

1483 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1484 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS scalar, `s`, has not been initialized by a call to  
1485 `Scalar_new` or `Scalar_dup`.

1486 `GrB_NULL_POINTER` The `t` pointer is NULL.

## 1487 Description

1488 Creates a new scalar  $t$  of domain  $\mathbf{D}(\mathbf{s})$  and contents  $\mathbf{L}(\mathbf{s})$ . The method returns a handle to the new  
1489 scalar in `t`.

1490 It is not an error to call this method more than once with the same output variable; however, the  
1491 handle to the previously created object will be overwritten.

### 1492 4.2.3.3 `Scalar_clear`: Clear/remove a stored value from a scalar

1493 Removes the stored value from a scalar.

## 1494 C Syntax

1495 `GrB_Info GrB_Scalar_clear(GrB_Scalar s);`

## 1496 Parameters

1497 `s` (INOUT) An existing GraphBLAS scalar to clear.

## 1498 Return Values

1499 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
1500 blocking mode, this indicates that the API checks for the input  
1501 arguments passed successfully. Either way, output scalar `s` is ready  
1502 to be used in the next method of the sequence.

1503 `GrB_PANIC` Unknown internal error.

1504 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
1505 GraphBLAS objects (input or output) is in an invalid state caused  
1506 by a previous execution error. Call `GrB_error()` to access any error  
1507 messages generated by the implementation.

1508 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.



1509 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, `s`, has not been initialized by a call to  
1510 `Scalar_new` or `Scalar_dup`.

## 1511 Description

1512 Removes the stored value from an existing scalar. After the call, `L(s)` is empty. The size of the  
1513 scalar does not change.

## 1514 4.2.3.4 `Scalar_nvals`: Number of stored elements in a scalar

1515 Retrieve the number of stored elements in a scalar (either zero or one).

## 1516 C Syntax

```
1517         GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,  
1518                                   const GrB_Scalar s);
```

## 1519 Parameters

1520 `nvals` (OUT) On successful return, this is set to the number of stored elements in the  
1521 scalar (zero or one).

1522 `s` (IN) An existing GraphBLAS scalar being queried.

## 1523 Return Values

1524 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1525 cessfully and the value of `nvals` has been set.

1526 GrB\_PANIC Unknown internal error.

1527 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1528 GraphBLAS objects (input or output) is in an invalid state caused  
1529 by a previous execution error. Call `GrB_error()` to access any error  
1530 messages generated by the implementation.

1531 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1532 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, `s`, has not been initialized by a call to  
1533 `Scalar_new` or `Scalar_dup`.

1534 GrB\_NULL\_POINTER The `nvals` pointer is NULL.

1535 **Description**

1536 Return **nvals(s)** in **nvals**. This is the number of stored elements in scalar **s**, which is the size of  
1537 **L(s)**, and can only be either zero or one (see Section 3.5.1).

1538 **4.2.3.5 Scalar\_setElement: Set the single element in a scalar**

1539 Set the single element of a scalar to a given value.

1540 **C Syntax**

```
1541         GrB_Info GrB_Scalar_setElement(GrB_Scalar    s,  
1542                                         <type>      val);
```

1543 **Parameters**

1544 **s** (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1545 **val** (IN) Scalar value to assign. The type must be compatible with the domain of **s**.

1546 **Return Values**

1547 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1548 blocking mode, this indicates that the compatibility tests on in-  
1549 dex/dimensions and domains for the input arguments passed suc-  
1550 cessfully. Either way, the output scalar **s** is ready to be used in the  
1551 next method of the sequence.

1552 **GrB\_PANIC** Unknown internal error.

1553 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1554 GraphBLAS objects (input or output) is in an invalid state caused  
1555 by a previous execution error. Call **GrB\_error()** to access any error  
1556 messages generated by the implementation.

1557 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1558 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to  
1559 **Scalar\_new** or **Scalar\_dup**.

1560 **GrB\_DOMAIN\_MISMATCH** The domains of **s** and **val** are incompatible.

## 1561 Description

1562 First, **val** and output GraphBLAS scalar are tested for domain compatibility as follows: **D(val)** must  
1563 be compatible with **D(s)**. Two domains are compatible with each other if values from one domain  
1564 can be cast to values in the other domain as per the rules of the C language. In particular, domains  
1565 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-  
1566 patible with itself. If any compatibility rule above is violated, execution of **GrB\_Scalar\_setElement**  
1567 ends and the domain mismatch error listed above is returned.

1568 We are now ready to carry out the assignment **val**; that is:

$$1569 \quad s(0) = \text{val}$$

1570 If **s** already had a stored value, it will be overwritten; otherwise, the new value is stored in **s**.

1571 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new contents  
1572 of **s** is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method exits with  
1573 return value **GrB\_SUCCESS** and the new content of scalar **s** is as defined above but may not be  
1574 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1575 4.2.3.6 Scalar\_extractElement: Extract a single element from a scalar.

1576 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

## 1577 C Syntax

```
1578      GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1579                                         const GrB_Scalar s);
```

## 1580 Parameters

1581 **val** (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain  
1582 of scalar **s**. On successful return, **val** holds the result of the operation, and any  
1583 previous value in **val** is overwritten.

1584 **s** (IN) The GraphBLAS scalar from which an element is extracted.

## 1585 Return Values

1586 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1587 cessfully. This indicates that the compatibility tests on dimensions  
1588 and domains for the input arguments passed successfully, and the  
1589 output scalar, **val**, has been computed and is ready to be used in  
1590 the next method of the sequence.

1591 **GrB\_PANIC** Unknown internal error.



## 1622 Parameters

1623            **v** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1624            vector.

1625            **d** (IN) The type corresponding to the domain of the vector being created. Can be  
1626            one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1627            GraphBLAS type.

1628            **nsz** (IN) The size of the vector being created.

## 1629 Return Values

1630            **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1631            blocking mode, this indicates that the API checks for the input  
1632            arguments passed successfully. Either way, output vector **v** is ready  
1633            to be used in the next method of the sequence.

1634            **GrB\_PANIC** Unknown internal error.

1635            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1636            GraphBLAS objects (input or output) is in an invalid state caused  
1637            by a previous execution error. Call **GrB\_error()** to access any error  
1638            messages generated by the implementation.

1639            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1640 **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
1641            (needed for user-defined types).

1642            **GrB\_NULL\_POINTER** The **v** pointer is **NULL**.

1643            **GrB\_INVALID\_VALUE** **nsz** is zero or outside the range of the type **GrB\_Index**.

## 1644 Description

1645            Creates a new vector **v** of domain **D(d)**, size **nsz**, and empty **L(v)**. The method returns a handle  
1646            to the new vector in **v**.

1647            It is not an error to call this method more than once on the same variable; however, the handle to  
1648            the previously created object will be overwritten.

### 1649 4.2.4.2 Vector\_dup: Construct a copy of a GraphBLAS vector

1650            Creates a new vector with the same domain, size, and contents as another vector.

## 1651 C Syntax

```
1652         GrB_Info GrB_Vector_dup(GrB_Vector      *w,  
1653                                 const GrB_Vector  u);
```

## 1654 Parameters

1655 **w** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1656 vector.

1657 **u** (IN) The GraphBLAS vector to be duplicated.

## 1658 Return Values

1659 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1660 blocking mode, this indicates that the API checks for the input  
1661 arguments passed successfully. Either way, output vector **w** is ready  
1662 to be used in the next method of the sequence.

1663 **GrB\_PANIC** Unknown internal error.

1664 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1665 GraphBLAS objects (input or output) is in an invalid state caused  
1666 by a previous execution error. Call **GrB\_error()** to access any error  
1667 messages generated by the implementation.

1668 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1669 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **u**, has not been initialized by a call to  
1670 **Vector\_new** or **Vector\_dup**.

1671 **GrB\_NULL\_POINTER** The **w** pointer is **NULL**.

## 1672 Description

1673 Creates a new vector **w** of domain **D(u)**, size **size(u)**, and contents **L(u)**. The method returns a  
1674 handle to the new vector in **w**.

1675 It is not an error to call this method more than once on the same variable; however, the handle to  
1676 the previously created object will be overwritten.

### 1677 4.2.4.3 Vector\_resize: Resize a vector

1678 Changes the size of an existing vector.

## 1679 C Syntax

```
1680         GrB_Info GrB_Vector_resize(GrB_Vector w,  
1681                                   GrB_Index  nsize);
```

## 1682 Parameters

1683 **w** (INOUT) An existing Vector object that is being resized.

1684 **nsize** (IN) The new size of the vector. It can be smaller or larger than the current size.

## 1685 Return Values

1686 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1687 blocking mode, this indicates that the API checks for the input  
1688 arguments passed successfully. Either way, output vector **w** is ready  
1689 to be used in the next method of the sequence.

1690 **GrB\_PANIC** Unknown internal error.

1691 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1692 GraphBLAS objects (input or output) is in an invalid state caused  
1693 by a previous execution error. Call **GrB\_error()** to access any error  
1694 messages generated by the implementation.

1695 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1696 **GrB\_NULL\_POINTER** The **w** pointer is NULL.

1697 **GrB\_INVALID\_VALUE** **nsize** is zero or outside the range of the type **GrB\_Index**.

## 1698 Description

1699 Changes the size of **w** to **nsize**. The domain **D(w)** of vector **w** remains the same. The contents **L(w)**  
1700 are modified as described below.

1701 Let  $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$  when the method is called. When the method returns,  $w = \langle \mathbf{D}(w), \mathbf{nsize}, \mathbf{L}'(w) \rangle$   
1702 where  $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < \mathbf{nsize})\}$ . That is, all elements of **w** with index greater  
1703 than or equal to the new vector size (**nsize**) are dropped.

### 1704 4.2.4.4 Vector\_clear: Clear a vector

1705 Removes all the elements (tuples) from a vector.

## 1706 C Syntax

1707 `GrB_Info GrB_Vector_clear(GrB_Vector v);`

## 1708 Parameters

1709 `v` (INOUT) An existing GraphBLAS vector to clear.

## 1710 Return Values

1711 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
1712 blocking mode, this indicates that the API checks for the input  
1713 arguments passed successfully. Either way, output vector `v` is ready  
1714 to be used in the next method of the sequence.

1715 `GrB_PANIC` Unknown internal error.

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

1720 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

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

## 1723 Description

1724 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,  
1725  $L(v) = \emptyset$ . The size of the vector does not change.

### 1726 4.2.4.5 Vector\_size: Size of a vector

1727 Retrieve the size of a vector.

## 1728 C Syntax

1729 `GrB_Info GrB_Vector_size(GrB_Index *nsize,`  
1730 `const GrB_Vector v);`



## 1731 Parameters

1732            **nsz** (OUT) On successful return, is set to the size of the vector.

1733            **v** (IN) An existing GraphBLAS vector being queried.

## 1734 Return Values

1735            **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1736                            cessfully and the value of **nsz** has been set.

1737            **GrB\_PANIC** Unknown internal error.

1738            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1739                                    GraphBLAS objects (input or output) is in an invalid state caused  
1740                                    by a previous execution error. Call **GrB\_error()** to access any error  
1741                                    messages generated by the implementation.

1742            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to  
1743                                    **Vector\_new** or **Vector\_dup**.

1744            **GrB\_NULL\_POINTER** **nsz** pointer is NULL.

## 1745 Description

1746    Return **size(v)** in **nsz**.

### 1747 4.2.4.6 Vector\_nvals: Number of stored elements in a vector

1748    Retrieve the number of stored elements (tuples) in a vector.

## 1749 C Syntax

```
1750            GrB_Info GrB_Vector_nvals(GrB_Index            *nvals,  
1751                                        const GrB_Vector    v);
```

## 1752 Parameters

1753            **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)  
1754                            in the vector.

1755            **v** (IN) An existing GraphBLAS vector being queried.

## 1756 Return Values

1757                   GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1758                   cessfully and the value of `nvals` has been set.

1759                   GrB\_PANIC Unknown internal error.

1760                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1761                   GraphBLAS objects (input or output) is in an invalid state caused  
1762                   by a previous execution error. Call `GrB_error()` to access any error  
1763                   messages generated by the implementation.

1764                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1765 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to  
1766                   Vector\_new or Vector\_dup.

1767                   GrB\_NULL\_POINTER The `nvals` pointer is NULL.

## 1768 Description

1769 Return `nvals(v)` in `nvals`. This is the number of stored elements in vector `v`, which is the size of  
1770 `L(v)` (see Section 3.5.2).

### 1771 4.2.4.7 Vector\_build: Store elements from tuples into a vector

## 1772 C Syntax

```
1773                   GrB_Info GrB_Vector_build(GrB_Vector                   w,  
1774                                           const GrB_Index               *indices,  
1775                                           const <type>                *values,  
1776                                           GrB_Index                    n,  
1777                                           const GrB_BinaryOp           dup);
```

## 1778 Parameters

1779                   w (INOUT) An existing Vector object to store the result.

1780                   indices (IN) Pointer to an array of indices.

1781                   values (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
1782                   vector `w`.

1783                   n (IN) The number of entries contained in each array (the same for `indices` and `values`).

1784        **dup** (IN) An associative and commutative binary operator to apply when duplicate  
 1785        values for the same location are present in the input arrays. All three domains of  
 1786        **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
 1787        then duplicate locations will result in an error.

## 1788    **Return Values**

1789        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 1790        blocking mode, this indicates that the API checks for the input  
 1791        arguments passed successfully. Either way, output vector **w** is  
 1792        ready to be used in the next method of the sequence.

1793        **GrB\_PANIC** Unknown internal error.

1794        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 1795        opaque GraphBLAS objects (input or output) is in an invalid  
 1796        state caused by a previous execution error. Call **GrB\_error()** to  
 1797        access any error messages generated by the implementation.

1798        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1799        **GrB\_UNINITIALIZED\_OBJECT** Either **w** has not been initialized by a call to **GrB\_Vector\_new**  
 1800        or by **GrB\_Vector\_dup**, or **dup** has not been initialized by a call  
 1801        to **GrB\_BinaryOp\_new**.

1802        **GrB\_NULL\_POINTER** indices or values pointer is **NULL**.

1803        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in indices is outside the allowed range for **w**.

1804        **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
 1805        not all the same, or the domains of **values** and **w** are incompatible  
 1806        with each other or  $D_{dup}$ .

1807        **GrB\_OUTPUT\_NOT\_EMPTY** Output vector **w** already contains valid tuples (elements). In  
 1808        other words, **GrB\_Vector\_nvals(C)** returns a positive value.

1809        **GrB\_INVALID\_VALUE** indices contains a duplicate location and **dup** is **GrB\_NULL**.

## 1810    **Description**

1811        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  
 1812        from **w** in its domain; otherwise,  $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$ .

1813        Each tuple  $\{\text{indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the output in the form of

$$1814 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \mathbf{dup} \neq \mathbf{GrB\_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1815 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
 1816 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{w}}$  as follows:

$$1817 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1818 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{w}}$  is copied into `w` via typecasting its  
 1819 values to  $\mathbf{D}(\mathbf{w})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

1820 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1821 It is an error to call this function on an output object with existing elements. In other words,  
 1822 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1823 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`  
 1824 or `values`.

#### 1825 4.2.4.8 Vector\_setElement: Set a single element in a vector

1826 Set one element of a vector to a given value.

#### 1827 C Syntax

```
1828 // scalar value
1829 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1830                               <type>         val,
1831                               GrB_Index       index);
1832
1833 // GraphBLAS scalar
1834 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1835                               const GrB_Scalar s,
1836                               GrB_Index       index);
```

#### 1837 Parameters

1838 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1839 `val` or `s` (IN) Scalar assign. Its domain (type) must be compatible with the domain of `w`.

1840 `index` (IN) The location of the element to be assigned.

#### 1841 Return Values

1842 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 1843 blocking mode, this indicates that the compatibility tests on in-  
 1844 dex/dimensions and domains for the input arguments passed suc-

cessfully. Either way, the output vector  $w$  is ready to be used in the next method of the sequence.

**GrB\_PANIC** Unknown internal error.

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

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

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

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

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

## Description

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

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

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

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

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

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

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

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

#### 1878 4.2.4.9 Vector\_removeElement: Remove an element from a vector

1879 Remove (annihilate) one stored element from a vector.

#### 1880 C Syntax

```
1881         GrB_Info GrB_Vector_removeElement(GrB_Vector  w,  
1882                                           GrB_Index    index);
```

#### 1883 Parameters

1884 w (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1885 index (IN) The location of the element to be removed.

#### 1886 Return Values

1887 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1888 blocking mode, this indicates that the compatibility tests on in-  
1889 dex/dimensions and domains for the input arguments passed suc-  
1890 cessfully. Either way, the output vector w is ready to be used in  
1891 the next method of the sequence.

1892 GrB\_PANIC Unknown internal error.

1893 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1894 GraphBLAS objects (input or output) is in an invalid state caused  
1895 by a previous execution error. Call GrB\_error() to access any error  
1896 messages generated by the implementation.

1897 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1898 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, w, has not been initialized by a call to  
1899 Vector\_new or Vector\_dup.

1900 GrB\_INVALID\_INDEX index specifies a location that is outside the dimensions of w.

#### 1901 Description

1902 First, the index parameter is checked for a valid value where the following condition must hold:

$$1903 \quad 0 \leq \text{index} < \text{size}(w)$$

1904 If this condition is violated, execution of GrB\_Vector\_removeElement ends and the invalid index  
1905 error listed above is returned.

1906 We are now ready to carry out the removal of a value that may be stored at the location specified  
 1907 by `index`. If a value does not exist at the specified location in `w`, no error is reported and the  
 1908 operation has no effect on the state of `w`. In either case, the following will be true on return from  
 1909 the method: `index`  $\notin$  `ind(w)`.

1910 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
 1911 of `w` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
 1912 return value `GrB_SUCCESS` and the new content of vector `w` is as defined above but may not be  
 1913 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1914 4.2.4.10 `Vector_extractElement`: Extract a single element from a vector.

1915 Extract one element of a vector into a scalar.

#### 1916 C Syntax

```
1917 // scalar value
1918 GrB_Info GrB_Vector_extractElement(<type>          *val,
1919                                   const GrB_Vector u,
1920                                   GrB_Index         index);
1921
1922 // GraphBLAS scalar
1923 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1924                                   const GrB_Vector u,
1925                                   GrB_Index         index);
```

#### 1926 Parameters

1927 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector  
 1928 `u`. On successful return, this scalar holds the result of the extract. Any previous  
 1929 value stored in `val` or `s` is overwritten.

1930 `u` (IN) The GraphBLAS vector from which an element is extracted.

1931 `index` (IN) The location in `u` to extract.

#### 1932 Return Values

1933 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-  
 1934 cessfully. This indicates that the compatibility tests on dimensions  
 1935 and domains for the input arguments passed successfully, and the  
 1936 output scalar, `val` or `s`, has been computed and is ready to be used  
 1937 in the next method of the sequence.

1938                   GrB\_NO\_VALUE When using the transparent scalar, `val`, this is returned when there  
1939   is no stored value at specified location.

1940                   GrB\_PANIC Unknown internal error.

1941                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1942   GraphBLAS objects (input or output) is in an invalid state caused  
1943   by a previous execution error. Call `GrB_error()` to access any error  
1944   messages generated by the implementation.

1945                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1946 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, `u`, or scalar, `s`, has not been initialized by  
1947   a call to a corresponding constructor.

1948                   GrB\_NULL\_POINTER `val` pointer is NULL.

1949                   GrB\_INVALID\_INDEX `index` specifies a location that is outside the dimensions of `w`.

1950                   GrB\_DOMAIN\_MISMATCH The domains of the vector and scalar are incompatible.

## 1951 Description

1952 First, the scalar and input vector are tested for domain compatibility as follows:  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\mathbf{s})$   
1953 must be compatible with  $\mathbf{D}(\mathbf{u})$ . Two domains are compatible with each other if values from  
1954 one domain can be cast to values in the other domain as per the rules of the C language. In  
1955 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1956 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1957 `GrB_Vector_extractElement` ends and the domain mismatch error listed above is returned.

1958 Then, the `index` parameter is checked for a valid value where the following condition must hold:

$$1959 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1960 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the invalid index  
1961 error listed above is returned.

1962 We are now ready to carry out the extract into the output scalar; that is:

$$1963 \qquad \left. \begin{array}{l} \mathbf{L}(\mathbf{s}) \\ \text{val} \end{array} \right\} = \mathbf{u}(\text{index})$$

1964 If  $\text{index} \in \mathbf{ind}(\mathbf{u})$ , then the corresponding value from `u` is copied into `s` or `val` with casting as  
1965 necessary. If  $\text{index} \notin \mathbf{ind}(\mathbf{u})$ , then one of the follow occurs depending on output scalar type:

- 1966       • The GraphBLAS scalar, `s`, is cleared and `GrB_SUCCESS` is returned.
- 1967       • The non-opaque scalar, `val`, is unchanged, and `GrB_NO_VALUE` is returned.



1968 When using the non-opaque scalar variant (`val`) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 1969 mode, the new contents of `val` are as defined above if the method exits with return value `GrB_SUCCESS`  
 1970 or `GrB_NO_VALUE`.

1971 When using the GraphBLAS scalar variant (`s`) with a `GrB_SUCCESS` return value, the method  
 1972 exits and the new contents of `s` is as defined above and fully computed in `GrB_BLOCKING` mode.  
 1973 In `GrB_NONBLOCKING` mode, the new contents of `s` is as defined above but may not be fully  
 1974 computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1975 **4.2.4.11 Vector\_extractTuples: Extract tuples from a vector**

1976 Extract the contents of a GraphBLAS vector into non-opaque data structures.

### 1977 **C Syntax**

```
1978      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,
1979                                     <type>          *values,
1980                                     GrB_Index        *n,
1981                                     const GrB_Vector v);
1982
```

1983 `indices` (OUT) Pointer to an array of indices that is large enough to hold all of the stored  
 1984 values' indices.

1985 `values` (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
 1986 the stored values whose type is compatible with `D(v)`.

1987 `n` (INOUT) Pointer to a value indicating (on input) the number of elements the  
 1988 `values` and `indices` arrays can hold. Upon return, it will contain the number of  
 1989 values written to the arrays.

1990 `v` (IN) An existing GraphBLAS vector.

### 1991 **Return Values**

1992 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-  
 1993 cessfully. This indicates that the compatibility tests on the input  
 1994 argument passed successfully, and the output arrays, `indices` and  
 1995 `values`, have been computed.

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.



```

2030         GrB_Index    nrows,
2031         GrB_Index    ncols);

```

## 2032 Parameters

2033        **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2034        matrix.

2035        **d** (IN) The type corresponding to the domain of the matrix being created. Can be  
2036        one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
2037        GraphBLAS type.

2038        **nrows** (IN) The number of rows of the matrix being created.

2039        **ncols** (IN) The number of columns of the matrix being created.

## 2040 Return Values

2041        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2042        blocking mode, this indicates that the API checks for the input ar-  
2043        guments passed successfully. Either way, output matrix **A** is ready  
2044        to be used in the next method of the sequence.

2045        **GrB\_PANIC** Unknown internal error.

2046        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2047        GraphBLAS objects (input or output) is in an invalid state caused  
2048        by a previous execution error. Call **GrB\_error()** to access any error  
2049        messages generated by the implementation.

2050        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2051        **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2052        (needed for user-defined types).

2053        **GrB\_NULL\_POINTER** The **A** pointer is **NULL**.

2054        **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

## 2055 Description

2056        Creates a new matrix **A** of domain **D(d)**, size **nrows**  $\times$  **ncols**, and empty **L(A)**. The method returns  
2057        a handle to the new matrix in **A**.

2058        It is not an error to call this method more than once on the same variable; however, the handle to  
2059        the previously created object will be overwritten.

#### 2060 4.2.5.2 Matrix\_dup: Construct a copy of a GraphBLAS matrix

2061 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

#### 2062 C Syntax

```
2063         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
2064                                const GrB_Matrix A);
```

#### 2065 Parameters

2066 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2067 matrix.

2068 A (IN) The GraphBLAS matrix to be duplicated.

#### 2069 Return Values

2070 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2071 blocking mode, this indicates that the API checks for the input  
2072 arguments passed successfully. Either way, output matrix C is ready  
2073 to be used in the next method of the sequence.

2074 GrB\_PANIC Unknown internal error.

2075 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2076 GraphBLAS objects (input or output) is in an invalid state caused  
2077 by a previous execution error. Call GrB\_error() to access any error  
2078 messages generated by the implementation.

2079 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2080 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2081 any matrix constructor.

2082 GrB\_NULL\_POINTER The C pointer is NULL.

#### 2083 Description

2084 Creates a new matrix **C** of domain **D(A)**, size **nrows(A) × ncols(A)**, and contents **L(A)**. It returns  
2085 a handle to it in C.

2086 It is not an error to call this method more than once on the same variable; however, the handle to  
2087 the previously created object will be overwritten.

### 2088 4.2.5.3 Matrix\_diag: Construct a diagonal GraphBLAS matrix

2089 Creates a new matrix with the same domain and contents as a GrB\_Vector, and square dimensions  
2090 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

#### 2091 C Syntax

```
2092         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
2093                                 const GrB_Vector  v,  
2094                                 int64_t           k);
```

#### 2095 Parameters

2096 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2097 matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

2098 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the  
2099 matrix.

2100 k (IN) The diagonal to which the vector is assigned. k = 0 represents the main  
2101 diagonal, k > 0 is above the main diagonal, and k < 0 is below.

#### 2102 Return Values

2103 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2104 blocking mode, this indicates that the API checks for the input  
2105 arguments passed successfully. Either way, output matrix C is ready  
2106 to be used in the next method of the sequence.

2107 GrB\_PANIC Unknown internal error.

2108 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2109 GraphBLAS objects (input or output) is in an invalid state caused  
2110 by a previous execution error. Call GrB\_error() to access any error  
2111 messages generated by the implementation.

2112 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

2113 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, v, has not been initialized by a call to  
2114 Vector\_new or Vector\_dup.

2115 GrB\_NULL\_POINTER The C pointer is NULL.

## 2116 Description

2117 Creates a new matrix **C** of domain **D(v)**, size  $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$ , and contents

$$2118 \quad \mathbf{L}(\mathbf{C}) = \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or}$$

$$2119 \quad \mathbf{L}(\mathbf{C}) = \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0.$$

2120 It returns a handle to it in **C**. It is not an error to call this method more than once on the same  
2121 variable; however, the handle to the previously created object will be overwritten.

## 2122 4.2.5.4 Matrix\_resize: Resize a matrix

2123 Changes the dimensions of an existing matrix.

## 2124 C Syntax

```
2125      GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
2126                               GrB_Index  nrows,  
2127                               GrB_Index  ncols);
```

## 2128 Parameters

2129 **C** (INOUT) An existing Matrix object that is being resized.

2130 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the  
2131 current number of rows.

2132 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than  
2133 the current number of columns.

## 2134 Return Values

2135 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2136 blocking mode, this indicates that the API checks for the input  
2137 arguments passed successfully. Either way, output matrix **C** is ready  
2138 to be used in the next method of the sequence.

2139 **GrB\_PANIC** Unknown internal error.

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

2144 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2145           GrB\_NULL\_POINTER The C pointer is NULL.

2146           GrB\_INVALID\_VALUE nrows or ncols is zero or outside the range of the type GrB\_Index.

## 2147   Description

2148   Changes the number of rows and columns of C to nrows and ncols, respectively. The domain  $\mathbf{D}(\mathbf{C})$   
2149   of matrix C remains the same. The contents  $\mathbf{L}(\mathbf{C})$  are modified as described below.

2150   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  
2151   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  
2152   greater than or equal to ncols are dropped.  
2153

### 2154   4.2.5.5   Matrix\_clear: Clear a matrix

2155   Removes all elements (tuples) from a matrix.

## 2156   C Syntax

2157           GrB\_Info GrB\_Matrix\_clear(GrB\_Matrix A);

## 2158   Parameters

2159           A (IN) An existing GraphBLAS matrix to clear.

## 2160   Return Values

2161           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2162                       blocking mode, this indicates that the API checks for the input ar-  
2163                       guments passed successfully. Either way, output matrix A is ready  
2164                       to be used in the next method of the sequence.

2165           GrB\_PANIC Unknown internal error.

2166           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2167                       GraphBLAS objects (input or output) is in an invalid state caused  
2168                       by a previous execution error. Call GrB\_error() to access any error  
2169                       messages generated by the implementation.

2170           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2171           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2172                       any matrix constructor.

2173 **Description**

2174 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,  
2175  $L(A) = \emptyset$ . The dimensions of the matrix do not change.

2176 **4.2.5.6 Matrix\_nrows: Number of rows in a matrix**

2177 Retrieve the number of rows in a matrix.

2178 **C Syntax**

```
2179         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2180                                   const GrB_Matrix A);
```

2181 **Parameters**

2182 nrows (OUT) On successful return, contains the number of rows in the matrix.

2183 A (IN) An existing GraphBLAS matrix being queried.

2184 **Return Values**

2185 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2186 cessfully and the value of `nrows` has been set.

2187 GrB\_PANIC Unknown internal error.

2188 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2189 GraphBLAS objects (input or output) is in an invalid state caused  
2190 by a previous execution error. Call `GrB_error()` to access any error  
2191 messages generated by the implementation.

2192 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2193 any matrix constructor.

2194 GrB\_NULL\_POINTER `nrows` pointer is NULL.

2195 **Description**

2196 Return `nrows(A)` in `nrows` (the number of rows).

2197 **4.2.5.7 Matrix\_ncols: Number of columns in a matrix**

2198 Retrieve the number of columns in a matrix.



## 2199 C Syntax

```
2200         GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2201                                   const GrB_Matrix A);
```

## 2202 Parameters

2203 ncols (OUT) On successful return, contains the number of columns in the matrix.

2204 A (IN) An existing GraphBLAS matrix being queried.

## 2205 Return Values

2206 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2207 cessfully and the value of ncols has been set.

2208 GrB\_PANIC Unknown internal error.

2209 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2210 GraphBLAS objects (input or output) is in an invalid state caused  
2211 by a previous execution error. Call GrB\_error() to access any error  
2212 messages generated by the implementation.

2213 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2214 any matrix constructor.

2215 GrB\_NULL\_POINTER ncols pointer is NULL.

## 2216 Description

2217 Return **ncols(A)** in ncols (the number of columns).

## 2218 4.2.5.8 Matrix\_nvals: Number of stored elements in a matrix

2219 Retrieve the number of stored elements (tuples) in a matrix.

## 2220 C Syntax

```
2221         GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2222                                   const GrB_Matrix A);
```

2223 **Parameters**

2224            **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in  
2225            the matrix.

2226            **A** (IN) An existing GraphBLAS matrix being queried.

2227 **Return Values**

2228            **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2229            cessfully and the value of **nvals** has been set.

2230            **GrB\_PANIC** Unknown internal error.

2231            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2232            GraphBLAS objects (input or output) is in an invalid state caused  
2233            by a previous execution error. Call **GrB\_error()** to access any error  
2234            messages generated by the implementation.

2235            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2236            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2237            any matrix constructor.

2238            **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

2239 **Description**

2240 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of  
2241 **L(A)** (see Section 3.5.3).

2242 **4.2.5.9 Matrix\_build: Store elements from tuples into a matrix**

2243 **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);
```

2244 **Parameters**

2245            **C** (INOUT) An existing Matrix object to store the result.

2246 **row\_indices** (IN) Pointer to an array of row indices.

2247 **col\_indices** (IN) Pointer to an array of column indices.

2248 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
2249 matrix, **C**.

2250 **n** (IN) The number of entries contained in each array (the same for **row\_indices**,  
2251 **col\_indices**, and **values**).

2252 **dup** (IN) An associative and commutative binary operator to apply when duplicate  
2253 values for the same location are present in the input arrays. All three domains of  
2254 **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
2255 then duplicate locations will result in an error.

## 2256 Return Values

2257 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2258 blocking mode, this indicates that the API checks for the input  
2259 arguments passed successfully. Either way, output matrix **C** is  
2260 ready to be used in the next method of the sequence.

2261 **GrB\_PANIC** Unknown internal error.

2262 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2263 opaque GraphBLAS objects (input or output) is in an invalid  
2264 state caused by a previous execution error. Call **GrB\_error()** to  
2265 access any error messages generated by the implementation.

2266 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2267 **GrB\_UNINITIALIZED\_OBJECT** Either **C** has not been initialized by a call to any matrix construc-  
2268 tor, or **dup** has not been initialized by a call to **GrB\_BinaryOp\_new**.

2269 **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices** or **values** pointer is **NULL**.

2270 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** or **col\_indices** is outside the allowed range  
2271 for **C**.

2272 **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
2273 not all the same, or the domains of **values** and **C** are incompatible  
2274 with each other or  $D_{dup}$ .

2275 **GrB\_OUTPUT\_NOT\_EMPTY** Output matrix **C** already contains valid tuples (elements). In  
2276 other words, **GrB\_Matrix\_nvals(C)** returns a positive value.

2277 **GrB\_INVALID\_VALUE** indices contains a duplicate location and **dup** is **GrB\_NULL**.

## 2278 Description

2279 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  
 2280 only differs from  $\mathbf{C}$  in its domain; otherwise,  $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$ .

2281 Each tuple  $\{\text{row\_indices}[k], \text{col\_indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the  
 2282 output in the form of

$$2283 \quad \tilde{\mathbf{C}}(\text{row\_indices}[k], \text{col\_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2284 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
 2285 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{C}}$  as follows:

$$2286 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row\_indices}[k]=i \wedge \text{col\_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2287 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{C}}$  is copied into  $\mathbf{C}$  via typecasting its  
 2288 values to  $\mathbf{D}(\mathbf{C})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

2289 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2290 It is an error to call this function on an output object with existing elements. In other words,  
 2291 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2292 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,  
 2293 `col_indices`, or `values`.

### 2294 4.2.5.10 Matrix\_setElement: Set a single element in matrix

2295 Set one element of a matrix to a given value.

## 2296 C Syntax

```
2297 // scalar value
2298 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2299                               <type>         val,
2300                               GrB_Index        row_index,
2301                               GrB_Index        col_index);
2302
2303 // GraphBLAS scalar
2304 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2305                               const GrB_Scalar s,
2306                               GrB_Index        row_index,
2307                               GrB_Index        col_index);
```

## 2308 Parameters

2309           **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.  
2310           **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of  
2311           **C**.  
2312           **row\_index** (IN) Row index of element to be assigned  
2313           **col\_index** (IN) Column index of element to be assigned

## 2314 Return Values

2315           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2316           blocking mode, this indicates that the compatibility tests on in-  
2317           dex/dimensions and domains for the input arguments passed suc-  
2318           cessfully. Either way, the output matrix **C** is ready to be used in  
2319           the next method of the sequence.  
2320           **GrB\_PANIC** Unknown internal error.  
2321           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2322           GraphBLAS objects (input or output) is in an invalid state caused  
2323           by a previous execution error. Call **GrB\_error()** to access any error  
2324           messages generated by the implementation.  
2325           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.  
2326           **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been  
2327           initialized by a call to a respective constructor.  
2328           **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e., not less  
2329           than **nrows(C)** or **ncols(C)**, respectively).  
2330           **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and the scalar are incompatible.

## 2331 Description

2332 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or  
2333 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from  
2334 one domain can be cast to values in the other domain as per the rules of the C language. In  
2335 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
2336 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
2337 **GrB\_Matrix\_setElement** ends and the domain mismatch error listed above is returned.

2338 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2339 \quad & 0 \leq \text{row\_index} < \text{nrows}(\text{C}), \\ & 0 \leq \text{col\_index} < \text{ncols}(\text{C}) \end{aligned}$$

2340 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid  
 2341 index error listed above is returned.

2342 We are now ready to carry out the assignment; that is:

$$2343 \quad C(\text{row\_index}, \text{col\_index}) = \begin{cases} \mathbf{L}(\mathbf{s}), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

2344 In the case of a transparent scalar or if  $\mathbf{L}(\mathbf{s})$  is not empty, then a value will be stored at the  
 2345 specified location in  $\mathbf{C}$ , overwriting any value that may have been stored there before. In the case  
 2346 of a GraphBLAS scalar and if  $\mathbf{L}(\mathbf{s})$  is empty, then any value stored at the specified location in  $\mathbf{C}$   
 2347 will be removed.

2348 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
 2349 of  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
 2350 return value `GrB_SUCCESS` and the new content of vector  $\mathbf{C}$  is as defined above but may not be  
 2351 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 2352 **4.2.5.11 Matrix\_removeElement: Remove an element from a matrix**

2353 Remove (annihilate) one stored element from a matrix.

#### 2354 **C Syntax**

```
2355      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2356                                       GrB_Index   row_index,
2357                                       GrB_Index   col_index);
```

#### 2358 **Parameters**

2359 `C` (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2360 `row_index` (IN) Row index of element to be removed

2361 `col_index` (IN) Column index of element to be removed

#### 2362 **Return Values**

2363 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 2364 blocking mode, this indicates that the compatibility tests on in-  
 2365 dex/dimensions and domains for the input arguments passed suc-  
 2366 cessfully. Either way, the output matrix  $\mathbf{C}$  is ready to be used in  
 2367 the next method of the sequence.

2368 `GrB_PANIC` Unknown internal error.

2369       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 2370       GraphBLAS objects (input or output) is in an invalid state caused  
 2371       by a previous execution error. Call GrB\_error() to access any error  
 2372       messages generated by the implementation.

2373       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2374       GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to  
 2375       any matrix constructor.

2376       GrB\_INVALID\_INDEX row\_index or col\_index is outside the allowable range (i.e., not less  
 2377       than **nrows(C)** or **ncols(C)**, respectively).

## 2378 Description

2379 First, both index parameters are checked for valid values where following conditions must hold:

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

2381 If either of these conditions is violated, execution of GrB\_Matrix\_removeElement ends and the  
 2382 invalid index error listed above is returned.

2383 We are now ready to carry out the removal of a value that may be stored at the location specified by  
 2384 (row\_index, col\_index). If a value does not exist at the specified location in C, no error is reported  
 2385 and the operation has no effect on the state of C. In either case, the following will be true on return  
 2386 from this method: (row\_index, col\_index)  $\notin$  ind(C)

2387 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 2388 of C is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 2389 return value GrB\_SUCCESS and the new content of vector C is as defined above but may not be  
 2390 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2391 4.2.5.12 Matrix\_extractElement: Extract a single element from a matrix

2392 Extract one element of a matrix into a scalar.

## 2393 C Syntax

```

2394     // scalar value
2395     GrB_Info GrB_Matrix_extractElement(<type>          *val,
2396                                     const GrB_Matrix  A,
2397                                     GrB_Index          row_index,
2398                                     GrB_Index          col_index);
2399
2400     // GraphBLAS scalar
  
```

```

2401         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2402                                             const GrB_Matrix A,
2403                                             GrB_Index      row_index,
2404                                             GrB_Index      col_index);
2405

```

## 2406 Parameters

2407     **val or s (INOUT)** An existing scalar whose domain is compatible with the domain of matrix  
2408     **A.** On successful return, this scalar holds the result of the extract. Any previous  
2409     value stored in **val** or **s** is overwritten.

2410     **A (IN)** The GraphBLAS matrix from which an element is extracted.

2411     **row\_index (IN)** The row index of location in **A** to extract.

2412     **col\_index (IN)** The column index of location in **A** to extract.

## 2413 Return Values

2414     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2415     cessfully. This indicates that the compatibility tests on dimensions  
2416     and domains for the input arguments passed successfully, and the  
2417     output scalar, **val** or **s**, has been computed and is ready to be used  
2418     in the next method of the sequence.

2419     **GrB\_NO\_VALUE** When using the transparent scalar, **val**, this is returned when there  
2420     is no stored value at specified location.

2421     **GrB\_PANIC** Unknown internal error.

2422     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2423     GraphBLAS objects (input or output) is in an invalid state caused  
2424     by a previous execution error. Call **GrB\_error()** to access any error  
2425     messages generated by the implementation.

2426     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2427     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by  
2428     a call to a corresponding constructor.

2429     **GrB\_NULL\_POINTER** **val** pointer is **NULL**.

2430     **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e. less than  
2431     zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-  
2432     tively).

2433     **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and scalar are incompatible.



## 2434 Description

2435 First, the scalar and input matrix are tested for domain compatibility as follows:  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\mathbf{s})$   
 2436 must be compatible with  $\mathbf{D}(\mathbf{A})$ . Two domains are compatible with each other if values from  
 2437 one domain can be cast to values in the other domain as per the rules of the C language. In  
 2438 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 2439 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 2440 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2441 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2442 \quad & 0 \leq \text{row\_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2443 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index  
 2444 error listed above is returned.

2445 We are now ready to carry out the extract into the output scalar; that is,

$$2446 \quad \left. \begin{array}{l} \mathbf{L}(\mathbf{s}) \\ \text{val} \end{array} \right\} = \mathbf{A}(\text{row\_index}, \text{col\_index})$$

2447 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  $\text{val}$   
 2448 with casting as necessary. If  $(\text{row\_index}, \text{col\_index}) \notin \mathbf{ind}(\mathbf{A})$ , then one of the follow occurs  
 2449 depending on output scalar type:

- 2450 • The GraphBLAS scalar,  $\mathbf{s}$ , is cleared and `GrB_SUCCESS` is returned.
- 2451 • The non-opaque scalar,  $\text{val}$ , is unchanged, and `GrB_NO_VALUE` is returned.

2452 When using the non-opaque scalar variant ( $\text{val}$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 2453 mode, the new contents of  $\text{val}$  are as defined above if the method exits with return value `GrB_SUCCESS`  
 2454 or `GrB_NO_VALUE`.

2455 When using the GraphBLAS scalar variant ( $\mathbf{s}$ ) with a `GrB_SUCCESS` return value, the method  
 2456 exits and the new contents of  $\mathbf{s}$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
 2457 In `GrB_NONBLOCKING` mode, the new contents of  $\mathbf{s}$  is as defined above but may not be fully  
 2458 computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2459 4.2.5.13 Matrix\_extractTuples: Extract tuples from a matrix

2460 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

## 2461 C Syntax

```
2462      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2463                                       GrB_Index      *col_indices,
```

```

2464                                     <type>          *values,
2465                                     GrB_Index        *n,
2466                                     const GrB_Matrix  A);

```

## 2467 Parameters

2468     **row\_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the  
2469     row indices.

2470     **col\_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the  
2471     column indices.

2472     **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
2473     the stored values whose type is compatible with  $\mathbf{D}(\mathbf{A})$ .

2474     **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,  
2475     **row\_indices**, and **col\_indices** arrays can hold. Upon return, it will contain the  
2476     number of values written to the arrays.

2477     **A** (IN) An existing GraphBLAS matrix.

## 2478 Return Values

2479     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2480     cessfully. This indicates that the compatibility tests on the input  
2481     argument passed successfully, and the output arrays, **indices** and  
2482     **values**, have been computed.

2483     **GrB\_PANIC** Unknown internal error.

2484     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2485     GraphBLAS objects (input or output) is in an invalid state caused  
2486     by a previous execution error. Call **GrB\_error()** to access any error  
2487     messages generated by the implementation.

2488     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2489     **GrB\_INSUFFICIENT\_SPACE** Not enough space in **row\_indices**, **col\_indices**, and **values** (as indi-  
2490     cated by the **n** parameter) to hold all of the tuples that will be  
2491     extracted.

2492     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2493     any matrix constructor.

2494     **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices**, **values** or **n** pointer is NULL.

2495     **GrB\_DOMAIN\_MISMATCH** The domains of the **A** matrix and **values** array are incompatible  
2496     with one another.

## 2497 Description

2498 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with  
2499 those tuples are placed in the **values** array, the column indices are placed in the **col\_indices** array,  
2500 and the row indices are placed in the **row\_indices** array. These output arrays are pre-allocated by  
2501 the user before calling this function such that each output array has enough space to hold at least  
2502 **GrB\_Matrix\_nvals(A)** elements.

2503 Upon return of this function, a pair of  $\{\text{row\_indices}[k], \text{col\_indices}[k]\}$  are unique for every valid  
2504  $k$ , but they are not required to be sorted in any particular order. Each tuple  $(i, j, A_{ij})$  in **A** is  
2505 unzipped and copied into a distinct  $k$ th location in output vectors:

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

2506 where  $0 \leq k < \text{GrB\_Matrix\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if **row\_indices**[ $k$ ],  
2507 **col\_indices**[ $k$ ] and **values**[ $k$ ] exist upon return, so does **row\_indices**[ $j$ ], **col\_indices**[ $j$ ] and **values**[ $j$ ] for  
2508 all  $j$  such that  $0 \leq j < k$ .

2509 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,  
2510 then a **GrB\_INSUFFICIENT\_SPACE** error is returned since it is undefined which subset of values  
2511 would be extracted.

2512 In both **GrB\_BLOCKING** mode **GrB\_NONBLOCKING** mode if the method exits with return value  
2513 **GrB\_SUCCESS**, the new contents of the arrays **row\_indices**, **col\_indices** and **values** are as defined  
2514 above.

2515 **4.2.5.14 Matrix\_exportHint: Provide a hint as to which storage format might be most**  
2516 **efficient for exporting a matrix**

## 2517 C Syntax

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

## 2518 Parameters

2519 **hint** (OUT) Pointer to a value of type **GrB\_Format**.

2520 **A** (IN) A GraphBLAS matrix object.

## 2521 Return Values

2522 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2523 cessfully and the value of **hint** has been set.

2524 **GrB\_PANIC** Unknown internal error.

2525           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 2526                                   opaque GraphBLAS objects (input or output) is in an invalid  
 2527                                   state caused by a previous execution error. Call GrB\_error() to  
 2528                                   access any error messages generated by the implementation.

2529           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2530           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
 2531                                   any matrix constructor.

2532           GrB\_NULL\_POINTER hint is NULL.

2533           GrB\_NO\_VALUE If the implementation does not have a preferred format, it may  
 2534                                   return the value GrB\_NO\_VALUE.

## 2535 Description

2536 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for  
 2537 exporting the matrix A. GraphBLAS implementations might return the current storage format of  
 2538 the matrix, or the format to which it could most efficiently be exported. However, implementations  
 2539 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is  
 2540 free to refuse to provide a format hint, returning GrB\_NO\_VALUE.

### 2541 4.2.5.15 Matrix\_exportSize: Return the array sizes necessary to export a GraphBLAS 2542 matrix object

## 2543 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);

```

## 2544 Parameters

2545           n\_indptr (OUT) Pointer to a value of type GrB\_Index.

2546           n\_indices (OUT) Pointer to a value of type GrB\_Index.

2547           n\_values (OUT) Pointer to a value of type GrB\_Index.

2548           format (IN) a value indicating the format in which the matrix will be exported, as defined  
 2549                                   in Section 3.5.3.1.

2550           A (IN) A GraphBLAS matrix object.

## 2551 Return Values

2552                   GrB\_SUCCESS In blocking mode or non-blocking mode, the operation com-  
2553                   pleted successfully. This indicates that the API checks for the  
2554                   input arguments passed successfully, and the number of elements  
2555                   necessary for the export buffers have been written to `n_indptr`,  
2556                   `n_indices`, and `n_values`, respectively.

2557                   GrB\_PANIC Unknown internal error.

2558                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2559                   opaque GraphBLAS objects (input or output) is in an invalid  
2560                   state caused by a previous execution error. Call `GrB_error()` to  
2561                   access any error messages generated by the implementation.

2562                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2563                   GrB\_UNINITIALIZED\_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to  
2564                   any matrix constructor.

2565                   GrB\_NULL\_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

## 2566 Description

2567 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to  
2568 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and  
2569 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be  
2570 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user  
2571 as one of the values defined in Section 3.5.3.1.

## 2572 4.2.5.16 Matrix\_export: Export a GraphBLAS matrix to a pre-defined format

### 2573 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);
```

## 2574 Parameters

2575        **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-  
2576        dices, depending on the value of **format**. It must be large enough to hold at  
2577        least **n\_indptr** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2578        **GrB\_Matrix\_exportSize()** method.

2579        **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements  
2580        in **values**, depending on the value of **format**. It must be large enough to hold at  
2581        least **n\_indices** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2582        **GrB\_Matrix\_exportSize()** method.

2583        **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-  
2584        ment must match the type of the values stored in **A**. It must be large enough  
2585        to hold at least **n\_values** elements of that type, where **n\_values** was returned from  
2586        **GrB\_Matrix\_exportSize**.

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

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

2593        **n\_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**  
2594        array can hold. Upon return, it will contain the number of elements written to the  
2595        array.

2596        **format** (IN) a value indicating the format in which the matrix will be exported, as defined  
2597        in Section 3.5.3.1.

2598        **A** (IN) A GraphBLAS matrix object.

## 2599 Return Values

2600        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2601        cessfully. This indicates that the compatibility tests on the input  
2602        argument passed successfully, and the output arrays, **indptr**, **in-**  
2603        **dices** and **values**, have been computed.

2604        **GrB\_PANIC** Unknown internal error.

2605        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2606        opaque GraphBLAS objects (input or output) is in an invalid  
2607        state caused by a previous execution error. Call **GrB\_error()** to  
2608        access any error messages generated by the implementation.

2609        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2610       GrB\_INSUFFICIENT\_SPACE Not enough space in `indptr`, `indices`, and/or `values` (as indicated  
2611                                   by the corresponding `n_*` parameter) to hold all of the corre-  
2612                                   sponding elements that will be extacted.

2613       GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2614                                   any matrix constructor.

2615               GrB\_NULL\_POINTER `indptr`, `indices`, `values` `n_indptr`, `n_indices`, `n_values` pointer is  
2616                                   NULL.

2617       GrB\_DOMAIN\_MISMATCH The domain of `A` does not match with the type of `values`.

## 2618 Description

2619 Given a matrix `A`, this method exports the contents of the matrix into one of the pre-defined  
2620 `GrB_Format` formats from Section 3.5.3.1. The user-allocated arrays pointed to by `indptr`, `indices`,  
2621 and `values` must be at least large enough to hold the corresponding number of elements returned  
2622 by calling `GrB_Matrix_exportSize`. The value of `format` can be chosen arbitrarily, but a call to  
2623 `GrB_Matrix_exportHint` may suggest a format that results in the most efficient export. Details  
2624 of the contents of `indptr`, `indices`, and `values` corresponding to each supported format is given in  
2625 Appendix B.

### 2626 4.2.5.17 Matrix\_import: Import a matrix into a GraphBLAS object

## 2627 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);

```

## 2628 Parameters

2629       `A` (INOUT) On a successful return, contains a handle to the newly created Graph-  
2630                   BLAS matrix.

2631       `d` (IN) The type corresponding to the domain of the matrix being created. Can be  
2632           one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
2633           GraphBLAS type.

2634        **nrows** (IN) Integer value holding the number of rows in the matrix.

2635        **ncols** (IN) Integer value holding the number of columns in the matrix.

2636        **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the  
2637        value of **format**.

2638        **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending  
2639        on the value of **format**.

2640        **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2641        **n\_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2642        **n\_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2643        **n\_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2644        **format** (IN) a value indicating the format of the matrix being imported, as defined in  
2645        Section 3.5.3.1.

## 2646    **Return Values**

2647        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2648        blocking mode, this indicates that the API checks for the input  
2649        arguments passed successfully and the input arrays have been  
2650        consumed. Either way, output matrix **A** is ready to be used in  
2651        the next method of the sequence.

2652        **GrB\_PANIC** Unknown internal error.

2653        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2654        **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2655        (needed for user-defined types).

2656        **GrB\_NULL\_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2657        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices  
2658        in **A** and or the size of **values**, **n\_values**, depending on the value  
2659        of **format**.

2660        **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

2661        **GrB\_DOMAIN\_MISMATCH** The domain given in parameter **d** does not match the element  
2662        type of **values**.



## 2663 Description

2664 Creates a new matrix **A** of domain **D**(d) and dimension **nrows**  $\times$  **ncols**. The new GraphBLAS  
2665 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.  
2666 The method returns a handle to the new matrix in **A**. The structure of the data being imported is  
2667 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of  
2668 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2669 It is not an error to call this method more than once on the same output matrix; however, the  
2670 handle to the previously created object will be overwritten.

## 2671 4.2.5.18 Matrix\_serializeSize: Compute the serialize buffer size

2672 Compute the buffer size (in bytes) necessary to serialize a GrB\_Matrix using GrB\_Matrix\_serialize.

## 2673 C Syntax

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

## 2674 Parameters

2675 **size** (OUT) Pointer to GrB\_Index value where size in bytes of serialized object will be  
2676 written.

2677 **A** (IN) A GraphBLAS matrix object.

## 2678 Return Values

2679 **GrB\_SUCCESS** The operation completed successfully and the value pointed to  
2680 by **\*size** has been computed and is ready to use.

2681 **GrB\_PANIC** Unknown internal error.

2682 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2683 **GrB\_NULL\_POINTER** **size** is NULL.

## 2684 Description

2685 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object **A**.  
2686 Users may then allocate a buffer of **size** bytes to pass as a parameter to GrB\_Matrix\_serialize.

2687 **4.2.5.19 Matrix\_serialize: Serialize a GraphBLAS matrix.**

2688 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2689 **C Syntax**

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

2690 **Parameters**

2691 **serialized\_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be  
2692 written.

2693 **serialized\_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized\_data**.  
2694 On output, the number of bytes written to **serialized\_data**.

2695 **A** (IN) A GraphBLAS matrix object.

2696 **Return Values**

2697 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2698 cessfully. This indicates that the compatibility tests on the in-  
2699 put argument passed successfully, and the output buffer **serial-  
2700 ized\_data** and **serialized\_size**, have been computed and are ready  
2701 to use.

2702 **GrB\_PANIC** Unknown internal error.

2703 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2704 opaque GraphBLAS objects (input or output) is in an invalid  
2705 state caused by a previous execution error. Call **GrB\_error()** to  
2706 access any error messages generated by the implementation.

2707 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2708 **GrB\_NULL\_POINTER** **serialized\_data** or **serialize\_size** is NULL.

2709 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2710 any matrix constructor.

2711 **GrB\_INSUFFICIENT\_SPACE** The size of the buffer **serialized\_data** (provided as an input **seri-  
2712 alized\_size**) was not large enough.

## 2713 Description

2714 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,  
2715 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must  
2716 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the  
2717 serialized matrix written to `serialized_data` is provided upon completion as an output written to  
2718 `serialized_size`.

2719 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created  
2720 with one library implementation is not necessarily valid for deserialization with another implemen-  
2721 tation.

### 2722 4.2.5.20 Matrix\_deserialize: Deserialize a GraphBLAS matrix.

2723 Construct a new GraphBLAS matrix from a serialized object.

## 2724 C Syntax

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

## 2725 Parameters

2726 A (INOUT) On a successful return, contains a handle to the newly created Graph-  
2727 BLAS matrix.

2728 d (IN) the type of the matrix that was serialized in `serialized_data`.  
2729 If d is `GrB_NULL`, the implementation must attempt to deserialize the matrix  
2730 without a provided type object.

2731 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2732 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

## 2733 Return Values

2734 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
2735 blocking mode, this indicates that the API checks for the input  
2736 arguments passed successfully. Either way, output matrix A is  
2737 ready to be used in the next method of the sequence.

2738 `GrB_PANIC` Unknown internal error.

2739 `GrB_INVALID_OBJECT` This is returned if `serialized_data` is invalid or corrupted.



## 2767 **Return Value**

2768                   GrB\_SUCCESS The method completed successfully.

2769                   GrB\_PANIC unknown internal error.

2770           GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2771           GrB\_NULL\_POINTER desc pointer is NULL.

## 2772 **Description**

2773 Creates a new descriptor object and returns a handle to it in **desc**. A newly created descriptor can  
2774 be populated by calls to **Descriptor\_set**.

2775 It is not an error to call this method more than once on the same variable; however, the handle to  
2776 the previously created object will be overwritten.

### 2777 **4.2.6.2 Descriptor\_set: Set content of descriptor**

2778 Sets the content for a field for an existing descriptor.

## 2779 **C Syntax**

```
2780           GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2781                                       GrB_Desc_Field        field,  
2782                                       GrB_Desc_Value        val);
```

## 2783 **Parameters**

2784           **desc** (IN) An existing GraphBLAS descriptor to be modified.

2785           **field** (IN) The field being set.

2786           **val** (IN) New value for the field being set.

## 2787 **Return Values**

2788                   GrB\_SUCCESS operation completed successfully.

2789                   GrB\_PANIC unknown internal error.

2790           GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2791 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to **new**.

2792           GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 2793 Description

2794 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor  
2795 to set the value associated with that field. Valid values for the `field` parameter include the following:

2796 `GrB_OUTP` refers to the output parameter (result) of the operation.

2797 `GrB_MASK` refers to the mask parameter of the operation.

2798 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2799 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2800 Valid values for the `val` parameter are:

2801 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask  
2802 (`GrB_MASK`) parameter.

2803 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-  
2804 eter. When combined with `GrB_STRUCTURE`, the complement of the  
2805 structure of the mask is used without evaluating the values stored.

2806 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input  
2807 matrix parameters only).

2808 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear  
2809 the matrix first (or clear the non-masked entries). The default behavior  
2810 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`  
2811 parameter only.

2812 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,  
2813 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).  
2814 A value for a given field may be set multiple times but will have no additional effect. Fields that  
2815 have no values set result in their default behavior, as defined in Section 3.6.

## 2816 4.2.7 free: Destroy an object and release its resources

2817 Destroys a previously created GraphBLAS object and releases any resources associated with the  
2818 object.

## 2819 C Syntax

2820 `GrB_Info GrB_free(<GrB_Object> *obj);`

## 2821 Parameters

2822       obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have  
2823       been created by an explicit call to a GraphBLAS constructor. It can be any of the  
2824       opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,  
2825       binary op, unary op, or type. On successful completion of GrB\_free, obj behaves  
2826       as an uninitialized object.

## 2827 Return Values

2828       GrB\_SUCCESS operation completed successfully

2829       GrB\_PANIC unknown internal error. If this return value is encountered when  
2830       in nonblocking mode, the error responsible for the panic condition  
2831       could be from any method involved in the computation of the input  
2832       object. The GrB\_error() method should be called for additional  
2833       information.

## 2834 Description

2835       GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime  
2836       system. A call to GrB\_free frees those resources so they are available for use by other GraphBLAS  
2837       objects.

2838       The parameter passed into GrB\_free is a handle referencing a GraphBLAS opaque object of a data  
2839       type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-  
2840       structor. The behavior of a program that calls GrB\_free on a pre-defined object is implementation  
2841       defined.

2842       After the GrB\_free method returns, the object referenced by the input handle is destroyed and the  
2843       handle has the value GrB\_INVALID\_HANDLE. The handle can be used in subsequent GraphBLAS  
2844       methods but only after the handle has been reinitialized with a call the the appropriate \_new or  
2845       \_dup method.

2846       Note that unlike other GraphBLAS methods, calling GrB\_free with an object with an invalid handle  
2847       is legal. The system may attempt to free resources that might be associated with that object, if  
2848       possible, and return normally.

2849       When using GrB\_free it is possible to create a dangling reference to an object. This would occur  
2850       when a handle is assigned to a second variable of the same opaque type. This creates two handles  
2851       that reference the same object. If GrB\_free is called with one of the variables, the object is destroyed  
2852       and the handle associated with the other variable no longer references a valid object. This is not an  
2853       error condition that the implementation of the GraphBLAS API can be expected to catch, hence  
2854       programmers must take care to prevent this situation from occurring.

2855 **4.2.8 wait: Return once an object is either *complete* or *materialized***

2856 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2857 **C Syntax**

2858 `GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);`

2859 **Parameters**

2860 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an  
2861 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2862 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2863 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another  
2864 thread (completion) or all computing to produce `obj` by all GraphBLAS operations  
2865 in its sequence have finished (materialization).

2866 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the  
2867 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or  
2868 `GrB_MATERIALIZE`.

2869 **Return values**

2870 `GrB_SUCCESS` operation completed successfully.

2871 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-  
2872 pletion of pending operations.

2873 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion  
2874 of pending operations.

2875 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2876 or other constructor, method.

2877 `GrB_PANIC` unknown internal error.

2878 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`  
2879 `GrB_MATERIALIZE`.

2880 **Description**

2881 On successful return from `GrB_wait()`, the input object, `obj` is in one of two states depending on  
2882 the mode of `GrB_wait`:



- 2883 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program  
2884 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another  
2885 thread. This result occurs when the mode parameter is set to `GrB_COMPLETE`.
- 2886 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on  
2887 behalf of `obj` and error information is available. This result occurs when the mode parameter  
2888 is set to `GrB_MATERIALIZE`.

2889 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,  
2890 `GrB_wait(obj,mode)` has no effect when called in blocking mode.

2891 In non-blocking mode, the status of any pending method calls, other than those associated with pro-  
2892 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to `GrB_wait(obj,mode)`.  
2893 Methods in the sequence for `obj`, however, most likely would be impacted by a call to `GrB_wait(obj,mode)`;  
2894 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be  
2895 finished prior to the return from `GrB_wait(obj,mode)`.

#### 2896 4.2.9 error: Retrieve an error string

2897 Retrieve an error-message about any errors encountered during the processing associated with an  
2898 object.

### 2899 C Syntax

```
2900      GrB_Info GrB_error(const char      **error,
2901                        const GrB_Object  obj);
```

#### 2902 Parameters

2903 `error` (OUT) A pointer to a null-terminated string. The contents of the string are im-  
2904 plementation defined.

2905 `obj` (IN) An existing GraphBLAS object. The object must have been created by an  
2906 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2907 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2908 or type.

#### 2909 Return value

2910 `GrB_SUCCESS` operation completed successfully.

2911 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2912 or other constructor, method.

2913 `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	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

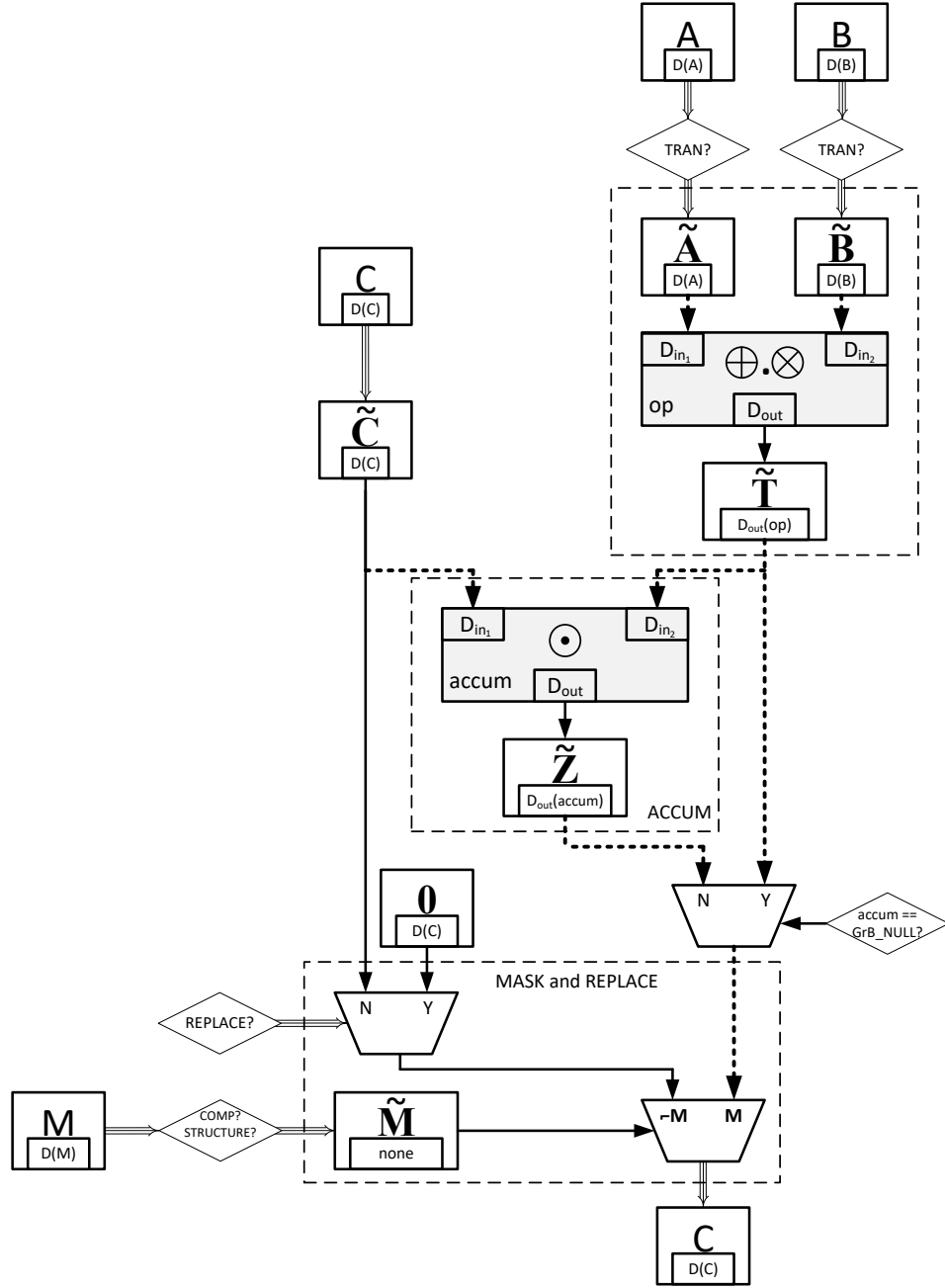


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows ( $\Rightarrow$ ) denote where “as if copy” takes place (including both collections and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

2983 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-  
 2984 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at  
 2985 the point of their declaration, as shown in the following examples:

```
2986         GrB_Type          type = GrB_INVALID_HANDLE;
2987         GrB_Semiring       semiring = GrB_INVALID_HANDLE;
2988         GrB_Matrix        matrix = GrB_INVALID_HANDLE;
```

## 2989 Compliance

2990 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.  
 2991 That is, for each operation we give a recipe for producing its outcome. Any implementation that  
 2992 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error  
 2993 model (Section 2.6) is a conforming implementation.

### 2994 4.3.1 mxm: Matrix-matrix multiply

2995 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

## 2996 C Syntax

```
2997         GrB_Info GrB_mxm(GrB_Matrix          C,
2998                           const GrB_Matrix    Mask,
2999                           const GrB_BinaryOp  accum,
3000                           const GrB_Semiring  op,
3001                           const GrB_Matrix    A,
3002                           const GrB_Matrix    B,
3003                           const GrB_Descriptor desc);
```

## 3004 Parameters

3005 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 3006 that may be accumulated with the result of the matrix product. On output, the  
 3007 matrix holds the results of the operation.

3008 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 3009 stored into the output matrix C. The mask dimensions must match those of the  
 3010 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
 3011 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
 3012 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 3013 dimensions of C), `GrB_NULL` should be specified.

3014 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
 3015 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 3016 specified.

3017 **op** (IN) The semiring used in the matrix-matrix multiply.

3018 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3019 multiplication.

3020 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 3021 multiplication.

3022 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 3023 should be specified. Non-default field/value pairs are listed as follows:  
 3024

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.

## 3026 Return Values

3027 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 3028 blocking mode, this indicates that the compatibility tests on di-  
 3029 mensions and domains for the input arguments passed successfully.  
 3030 Either way, output matrix **C** is ready to be used in the next method  
 3031 of the sequence.

3032 **GrB\_PANIC** Unknown internal error.

3033 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 3034 GraphBLAS objects (input or output) is in an invalid state caused  
 3035 by a previous execution error. Call **GrB\_error()** to access any error  
 3036 messages generated by the implementation.

3037 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3038 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 3039 a call to **new** (or **Matrix\_dup** for matrix parameters).

3040 **GrB\_DIMENSION\_MISMATCH** Mask and/or matrix dimensions are incompatible.

3041 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3042 corresponding domains of the semiring or accumulation operator,  
 3043 or the mask's domain is not compatible with `bool` (in the case where  
 3044 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3045 Description

3046 GrB\_mxm computes the matrix product  $C = A \oplus . \otimes B$  or, if an optional binary accumulation operator  
 3047  $(\odot)$  is provided,  $C = C \odot (A \oplus . \otimes B)$  (where matrices  $A$  and  $B$  can be optionally transposed).  
 3048 Logically, this operation occurs in three steps:

3049 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3050 and dimensions are tested for compatibility.

3051 **Compute** The indicated computations are carried out.

3052 **Output** The result is written into the output matrix, possibly under control of a mask.

3053 Up to four argument matrices are used in the GrB\_mxm operation:

- 3054 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3055 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)
- 3056 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3057 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3058 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for  
 3059 domain compatibility as follows:

- 3060 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3061 must be from one of the pre-defined types of Table 3.2.
- 3062 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3063 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3064 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3065 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3066 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 3067 of the accumulation operator.

3068 Two domains are compatible with each other if values from one domain can be cast to values in  
 3069 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3070 all compatible with each other. A domain from a user-defined type is only compatible with itself.



3071 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch  
 3072 error listed above is returned.

3073 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3074 ( $\leftarrow$  denotes copy):

- 3075 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3076 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3077 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3078  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3079 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3080 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3081  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3082 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3083  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 3084 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3085 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3086 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3087 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3088 must hold:

- 3089 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 3090 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 3091 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 3092 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .
- 3093 5.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .

3094 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch  
 3095 error listed above is returned.

3096 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3097 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3098 We are now ready to carry out the matrix multiplication and any additional associated operations.  
 3099 We describe this in terms of two intermediate matrices:

- 3100 •  $\tilde{\mathbf{T}}$ : The matrix holding the product of matrices  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3101 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3102 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

$$3104 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3105 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring  $\mathbf{op}$ , respectively.

3106 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3107 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3108 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3109 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3110 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
3111 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 3112 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3113 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3114 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3115 \end{aligned}$$

3116 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3118 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
3119 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
3120 mask which acts as a “write mask”.

- 3121 • If  $\mathbf{desc}[\mathbf{GrB\_OUTP}].\mathbf{GrB\_REPLACE}$  is set, then any values in  $\mathbf{C}$  on input to this operation are  
3122 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3123 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3124 • If  $\mathbf{desc}[\mathbf{GrB\_OUTP}].\mathbf{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
3125 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
3126 mask are unchanged:

$$3127 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3128 In  $\mathbf{GrB\_BLOCKING}$  mode, the method exits with return value  $\mathbf{GrB\_SUCCESS}$  and the new content  
3129 of matrix  $\mathbf{C}$  is as defined above and fully computed. In  $\mathbf{GrB\_NONBLOCKING}$  mode, the method  
3130 exits with return value  $\mathbf{GrB\_SUCCESS}$  and the new content of matrix  $\mathbf{C}$  is as defined above but  
3131 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3132 sequence.

### 3133 4.3.2 vxm: Vector-matrix multiply

3134 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

#### 3135 C Syntax

```
3136      GrB_Info GrB_vxm(GrB_Vector      w,  
3137                      const GrB_Vector mask,  
3138                      const GrB_BinaryOp accum,  
3139                      const GrB_Semiring op,  
3140                      const GrB_Vector u,  
3141                      const GrB_Matrix A,  
3142                      const GrB_Descriptor desc);
```

#### 3143 Parameters

3144 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3145 that may be accumulated with the result of the vector-matrix product. On output,  
3146 this vector holds the results of the operation.

3147 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3148 stored into the output vector **w**. The mask dimensions must match those of the  
3149 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3150 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3151 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3152 dimensions of **w**), **GrB\_NULL** should be specified.

3153 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3154 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3155 specified.

3156 **op** (IN) Semiring used in the vector-matrix multiply.

3157 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3158 multiplication.

3159 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
3160 multiplication.

3161 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
3162 should be specified. Non-default field/value pairs are listed as follows:  
3163

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.

3191     **Output** The result is written into the output vector, possibly under control of a mask.

3192 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3193     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3194     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3195     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3196     4.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3197 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
3198 tested for domain compatibility as follows:

- 3199     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
3200         must be from one of the pre-defined types of Table 3.2.
- 3201     2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the semiring.
- 3202     3.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the semiring.
- 3203     4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring.
- 3204     5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
3205         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$   
3206         of the accumulation operator.

3207 Two domains are compatible with each other if values from one domain can be cast to values in  
3208 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
3209 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
3210 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch  
3211 error listed above is returned.

3212 From the argument vectors and matrices, the internal matrices and mask used in the computation  
3213 are formed ( $\leftarrow$  denotes copy):

- 3214     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3215     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3216         (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3217         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3218             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 3219             ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 3220         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3221     3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3222 4. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

3223 The internal matrices and masks are checked for shape compatibility. The following conditions  
3224 must hold:

3225 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$ .

3226 2.  $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$ .

3227 3.  $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$ .

3228 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch  
3229 error listed above is returned.

3230 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3231 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3232 We are now ready to carry out the vector-matrix multiplication and any additional associated  
3233 operations. We describe this in terms of two intermediate vectors:

- 3234 •  $\tilde{\mathbf{t}}$ : The vector holding the product of vector  $\tilde{\mathbf{u}}^T$  and matrix  $\tilde{\mathbf{A}}$ .
- 3235 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3236 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.  
3237 The value of each of its elements is computed by

$$3238 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3239 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring `op`, respectively.

3240 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3241 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3242 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3243 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3244 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3245 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3246 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3247 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3248 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3249 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3250 \end{aligned}$$

3251 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3252 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3253 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3254 mask which acts as a “write mask”.

- 3255 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3256 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3257 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3258 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3259 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3260 mask are unchanged:

$$3261 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3262 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3263 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3264 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3265 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3266 sequence.

### 3267 4.3.3 mxv: Matrix-vector multiply

3268 Multiplies a matrix by a vector on a semiring. The result is a vector.

## 3269 C Syntax

```
3270      GrB_Info GrB_mxv(GrB_Vector      w,
3271                      const GrB_Vector mask,
3272                      const GrB_BinaryOp accum,
3273                      const GrB_Semiring op,
3274                      const GrB_Matrix A,
3275                      const GrB_Vector u,
3276                      const GrB_Descriptor desc);
```

## 3277 Parameters

3278 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
 3279 that may be accumulated with the result of the matrix-vector product. On output,  
 3280 this vector holds the results of the operation.

3281 **mask** (IN) An optional “write” mask that controls which results from this operation are  
 3282 stored into the output vector  $\mathbf{w}$ . The mask dimensions must match those of the  
 3283 vector  $\mathbf{w}$ . If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain

3284 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types  
 3285 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 3286 dimensions of `w`), `GrB_NULL` should be specified.

3287 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`  
 3288 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
 3289 specified.

3290 `op` (IN) Semiring used in the vector-matrix multiply.

3291 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3292 multiplication.

3293 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3294 multiplication.

3295 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
 3296 should be specified. Non-default field/value pairs are listed as follows:  
 3297

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.

## 3299 Return Values

3300 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 3301 blocking mode, this indicates that the compatibility tests on di-  
 3302 mensions and domains for the input arguments passed successfully.  
 3303 Either way, output vector `w` is ready to be used in the next method  
 3304 of the sequence.

3305 `GrB_PANIC` Unknown internal error.

3306 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
 3307 GraphBLAS objects (input or output) is in an invalid state caused  
 3308 by a previous execution error. Call `GrB_error()` to access any error  
 3309 messages generated by the implementation.

3310 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3311 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by  
 3312 a call to `new` (or `dup` for matrix or vector parameters).



3313 GrB\_DIMENSION\_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3314 GrB\_DOMAIN\_MISMATCH The domains of the various vectors/matrices are incompatible with  
3315 the corresponding domains of the semiring or accumulation opera-  
3316 tor, or the mask's domain is not compatible with **bool** (in the case  
3317 where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3318 Description

3319 GrB\_mvx computes the matrix-vector product  $w = A \oplus . \otimes u$ , or, if an optional binary accumulation  
3320 operator ( $\odot$ ) is provided,  $w = w \odot (A \oplus . \otimes u)$  (where matrix  $A$  can be optionally transposed).  
3321 Logically, this operation occurs in three steps:

3322 **Setup** The internal vectors, matrices and mask used in the computation are formed and their  
3323 domains/dimensions are tested for compatibility.

3324 **Compute** The indicated computations are carried out.

3325 **Output** The result is written into the output vector, possibly under control of a mask.

3326 Up to four argument vectors or matrices are used in the GrB\_mvx operation:

- 3327 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3328 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3329 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3330 4.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3331 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
3332 tested for domain compatibility as follows:

- 3333 1. If **mask** is not GrB\_NULL, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
3334 must be from one of the pre-defined types of Table 3.2.
- 3335 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3336 3.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3337 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3338 5. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3339 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
3340 of the accumulation operator.

3341 Two domains are compatible with each other if values from one domain can be cast to values in  
 3342 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3343 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
 3344 If any compatibility rule above is violated, execution of `GrB_mxv` ends and the domain mismatch  
 3345 error listed above is returned.

3346 From the argument vectors and matrices, the internal matrices and mask used in the computation  
 3347 are formed ( $\leftarrow$  denotes copy):

- 3348 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3349 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3350 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 3351 (b) If `mask  $\neq$  GrB_NULL`,
    - 3352 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 3353 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 3354 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3355 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3356 4. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3357 The internal matrices and masks are checked for shape compatibility. The following conditions  
 3358 must hold:

- 3359 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$ .
- 3360 2.  $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}(\tilde{\mathbf{A}})$ .
- 3361 3.  $\text{size}(\tilde{\mathbf{u}}) = \text{ncols}(\tilde{\mathbf{A}})$ .

3362 If any compatibility rule above is violated, execution of `GrB_mxv` ends and the dimension mismatch  
 3363 error listed above is returned.

3364 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3365 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3366 We are now ready to carry out the matrix-vector multiplication and any additional associated  
 3367 operations. We describe this in terms of two intermediate vectors:

- 3368 •  $\tilde{\mathbf{t}}$ : The vector holding the product of matrix  $\tilde{\mathbf{A}}$  and vector  $\tilde{\mathbf{u}}$ .
- 3369 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3370 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.  
 3371 The value of each of its elements is computed by

$$3372 \quad t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3373 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring **op**, respectively.

3374 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3375 • If **accum** = **GrB\_NULL**, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3376 • If **accum** is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3377 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3378 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3379 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3380 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3381 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3382 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3383 \end{aligned}$$

3384 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3386 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector **w**,  
 3387 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3388 mask which acts as a “write mask”.

- 3389 • If **desc[GrB\_OUTP].GrB\_REPLACE** is set, then any values in **w** on input to this operation are  
 3390 deleted and the content of the new output vector, **w**, is defined as,

$$3391 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3392 • If **desc[GrB\_OUTP].GrB\_REPLACE** is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3393 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the  
 3394 mask are unchanged:

$$3395 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3396 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 3397 of vector **w** is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 3398 exits with return value **GrB\_SUCCESS** and the new content of vector **w** is as defined above but  
 3399 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3400 sequence.

#### 3401 4.3.4 eWiseMult: Element-wise multiplication

3402 **Note:** The difference between **eWiseAdd** and **eWiseMult** is not about the element-wise operation  
 3403 but how the index sets are treated. **eWiseAdd** returns an object whose indices are the “union” of  
 3404 the indices of the inputs whereas **eWiseMult** returns an object whose indices are the “intersection”  
 3405 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3406 the set of values from the resulting index set.

#### 3407 4.3.4.1 eWiseMult: Vector variant

3408 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-  
3409 ducing a third vector as result.

#### 3410 C Syntax

```
3411     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3412                           const GrB_Vector mask,  
3413                           const GrB_BinaryOp accum,  
3414                           const GrB_Semiring op,  
3415                           const GrB_Vector u,  
3416                           const GrB_Vector v,  
3417                           const GrB_Descriptor desc);  
3418  
3419     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3420                           const GrB_Vector mask,  
3421                           const GrB_BinaryOp accum,  
3422                           const GrB_Monoid op,  
3423                           const GrB_Vector u,  
3424                           const GrB_Vector v,  
3425                           const GrB_Descriptor desc);  
3426  
3427     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3428                           const GrB_Vector mask,  
3429                           const GrB_BinaryOp accum,  
3430                           const GrB_BinaryOp op,  
3431                           const GrB_Vector u,  
3432                           const GrB_Vector v,  
3433                           const GrB_Descriptor desc);
```

#### 3434 Parameters

3435 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3436 that may be accumulated with the result of the element-wise operation. On output,  
3437 this vector holds the results of the operation.

3438 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3439 stored into the output vector **w**. The mask dimensions must match those of the  
3440 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3441 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3442 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3443 dimensions of **w**), **GrB\_NULL** should be specified.

3444 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3445 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3446 specified.

3447 op (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
3448 operation. Depending on which type is passed, the following defines the binary  
3449 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3450 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3451 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-  
3452 nored.

3453 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  
3454 is ignored.

3455 u (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3456 operation.

3457 v (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
3458 operation.

3459 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
3460 should be specified. Non-default field/value pairs are listed as follows:

3461

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.

3462

## 3463 Return Values

3464 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
3465 blocking mode, this indicates that the compatibility tests on di-  
3466 mensions and domains for the input arguments passed successfully.  
3467 Either way, output vector w is ready to be used in the next method  
3468 of the sequence.

3469 GrB\_PANIC Unknown internal error.

3470 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
3471 GraphBLAS objects (input or output) is in an invalid state caused  
3472 by a previous execution error. Call GrB\_error() to access any error  
3473 messages generated by the implementation.

3474 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3475 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3476 a call to `new` (or `dup` for vector parameters).

3477 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3478 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
 3479 responding domains of the binary operator (`op`) or accumulation  
 3480 operator, or the mask's domain is not compatible with `bool` (in the  
 3481 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3482 Description

3483 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:  
 3484  $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$ .  
 3485 Logically, this operation occurs in three steps:

3486 **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3487 and dimensions are tested for compatibility.

3488 **Compute** The indicated computations are carried out.

3489 **Output** The result is written into the output vector, possibly under control of a mask.

3490 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3491 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3492 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3493 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3494 4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3495 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)  
 3496 are tested for domain compatibility as follows:

- 3497 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3498 must be from one of the pre-defined types of Table 3.2.
- 3499 2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3500 3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3501 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3502 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3503 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
 3504 the accumulation operator.

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

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

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

The internal vectors and mask are checked for dimension compatibility. The following conditions must hold:

1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}}) = \mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{v}})$ .

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

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

We are now ready to carry out the element-wise “product” and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$ : The vector holding the element-wise “product” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The value of each of its elements is computed by:

$$t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}))$$

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

3536 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

3537 • If  $\text{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

3538 
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3539 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3540 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

3541 
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3542

3543 
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3544

3545 
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3546 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3547 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3548 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3549 mask which acts as a “write mask”.

3550 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3551 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

3552 
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3553 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3554 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3555 mask are unchanged:

3556 
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3557 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 3558 of vector  $\mathbf{w}$  is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 3559 exits with return value **GrB\_SUCCESS** and the new content of vector  $\mathbf{w}$  is as defined above but  
 3560 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3561 sequence.

#### 3562 4.3.4.2 eWiseMult: Matrix variant

3563 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-  
 3564 ducing a third matrix as result.



## 3565 C Syntax

```

3566         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3567                                const GrB_Matrix  Mask,
3568                                const GrB_BinaryOp accum,
3569                                const GrB_Semiring op,
3570                                const GrB_Matrix  A,
3571                                const GrB_Matrix  B,
3572                                const GrB_Descriptor desc);
3573
3574         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3575                                const GrB_Matrix  Mask,
3576                                const GrB_BinaryOp accum,
3577                                const GrB_Monoid   op,
3578                                const GrB_Matrix  A,
3579                                const GrB_Matrix  B,
3580                                const GrB_Descriptor desc);
3581
3582         GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3583                                const GrB_Matrix  Mask,
3584                                const GrB_BinaryOp accum,
3585                                const GrB_BinaryOp op,
3586                                const GrB_Matrix  A,
3587                                const GrB_Matrix  B,
3588                                const GrB_Descriptor desc);

```

## 3589 Parameters

3590 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3591 that may be accumulated with the result of the element-wise operation. On output,  
3592 the matrix holds the results of the operation.

3593 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3594 stored into the output matrix C. The mask dimensions must match those of the  
3595 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3596 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types  
3597 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3598 dimensions of C), `GrB_NULL` should be specified.

3599 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3600 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3601 specified.

3602 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
3603 operation. Depending on which type is passed, the following defines the binary  
3604 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3605 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
 3606 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-  
 3607 nored.  
 3608 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  
 3609 is ignored.

3610 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3611 operation.

3612 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 3613 operation.

3614 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 3615 should be specified. Non-default field/value pairs are listed as follows:  
 3616

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.

## 3618 Return Values

3619 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 3620 blocking mode, this indicates that the compatibility tests on di-  
 3621 mensions and domains for the input arguments passed successfully.  
 3622 Either way, output matrix C is ready to be used in the next method  
 3623 of the sequence.

3624 GrB\_PANIC Unknown internal error.

3625 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 3626 GraphBLAS objects (input or output) is in an invalid state caused  
 3627 by a previous execution error. Call GrB\_error() to access any error  
 3628 messages generated by the implementation.

3629 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3630 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3631 a call to new (or Matrix\_dup for matrix parameters).

3632 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

3633 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3634 corresponding domains of the binary operator ( $\otimes$ ) or accumulation  
 3635 operator, or the mask's domain is not compatible with `bool` (in the  
 3636 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3637 Description

3638 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:  
 3639  $C = A \otimes B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$ .  
 3640 Logically, this operation occurs in three steps:

3641 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3642 and dimensions are tested for compatibility.

3643 **Compute** The indicated computations are carried out.

3644 **Output** The result is written into the output matrix, possibly under control of a mask.

3645 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3646 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3647 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)
- 3648 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3649 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3650 The argument matrices, the “product” operator ( $\otimes$ ), and the accumulation operator (if provided)  
 3651 are tested for domain compatibility as follows:

- 3652 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3653 must be from one of the pre-defined types of Table 3.2.
- 3654 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\otimes)$ .
- 3655 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\otimes)$ .
- 3656 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\otimes)$ .
- 3657 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3658 of the accumulation operator and  $\mathbf{D}_{out}(\otimes)$  of  $\otimes$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3659 the accumulation operator.

3660 Two domains are compatible with each other if values from one domain can be cast to values in  
 3661 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3662 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3663 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3664 error listed above is returned.

3665 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3666 ( $\leftarrow$  denotes copy):

- 3667 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3668 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3669 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3670  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3671 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3672 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3673  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3674 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3675  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$ .
  - 3676 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3677 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3678 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3679 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3680 must hold:

- 3681 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
- 3682 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

3683 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
 3684 mismatch error listed above is returned.

3685 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3686 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3687 We are now ready to carry out the element-wise “product” and any additional associated operations.  
 3688 We describe this in terms of two intermediate matrices:

- 3689 •  $\tilde{\mathbf{T}}$ : The matrix holding the element-wise product of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3690 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3691 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$   
 3692 is created. The value of each of its elements is computed by

$$3693 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3694 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

3695 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

3696 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3697 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3698 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 3699 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$3700 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3701 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3702 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3703 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3704 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3705 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3706 mask which acts as a “write mask”.

3707 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3708 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3709 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3710 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3711 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3712 mask are unchanged:

$$3713 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3714 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 3715 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 3716 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 3717 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3718 sequence.

### 3721 4.3.5 eWiseAdd: Element-wise addition

3722 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation  
 3723 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of  
 3724 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”  
 3725 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3726 the set of values from the resulting index set.

#### 3727 4.3.5.1 eWiseAdd: Vector variant

3728 Perform element-wise (general) addition on the elements of two vectors, producing a third vector  
3729 as result.

#### 3730 C Syntax

```
3731     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3732                          const GrB_Vector mask,  
3733                          const GrB_BinaryOp accum,  
3734                          const GrB_Semiring op,  
3735                          const GrB_Vector u,  
3736                          const GrB_Vector v,  
3737                          const GrB_Descriptor desc);  
3738  
3739     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3740                          const GrB_Vector mask,  
3741                          const GrB_BinaryOp accum,  
3742                          const GrB_Monoid op,  
3743                          const GrB_Vector u,  
3744                          const GrB_Vector v,  
3745                          const GrB_Descriptor desc);  
3746  
3747     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3748                          const GrB_Vector mask,  
3749                          const GrB_BinaryOp accum,  
3750                          const GrB_BinaryOp op,  
3751                          const GrB_Vector u,  
3752                          const GrB_Vector v,  
3753                          const GrB_Descriptor desc);
```

#### 3754 Parameters

3755 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3756 that may be accumulated with the result of the element-wise operation. On output,  
3757 this vector holds the results of the operation.

3758 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3759 stored into the output vector **w**. The mask dimensions must match those of the  
3760 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3761 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3762 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3763 dimensions of **w**), **GrB\_NULL** should be specified.

3764 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3765 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3766 specified.

3767 op (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3768 operation. Depending on which type is passed, the following defines the binary  
3769 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3770 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3771 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-  
3772 nored.

3773 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-  
3774 nary op and additive identity are ignored.

3775 u (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3776 operation.

3777 v (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
3778 operation.

3779 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
3780 should be specified. Non-default field/value pairs are listed as follows:

3781

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.

3782

## 3783 Return Values

3784 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
3785 blocking mode, this indicates that the compatibility tests on di-  
3786 mensions and domains for the input arguments passed successfully.  
3787 Either way, output vector w is ready to be used in the next method  
3788 of the sequence.

3789 GrB\_PANIC Unknown internal error.

3790 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
3791 GraphBLAS objects (input or output) is in an invalid state caused  
3792 by a previous execution error. Call GrB\_error() to access any error  
3793 messages generated by the implementation.

3794 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3795 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
3796 a call to `new` (or `dup` for vector parameters).

3797 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3798 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
3799 responding domains of the binary operator (`op`) or accumulation  
3800 operator, or the mask's domain is not compatible with `bool` (in the  
3801 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3802 Description

3803 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors:  $\mathbf{w} =$   
3804  $\mathbf{u} \oplus \mathbf{v}$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$ . Logically,  
3805 this operation occurs in three steps:

3806 **Setup** The internal vectors and mask used in the computation are formed and their domains  
3807 and dimensions are tested for compatibility.

3808 **Compute** The indicated computations are carried out.

3809 **Output** The result is written into the output vector, possibly under control of a mask.

3810 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3811 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3812 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3813 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3814 4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3815 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are  
3816 tested for domain compatibility as follows:

- 3817 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
3818 must be from one of the pre-defined types of Table 3.2.
- 3819 2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3820 3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3821 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3822 5.  $\mathbf{D}(\mathbf{u})$  and  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3823 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
3824 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
3825 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 vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$  denotes copy):

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

The internal vectors and mask are checked for dimension compatibility. The following conditions must hold:

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

If any compatibility rule above is violated, execution of `GrB_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 vectors:

- $\tilde{\mathbf{t}}$ : The vector holding the element-wise “sum” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{u}}) \cup \text{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The value of each of its elements is computed by:

$$t_i = (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))$$

$$t_i = \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})))$$

3858  
3859

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3860 where the difference operator in the previous expressions refers to set difference.

3861 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3862 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3863 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

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

3865 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3866 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3867 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3868 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3869 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3870 \end{aligned}$$

3871 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3872 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
3873 using what is called a *standard vector mask and replace*. This is carried out under control of the  
3874 mask which acts as a “write mask”.  
3875

- 3876 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
3877 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

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

- 3879 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
3880 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
3881 mask are unchanged:

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

3883 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
3884 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
3885 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
3886 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3887 sequence.

#### 3888 4.3.5.2 eWiseAdd: Matrix variant

3889 Perform element-wise (general) addition on the elements of two matrices, producing a third matrix  
3890 as result.

## 3891 C Syntax

```

3892     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3893                           const GrB_Matrix Mask,
3894                           const GrB_BinaryOp accum,
3895                           const GrB_Semiring op,
3896                           const GrB_Matrix A,
3897                           const GrB_Matrix B,
3898                           const GrB_Descriptor desc);
3899
3900     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3901                           const GrB_Matrix Mask,
3902                           const GrB_BinaryOp accum,
3903                           const GrB_Monoid op,
3904                           const GrB_Matrix A,
3905                           const GrB_Matrix B,
3906                           const GrB_Descriptor desc);
3907
3908     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3909                           const GrB_Matrix Mask,
3910                           const GrB_BinaryOp accum,
3911                           const GrB_BinaryOp op,
3912                           const GrB_Matrix A,
3913                           const GrB_Matrix B,
3914                           const GrB_Descriptor desc);

```

## 3915 Parameters

3916 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3917 that may be accumulated with the result of the element-wise operation. On output,  
3918 the matrix holds the results of the operation.

3919 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3920 stored into the output matrix C. The mask dimensions must match those of the  
3921 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3922 of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types  
3923 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3924 dimensions of C), `GrB_NULL` should be specified.

3925 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3926 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3927 specified.

3928 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3929 operation. Depending on which type is passed, the following defines the binary  
3930 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3931 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
 3932 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-  
 3933 nored.  
 3934 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-  
 3935 nary op and additive identity are ignored.

3936 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3937 operation.

3938 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 3939 operation.

3940 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 3941 should be specified. Non-default field/value pairs are listed as follows:  
 3942

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.

## 3944 Return Values

3945 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 3946 blocking mode, this indicates that the compatibility tests on di-  
 3947 mensions and domains for the input arguments passed successfully.  
 3948 Either way, output matrix C is ready to be used in the next method  
 3949 of the sequence.

3950 GrB\_PANIC Unknown internal error.

3951 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 3952 GraphBLAS objects (input or output) is in an invalid state caused  
 3953 by a previous execution error. Call GrB\_error() to access any error  
 3954 messages generated by the implementation.

3955 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3956 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3957 a call to new (or Matrix\_dup for matrix parameters).

3958 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

3959 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3960 corresponding domains of the binary operator ( $\text{op}$ ) or accumulation  
 3961 operator, or the mask's domain is not compatible with `bool` (in the  
 3962 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3963 Description

3964 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:  
 3965  $C = A \oplus B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \oplus B)$ .  
 3966 Logically, this operation occurs in three steps:

3967 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3968 and dimensions are tested for compatibility.

3969 **Compute** The indicated computations are carried out.

3970 **Output** The result is written into the output matrix, possibly under control of a mask.

3971 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3972 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3973 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)
- 3974 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3975 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3976 The argument matrices, the “sum” operator ( $\text{op}$ ), and the accumulation operator (if provided) are  
 3977 tested for domain compatibility as follows:

- 3978 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3979 must be from one of the pre-defined types of Table 3.2.
- 3980 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3981 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3982 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3983 5.  $\mathbf{D}(A)$  and  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3984 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3985 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of  $\text{op}$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3986 the accumulation operator.

3987 Two domains are compatible with each other if values from one domain can be cast to values in  
 3988 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3989 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 3990 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch  
 3991 error listed above is returned.

3992 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3993 ( $\leftarrow$  denotes copy):

- 3994 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3995 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3996 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3997  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3998 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3999 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 4000  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 4001 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 4002  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 4003 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4004 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4005 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

4006 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 4007 must hold:

- 4008 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
- 4009 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

4010 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension  
 4011 mismatch error listed above is returned.

4012 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4013 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4014 We are now ready to carry out the element-wise “sum” and any additional associated operations.  
 4015 We describe this in terms of two intermediate matrices:

- 4016 •  $\tilde{\mathbf{T}}$ : The matrix holding the element-wise sum of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 4017 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4018 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$   
 4019 is created. The value of each of its elements is computed by

$$\begin{aligned} 4020 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\ 4021 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\ 4022 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \end{aligned}$$

4025 where the difference operator in the previous expressions refers to set difference.

4026 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4027 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 4028 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4029 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4030 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4031 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 4032 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4033 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4034 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \end{aligned}$$

4037 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4038 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 4039 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 4040 mask which acts as a “write mask”.

- 4041 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 4042 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$4043 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4044 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 4045 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 4046 mask are unchanged:

$$4047 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4048 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 4049 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 4050 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 4051 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 4052 sequence.

### 4053 4.3.6 extract: Selecting sub-graphs

4054 Extract a subset of a matrix or vector.

#### 4055 4.3.6.1 extract: Standard vector variant

4056 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector  
4057 whose size is equal to the number of indices.

### 4058 C Syntax

```
4059         GrB_Info GrB_extract(GrB_Vector          w,  
4060                             const GrB_Vector    mask,  
4061                             const GrB_BinaryOp   accum,  
4062                             const GrB_Vector    u,  
4063                             const GrB_Index     *indices,  
4064                             GrB_Index          nindices,  
4065                             const GrB_Descriptor desc);
```

### 4066 Parameters

4067 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4068 that may be accumulated with the result of the extract operation. On output, this  
4069 vector holds the results of the operation.

4070 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4071 stored into the output vector **w**. The mask dimensions must match those of the  
4072 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4073 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4074 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4075 dimensions of **w**), **GrB\_NULL** should be specified.

4076 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4077 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4078 specified.

4079 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4080 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of  
4081 elements from **u** that are extracted. If all elements of **u** are to be extracted in order  
4082 from 0 to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution  
4083 mode and return value, this array may be manipulated by the caller after this  
4084 operation returns without affecting any deferred computations for this operation.

4085 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.



4086 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4087 should be specified. Non-default field/value pairs are listed as follows:  
 4088

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> .

## 4090 Return Values

4091 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 4092 blocking mode, this indicates that the compatibility tests on  
 4093 dimensions and domains for the input arguments passed suc-  
 4094 cessfully. Either way, output vector **w** is ready to be used in the  
 4095 next method of the sequence.

4096 GrB\_PANIC Unknown internal error.

4097 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 4098 opaque GraphBLAS objects (input or output) is in an invalid  
 4099 state caused by a previous execution error. Call GrB\_error() to  
 4100 access any error messages generated by the implementation.

4101 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

4102 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
 4103 by a call to **new** (or **dup** for vector parameters).

4104 GrB\_INDEX\_OUT\_OF\_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-  
 4105 blocking mode, this error can be deferred.

4106 GrB\_DIMENSION\_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices**  $\neq$  **size(w)**.

4107 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with each  
 4108 other or the corresponding domains of the accumulation oper-  
 4109 ator, or the mask's domain is not compatible with **bool** (in the  
 4110 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

4111 GrB\_NULL\_POINTER Argument **row\_indices** is a NULL pointer.

## 4112 Description

4113 This variant of GrB\_extract computes the result of extracting a subset of locations from a Graph-  
 4114 BLAS vector in a specific order:  $w = u(\text{indices})$ ; or, if an optional binary accumulation operator

4115  $(\odot)$  is provided,  $w = w \odot u(\text{indices})$ . More explicitly:

$$4116 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4117 Logically, this operation occurs in three steps:

4118     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4119             and dimensions are tested for compatibility.

4120     **Compute** The indicated computations are carried out.

4121     **Output** The result is written into the output vector, possibly under control of a mask.

4122 Up to three argument vectors are used in this `GrB_extract` operation:

- 4123     1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4124     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4125     3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4126 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4127 bility as follows:

- 4128     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4129         must be from one of the pre-defined types of Table 3.2.
- 4130     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(u)$ .
- 4131     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4132         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4133         mulation operator.

4134 Two domains are compatible with each other if values from one domain can be cast to values in  
4135 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4136 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4137 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4138 error listed above is returned.

4139 From the arguments, the internal vectors, mask, and index array used in the computation are  
4140 formed ( $\leftarrow$  denotes copy):

- 4141     1. Vector  $\tilde{w} \leftarrow w$ .
- 4142     2. One-dimensional mask,  $\tilde{m}$ , is computed from argument `mask` as follows:  
4143         (a) If `mask` = `GrB_NULL`, then  $\tilde{m} = \langle \mathbf{size}(w), \{i, \forall i : 0 \leq i < \mathbf{size}(w)\} \rangle$ .

- 4144 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,
- 4145 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
- 4146 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
- 4147 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 4148 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 4149 4. The internal index array,  $\widetilde{\mathbf{I}}$ , is computed from argument indices as follows:
- 4150 (a) If  $\text{indices} = \text{GrB\_ALL}$ , then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .
- 4151 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4152 The internal vectors and mask are checked for dimension compatibility. The following conditions  
4153 must hold:

- 4154 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4155 2.  $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$ .

4156 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4157 match error listed above is returned.

4158 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4159 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4160 We are now ready to carry out the extract and any additional associated operations. We describe  
4161 this in terms of two intermediate vectors:

- 4162 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from  $\widetilde{\mathbf{u}}$  in their destination locations relative to  $\widetilde{\mathbf{w}}$ .
- 4163 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4164 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

4165 
$$\widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4166 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  
4167 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`  
4168 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4169 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4170 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 4171 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 4172 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

4173 
$$\widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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.

## 4240 Return Values

4241	<b>GrB_SUCCESS</b>	In blocking mode, the operation completed successfully. In non-
4242		blocking mode, this indicates that the compatibility tests on
4243		dimensions and domains for the input arguments passed suc-
4244		cessfully. Either way, output matrix C is ready to be used in the
4245		next method of the sequence.
4246	<b>GrB_PANIC</b>	Unknown internal error.
4247	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
4248		opaque GraphBLAS objects (input or output) is in an invalid
4249		state caused by a previous execution error. Call <code>GrB_error()</code> to
4250		access any error messages generated by the implementation.
4251	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
4252	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
4253		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4254	<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
4255		a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In
4256		non-blocking mode, this error can be deferred.
4257	<b>GrB_DIMENSION_MISMATCH</b>	Mask and C dimensions are incompatible, <code>nrows</code> $\neq$ <code>nrows(C)</code> , or
4258		<code>ncols</code> $\neq$ <code>ncols(C)</code> .
4259	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various matrices are incompatible with each
4260		other or the corresponding domains of the accumulation oper-
4261		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4262		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4263	<b>GrB_NULL_POINTER</b>	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4264		is a NULL pointer, or both.

## 4265 Description

4266 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified  
 4267 rows and columns of a GraphBLAS matrix in a specific order:  $C = A(\text{row\_indices}, \text{col\_indices})$ ; or,  
 4268 if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A(\text{row\_indices}, \text{col\_indices})$ .  
 4269 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4271 Logically, this operation occurs in three steps:

4272 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 4273 and dimensions are tested for compatibility.

4274 **Compute** The indicated computations are carried out.

4275 **Output** The result is written into the output matrix, possibly under control of a mask.

4276 Up to three argument matrices are used in the `GrB_extract` operation:

- 4277 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$   
4278 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)  
4279 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4280 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
4281 ibility as follows:

- 4282 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
4283 must be from one of the pre-defined types of Table 3.2.  
4284 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .  
4285 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4286 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4287 mulation operator.

4288 Two domains are compatible with each other if values from one domain can be cast to values in  
4289 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4290 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4291 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4292 error listed above is returned.

4293 From the arguments, the internal matrices, mask, and index arrays used in the computation are  
4294 formed ( $\leftarrow$  denotes copy):

- 4295 1. Matrix  $\tilde{C} \leftarrow C$ .  
4296 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:  
4297 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
4298  $j < \mathbf{ncols}(C)\} \rangle$ .  
4299 (b) If `Mask`  $\neq$  `GrB_NULL`,  
4300 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
4301  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,  
4302 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
4303  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .  
4304 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .  
4305 3. Matrix  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$ .

- 4306 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4307 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .
- 4308 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \text{nrows}$ .
- 4309 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4310 (a) If `col_indices` = `GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .
- 4311 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

4312 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4313 must hold:

- 4314 1.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$ .
- 4315 2.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$ .
- 4316 3.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$ .
- 4317 4.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$ .

4318 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4319 match error listed above is returned.

4320 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4321 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4322 We are now ready to carry out the extract and any additional associated operations. We describe  
4323 this in terms of two intermediate matrices:

- 4324 •  $\tilde{\mathbf{T}}$ : The matrix holding the extraction from  $\tilde{\mathbf{A}}$ .
- 4325 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4326 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

4327 
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{ (i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}}) \} \rangle.$$

4328 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  $\tilde{\mathbf{J}}$   
4329 array is not in the range  $[0, \text{ncols}(\tilde{\mathbf{A}}))$ , the execution of `GrB_extract` ends and the index out-of-  
4330 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
4331 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
4332 this point forward in the sequence.

4333 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4334 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .



- If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

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

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

$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}})),$$

$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{C}}) - (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}}))),$$

$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{T}}) - (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}}))),$$

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

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

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

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

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are copied into the result 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.

#### 4.3.6.3 extract: Column (and row) variant

Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the source matrix, elements of an arbitrary row of the matrix can be extracted with this function as well.

## 4364 C Syntax

```

4365         GrB_Info GrB_extract(GrB_Vector      w,
4366                             const GrB_Vector mask,
4367                             const GrB_BinaryOp accum,
4368                             const GrB_Matrix  A,
4369                             const GrB_Index  *row_indices,
4370                             GrB_Index        nrows,
4371                             GrB_Index        col_index,
4372                             const GrB_Descriptor desc);

```

## 4373 Parameters

4374     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4375     that may be accumulated with the result of the extract operation. On output, this  
4376     vector holds the results of the operation.

4377     **mask** (IN) An optional “write” mask that controls which results from this operation are  
4378     stored into the output vector **w**. The mask dimensions must match those of the  
4379     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4380     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4381     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4382     dimensions of **w**), **GrB\_NULL** should be specified.

4383     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4384     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4385     specified.

4386     **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4387     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations  
4388     within the specified column of **A** from which elements are extracted. If elements in  
4389     all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless  
4390     of execution mode and return value, this array may be manipulated by the caller  
4391     after this operation returns without affecting any deferred computations for this  
4392     operation.

4393     **nrows** (IN) The number of indices in the **row\_indices** array. Must be equal to **size(w)**.

4394     **col\_index** (IN) The index of the column of **A** from which to extract values. It must be in the  
4395     range  $[0, \mathbf{ncols}(A))$ .

4396     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4397     should be specified. Non-default field/value pairs are listed as follows:

4398

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

4399

## 4400 Return Values

4401	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
4402		blocking mode, this indicates that the compatibility tests on
4403		dimensions and domains for the input arguments passed suc-
4404		cessfully. Either way, output vector w is ready to be used in the
4405		next method of the sequence.
4406	GrB_PANIC	Unknown internal error.
4407	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
4408		opaque GraphBLAS objects (input or output) is in an invalid
4409		state caused by a previous execution error. Call GrB_error() to
4410		access any error messages generated by the implementation.
4411	GrB_OUT_OF_MEMORY	Not enough memory available for operation.
4412	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
4413		by a call to new (or dup for vector or matrix parameters).
4414	GrB_INVALID_INDEX	col_index is outside the allowable range (i.e., greater than ncols(A)).
4415	GrB_INDEX_OUT_OF_BOUNDS	A value in row_indices is greater than or equal to nrows(A). In
4416		non-blocking mode, this error can be deferred.
4417	GrB_DIMENSION_MISMATCH	mask and w dimensions are incompatible, or nrows $\neq$ size(w).
4418	GrB_DOMAIN_MISMATCH	The domains of the vector or matrix are incompatible with each
4419		other or the corresponding domains of the accumulation oper-
4420		ator, or the mask's domain is not compatible with bool (in the
4421		case where desc[GrB_MASK].GrB_STRUCTURE is not set).
4422	GrB_NULL_POINTER	Argument row_indices is a NULL pointer.

## 4423 Description

4424 This variant of GrB\_extract computes the result of extracting a subset of locations (in a specific  
4425 order) from a specified column of a GraphBLAS matrix:  $w = A(:, col\_index)(row\_indices)$ ; or, if

4426 an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot A(:, \text{col\_index})(\text{row\_indices})$ .  
 4427 More explicitly:

$$4428 \quad \begin{aligned} w(i) &= A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4429 Logically, this operation occurs in three steps:

4430     **Setup** The internal matrices, vectors, and mask used in the computation are formed and their  
 4431 domains and dimensions are tested for compatibility.

4432     **Compute** The indicated computations are carried out.

4433     **Output** The result is written into the output vector, possibly under control of a mask.

4434 Up to three argument vectors and matrices are used in this GrB\_extract operation:

- 4435 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4436 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4437 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4438 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain  
 4439 compatibility as follows:

- 4440 1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
 4441 must be from one of the pre-defined types of Table 3.2.
- 4442 2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(A)$ .
- 4443 3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4444 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4445 mulation operator.

4446 Two domains are compatible with each other if values from one domain can be cast to values in  
 4447 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4448 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4449 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
 4450 error listed above is returned.

4451 From the arguments, the internal vector, matrix, mask, and index array used in the computation  
 4452 are formed ( $\leftarrow$  denotes copy):

- 4453 1. Vector  $\tilde{w} \leftarrow w$ .
- 4454 2. One-dimensional mask,  $\tilde{m}$ , is computed from argument **mask** as follows:  
 4455 (a) If **mask** = GrB\_NULL, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .

4456 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
 4457 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
 4458 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
 4459 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
 4460 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .  
 4461 4. The internal row index array,  $\widetilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:  
 4462 (a) If `indices = GrB_ALL`, then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .  
 4463 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$ .

4464 The internal vector, `mask`, and index array are checked for dimension compatibility. The following  
 4465 conditions must hold:

- 4466 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4467 2.  $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$ .

4468 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4469 match error listed above is returned.

4470 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4471 1.  $0 \leq \text{col\_index} < \text{ncols}(\mathbf{A})$

4472 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above  
 4473 is returned.

4474 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4475 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4476 We are now ready to carry out the extract and any additional associated operations. We describe  
 4477 this in terms of two intermediate vectors:

- 4478 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from a column of  $\widetilde{\mathbf{A}}$ .
- 4479 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4480 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$4481 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col\_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col\_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4482 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`  
 4483 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,  
 4484 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result  
 4485 vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4486 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

4487 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

4488 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4489 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4490 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4491 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4492 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

$$4493 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4494 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4497 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4498 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
4499 using what is called a *standard vector mask and replace*. This is carried out under control of the  
4500 mask which acts as a “write mask”.

4501 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
4502 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4503 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4504 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
4505 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
4506 mask are unchanged:

$$4507 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4508 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4509 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4510 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
4511 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
4512 sequence.

### 4513 4.3.7 assign: Modifying sub-graphs

4514 Assign the contents of a subset of a matrix or vector.

#### 4515 4.3.7.1 assign: Standard vector variant

4516 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.  
4517 The size of the input vector is the same size as the index array provided.

## 4518 C Syntax

```
4519         GrB_Info GrB_assign(GrB_Vector      w,  
4520                             const GrB_Vector mask,  
4521                             const GrB_BinaryOp accum,  
4522                             const GrB_Vector u,  
4523                             const GrB_Index *indices,  
4524                             GrB_Index      nindices,  
4525                             const GrB_Descriptor desc);
```

## 4526 Parameters

4527        **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4528        that may be accumulated with the result of the assign operation. On output, this  
4529        vector holds the results of the operation.

4530        **mask** (IN) An optional “write” mask that controls which results from this operation are  
4531        stored into the output vector **w**. The mask dimensions must match those of the  
4532        vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4533        of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4534        in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4535        dimensions of **w**), **GrB\_NULL** should be specified.

4536        **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4537        entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4538        specified.

4539        **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4540        **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4541        **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0  
4542        to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4543        and return value, this array may be manipulated by the caller after this operation  
4544        returns without affecting any deferred computations for this operation. If this  
4545        array contains duplicate values, it implies in assignment of more than one value to  
4546        the same location which leads to undefined results.

4547        **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4548        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4549        should be specified. Non-default field/value pairs are listed as follows:  
4550

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

## Return Values

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

**GrB\_PANIC** Unknown internal error.

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

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

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

**GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indices** is greater than or equal to **size(w)**. In non-blocking mode, this can be reported as an execution error.

**GrB\_DIMENSION\_MISMATCH** mask and w dimensions are incompatible, or **nindices**  $\neq$  **size(u)**.

**GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

**GrB\_NULL\_POINTER** Argument **indices** is a NULL pointer.

## Description

This variant of **GrB\_assign** computes the result of assigning elements from a source GraphBLAS vector to a destination GraphBLAS vector in a specific order:  $w(\text{indices}) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w(\text{indices}) = w(\text{indices}) \odot u$ . More explicitly:

$$\begin{aligned}
 w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
 \end{aligned}$$



4579 Logically, this operation occurs in three steps:

4580     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4581             and dimensions are tested for compatibility.

4582     **Compute** The indicated computations are carried out.

4583     **Output** The result is written into the output vector, possibly under control of a mask.

4584 Up to three argument vectors are used in the GrB\_assign operation:

- 4585     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4586     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4587     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4588 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4589 bility as follows:

- 4590     1. If  $\mathbf{mask}$  is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\mathbf{mask})$   
4591         must be from one of the pre-defined types of Table 3.2.
- 4592     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4593     3. If  $\mathbf{accum}$  is not GrB\_NULL, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
4594         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
4595         mulation operator.

4596 Two domains are compatible with each other if values from one domain can be cast to values in  
4597 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4598 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4599 any compatibility rule above is violated, execution of GrB\_assign ends and the domain mismatch  
4600 error listed above is returned.

4601 From the arguments, the internal vectors, mask and index array used in the computation are formed  
4602 ( $\leftarrow$  denotes copy):

- 4603     1. Vector  $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 4604     2. One-dimensional mask,  $\widetilde{\mathbf{m}}$ , is computed from argument  $\mathbf{mask}$  as follows:
  - 4605         (a) If  $\mathbf{mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 4606         (b) If  $\mathbf{mask} \neq \text{GrB\_NULL}$ ,
    - 4607             i. If desc[GrB\_MASK].GrB\_STRUCTURE is set, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 4608             ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\text{bool})\mathbf{mask}(i) = \text{true}\} \rangle$ .
  - 4609         (c) If desc[GrB\_MASK].GrB\_COMP is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .

4610 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4611 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument indices as follows:

4612 (a) If `indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .

4613 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4614 The internal vector and mask are checked for dimension compatibility. The following conditions  
4615 must hold:

4616 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4617 2.  $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$ .

4618 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4619 match error listed above is returned.

4620 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4621 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4622 We are now ready to carry out the assign and any additional associated operations. We describe  
4623 this in terms of two intermediate vectors:

- 4624 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{w}}$ .
- 4625 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4626 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4627 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4628 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{w}}$ , computation  
4629 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`  
4630 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4631 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4632 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 4633 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$4634 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4635 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
4636 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  
4637 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4638 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4639 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4640 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4641 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4643 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,
```

```

4677         const GrB_Index      *row_indices,
4678         GrB_Index             nrows,
4679         const GrB_Index      *col_indices,
4680         GrB_Index             ncols,
4681         const GrB_Descriptor desc);

```

## 4682 Parameters

4683     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
4684     that may be accumulated with the result of the assign operation. On output, the  
4685     matrix holds the results of the operation.

4686     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
4687     stored into the output matrix **C**. The mask dimensions must match those of the  
4688     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4689     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
4690     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4691     dimensions of **C**), **GrB\_NULL** should be specified.

4692     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4693     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4694     specified.

4695     **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4696     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
4697     that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,  
4698     then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
4699     this array may be manipulated by the caller after this operation returns without  
4700     affecting any deferred computations for this operation. If this array contains du-  
4701     plicate values, it implies assignment of more than one value to the same location  
4702     which leads to undefined results.

4703     **nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(A)**  
4704     if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4705     **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns  
4706     of **C** that are assigned. If all columns of **C** are to be assigned in order from 0  
4707     to **ncols** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4708     and return value, this array may be manipulated by the caller after this operation  
4709     returns without affecting any deferred computations for this operation. If this  
4710     array contains duplicate values, it implies assignment of more than one value to  
4711     the same location which leads to undefined results.

4712     **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **ncols(A)** if **A** is  
4713     not transposed, or equal to **nrows(A)** if **A** is transposed.

4714  
4715  
4716

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:

4717

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.

## 4718 Return Values

4719  
4720  
4721  
4722  
4723

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.

4724

GrB\_PANIC Unknown internal error.

4725  
4726  
4727  
4728

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.

4729

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

4730  
4731

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).

4732  
4733  
4734

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.

4735  
4736

GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, nrows  $\neq$  nrows(A), or ncols  $\neq$  ncols(A).

4737  
4738  
4739  
4740

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).

4741  
4742

GrB\_NULL\_POINTER Either argument row\_indices is a NULL pointer, argument col\_indices is a NULL pointer, or both.

## 4743 Description

4744 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows  
 4745 and columns in `C` in a specified order:  $C(\text{row\_indices}, \text{col\_indices}) = A$ ; or, if an optional binary  
 4746 accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot$   
 4747 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4748 & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot A(i, j), \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4749 Logically, this operation occurs in three steps:

4750     Setup   The internal matrices and mask used in the computation are formed and their domains  
 4751             and dimensions are tested for compatibility.

4752     Compute   The indicated computations are carried out.

4753     Output   The result is written into the output matrix, possibly under control of a mask.

4754 Up to three argument matrices are used in the `GrB_assign` operation:

- 4755     1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4756     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)
- 4757     3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4758 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
 4759 ibility as follows:

- 4760     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 4761         must be from one of the pre-defined types of Table 3.2.
- 4762     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4763     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4764         of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4765         mulation operator.

4766 Two domains are compatible with each other if values from one domain can be cast to values in  
 4767 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4768 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4769 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 4770 error listed above is returned.

4771 From the arguments, the internal matrices, mask, and index arrays used in the computation are  
 4772 formed ( $\leftarrow$  denotes copy):

- 4773 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 4774 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
- 4775 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
4776  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 4777 (b) If `Mask  $\neq$  GrB_NULL`,
- 4778 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
4779  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
- 4780 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
4781  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
- 4782 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4783 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4784 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4785 (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4786 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4787 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4788 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
- 4789 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

4790 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4791 must hold:

- 4792 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 4793 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 4794 3.  $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$ .
- 4795 4.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$ .

4796 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4797 match error listed above is returned.

4798 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4799 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4800 We are now ready to carry out the assign and any additional associated operations. We describe  
4801 this in terms of two intermediate vectors:

- 4802 •  $\tilde{\mathbf{T}}$ : The matrix holding the contents from  $\tilde{\mathbf{A}}$  in their destination locations relative to  $\tilde{\mathbf{C}}$ .
- 4803 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4804 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$4805 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{( \tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j) ) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \} \}.$$

4806 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  
 4807  $\tilde{\mathbf{J}}$  array is not in the range  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-  
 4808 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
 4809 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
 4810 this point forward in the sequence.

4811 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 4812 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$4813 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4814 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})) \} \}.$$

4815 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 4816 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  
 4817 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

4818 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4819 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4820 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4821 \\ 4822 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4823 where the difference operator refers to set difference.

- 4824 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4825 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}}) \} \}.$$

4826 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4827 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4828 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4829 \\ 4830 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4831 \\ 4832 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4833 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4834 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 4835 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 4836 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

4870 bool or any of the predefined “built-in” types in Table 3.2. If the default mask  
 4871 is desired (i.e., a mask that is all true with the dimensions of a column of C),  
 4872 GrB\_NULL should be specified.

4873 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
 4874 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
 4875 specified.

4876 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column  
 4877 of C.

4878 **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
 4879 the specified column of C that are to be assigned. If all elements of the column  
 4880 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB\_ALL should  
 4881 be specified. Regardless of execution mode and return value, this array may be  
 4882 manipulated by the caller after this operation returns without affecting any de-  
 4883 ferred computations for this operation. If this array contains duplicate values, it  
 4884 implies in assignment of more than one value to the same location which leads to  
 4885 undefined results.

4886 **nrows** (IN) The number of values in **row\_indices** array. Must be equal to **size(u)**.

4887 **col\_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4888 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4889 should be specified. Non-default field/value pairs are listed as follows:

4890

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.

4891

## 4892 Return Values

4893 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4894 blocking mode, this indicates that the compatibility tests on  
 4895 dimensions and domains for the input arguments passed suc-  
 4896 cessfully. Either way, output matrix C is ready to be used in the  
 4897 next method of the sequence.

4898 **GrB\_PANIC** Unknown internal error.



4930 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4931 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
4932 compatibility as follows:

4933 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4934 must be from one of the pre-defined types of Table 3.2.

4935 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .

4936 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4937 of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4938 mulation operator.

4939 Two domains are compatible with each other if values from one domain can be cast to values in  
4940 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4941 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4942 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
4943 error listed above is returned.

4944 The `col_index` parameter is checked for a valid value. The following condition must hold:

4945 1.  $0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{C})$

4946 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
4947 is returned.

4948 From the arguments, the internal vectors, `mask`, and index array used in the computation are  
4949 formed ( $\leftarrow$  denotes copy):

4950 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a column of  $\mathbf{C}$  as follows:

4951 
$$\tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4952 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

4953 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$ .

4954 (b) If `mask`  $\neq$  `GrB_NULL`,

4955 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

4956 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

4957 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

4958 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4959 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:

4960 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .

4961 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i]$ ,  $\forall i : 0 \leq i < \text{nrows}$ .

4962 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
 4963 conditions must hold:

- 4964 1.  $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4965 2.  $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$ .

4966 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 4967 match error listed above is returned.

4968 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4969 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4970 We are now ready to carry out the assign and any additional associated operations. We describe  
 4971 this in terms of two intermediate vectors:

- 4972 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 4973 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4974 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4975 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4976 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
 4977 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
 4978 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
 4979 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

4980 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 4981 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$4982 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4983 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
 4984 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  
 4985 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4986 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 4987 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$4988 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4989 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4991 where the difference operator refers to set difference.

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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

$$z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

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

Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final result matrix,  $\mathbf{C}(:, \text{col\_index})$ . This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(:, \text{col\_index})$  on input to this operation are deleted and the new contents of the column is given by:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the column of the final result matrix,  $\mathbf{C}(:, \text{col\_index})$ , and elements of this column that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} \mathbf{L}(\mathbf{C}) = & \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \\ & \{(i, \text{col\_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ & \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

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

#### 4.3.7.4 assign: Row variant

Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the output cannot be transposed, a different variant of `assign` is provided to assign to a column of a matrix.

## 5022 C Syntax

```
5023         GrB_Info GrB_assign(GrB_Matrix      C,  
5024                             const GrB_Vector mask,  
5025                             const GrB_BinaryOp accum,  
5026                             const GrB_Vector u,  
5027                             GrB_Index      row_index,  
5028                             const GrB_Index *col_indices,  
5029                             GrB_Index      ncols,  
5030                             const GrB_Descriptor desc);
```

## 5031 Parameters

5032 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values  
5033 that may be accumulated with the result of the assign operation. On output, this  
5034 matrix holds the results of the operation.

5035 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5036 stored into the specified row of the output matrix **C**. The mask dimensions must  
5037 match those of a single row of the matrix **C**. If the **GrB\_STRUCTURE** descriptor  
5038 is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or  
5039 any of the predefined “built-in” types in Table 3.2. If the default mask is desired  
5040 (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB\_NULL** should  
5041 be specified.

5042 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
5043 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5044 specified.

5045 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of  
5046 **C**.

5047 **row\_index** (IN) The index of the row in **C** to assign. Must be in the range  $[0, \mathbf{nrows}(\mathbf{C})]$ .

5048 **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5049 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to  
5050 be assigned in order from index 0 to  $\mathbf{ncols} - 1$ , then **GrB\_ALL** should be specified.  
5051 Regardless of execution mode and return value, this array may be manipulated by  
5052 the caller after this operation returns without affecting any deferred computations  
5053 for this operation. If this array contains duplicate values, it implies in assignment  
5054 of more than one value to the same location which leads to undefined results.

5055 **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **size(u)**.

5056 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5057 should be specified. Non-default field/value pairs are listed as follows:  
5058

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:



5087  $C(\text{row\_index}, :) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_index}, :$   
 5088  $) = C(\text{row\_index}, :) \odot u$ . Taking order of `col_indices` into account it is more explicitly written as:

5089  $C(\text{row\_index}, \text{col\_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$   
 $C(\text{row\_index}, \text{col\_indices}[j]) = C(\text{row\_index}, \text{col\_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$

5090 Logically, this operation occurs in three steps:

5091     **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
 5092             domains and dimensions are tested for compatibility.

5093     **Compute** The indicated computations are carried out.

5094     **Output** The result is written into the output matrix, possibly under control of a mask.

5095 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5096     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5097     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 5098     3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5099 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
 5100 compatibility as follows:

- 5101     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 5102         must be from one of the pre-defined types of Table 3.2.
- 5103     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(u)$ .
- 5104     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5105         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 5106         mulation operator.

5107 Two domains are compatible with each other if values from one domain can be cast to values in  
 5108 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5109 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5110 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 5111 error listed above is returned.

5112 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5113     1.  $0 \leq \text{row\_index} < \mathbf{nrows}(C)$

5114 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
 5115 is returned.

5116 From the arguments, the internal vectors, mask, and index array used in the computation are  
 5117 formed ( $\leftarrow$  denotes copy):

5118 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a row of  $\mathbf{C}$  as follows:

$$5119 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5120 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

5121 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$ .

5122 (b) If `mask  $\neq$  GrB_NULL`,

5123 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

5124 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

5125 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

5126 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5127 4. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:

5128 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .

5129 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

5130 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
5131 conditions must hold:

5132 1.  $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5133 2.  $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$ .

5134 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
5135 match error listed above is returned.

5136 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5137 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5138 We are now ready to carry out the assign and any additional associated operations. We describe  
5139 this in terms of two intermediate vectors:

- 5140 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 5141 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5142 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$5143 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5144 At this point, if any value of  $\tilde{\mathbf{J}}[j]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
5145 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
5146 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
5147 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

5148 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

5149 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5150 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5151 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
5152 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  
5153 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5154 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5155 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$5156 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5157 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5159 where the difference operator refers to set difference.

5160 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5161 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5162 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5163 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5164 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5165 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5166 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5169 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5170 Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final  
5171 result matrix,  $\mathbf{C}(\text{row\_index}, :)$ . This is carried out under control of the mask which acts as a “write  
5172 mask”.

5173 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(\text{row\_index}, :)$  on input to this  
5174 operation are deleted and the new contents of the column is given by:

$$5175 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5176 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
5177 copied into the column of the final result matrix,  $\mathbf{C}(\text{row\_index}, :)$ , and elements of this column  
5178 that fall outside the set indicated by the mask are unchanged:

$$5179 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup$$

$$5180 \quad \{(\text{row\_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5181 \quad \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5182 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5183 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5184 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may  
5185 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

5186 **4.3.7.5 assign: Constant vector variant**[Scott: NEW CONTENT]

5187 Assign the same value to a specified subset of vector elements. With the use of GrB\_ALL, the entire  
5188 destination vector can be filled with the constant.

5189 **C Syntax**

```
5190         GrB_Info GrB_assign(GrB_Vector          w,  
5191                             const GrB_Vector    mask,  
5192                             const GrB_BinaryOp   accum,  
5193                             <type>              val,  
5194                             const GrB_Index     *indices,  
5195                             GrB_Index           nindices,  
5196                             const GrB_Descriptor desc);
```

```
5197         GrB_Info GrB_assign(GrB_Vector          w,  
5198                             const GrB_Vector    mask,  
5199                             const GrB_BinaryOp   accum,  
5200                             const GrB_Scalar     s,  
5201                             const GrB_Index     *indices,  
5202                             GrB_Index           nindices,  
5203                             const GrB_Descriptor desc);
```

5204 **Parameters**

5205 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5206 that may be accumulated with the result of the assign operation. On output, this  
5207 vector holds the results of the operation.

5208 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5209 stored into the output vector w. The mask dimensions must match those of the  
5210 vector w. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
5211 of the mask vector must be of type bool or any of the predefined “built-in” types  
5212 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
5213 dimensions of w), GrB\_NULL should be specified.

5214 **accum** (IN) An optional binary operator used for accumulating entries into existing w  
5215 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
5216 specified.

5217 **val** (IN) Scalar value to assign to (a subset of) w.

5218 **s** (IN) Scalar value to assign to (a subset of) w.

5219 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5220 w that are to be assigned. If all elements of w are to be assigned in order from 0

5221 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode  
5222 and return value, this array may be manipulated by the caller after this operation  
5223 returns without affecting any deferred computations for this operation. In this  
5224 variant, the specific order of the values in the array has no effect on the result.  
5225 Unlike other variants, if there are duplicated values in this array the result is still  
5226 defined.

5227 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If  
5228 `nindices` is zero, the operation becomes a NO-OP.

5229 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
5230 should be specified. Non-default field/value pairs are listed as follows:

5231

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> .

5232

## 5233 Return Values

5234 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5235 blocking mode, this indicates that the compatibility tests on  
5236 dimensions and domains for the input arguments passed suc-  
5237 cessfully. Either way, output vector `w` is ready to be used in the  
5238 next method of the sequence.

5239 **GrB\_PANIC** Unknown internal error.

5240 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
5241 opaque GraphBLAS objects (input or output) is in an invalid  
5242 state caused by a previous execution error. Call `GrB_error()` to  
5243 access any error messages generated by the implementation.

5244 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5245 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
5246 by a call to `new` (or `dup` for vector parameters).

5247 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-  
5248 blocking mode, this can be reported as an execution error.

5249 **GrB\_DIMENSION\_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less  
5250 than `size(w)`.



5281 4. If **accum** is not **GrB\_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the  
 5282 method, must be compatible with **D<sub>in2</sub>(accum)** of the accumulation operator.

5283 Two domains are compatible with each other if values from one domain can be cast to values in  
 5284 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5285 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5286 any compatibility rule above is violated, execution of **GrB\_assign** ends and the domain mismatch  
 5287 error listed above is returned.

5288 From the arguments, the internal vectors, mask and index array used in the computation are formed  
 5289 ( $\leftarrow$  denotes copy):

- 5290 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5291 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:
  - 5292 (a) If **mask** = **GrB\_NULL**, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 5293 (b) If **mask**  $\neq$  **GrB\_NULL**,
    - 5294 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 5295 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 5296 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5297 3. Scalar  $\tilde{s} \leftarrow \mathbf{s}$  (**GrB\_Scalar** version only).
- 5298 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument **indices** as follows:
  - 5299 (a) If **indices** = **GrB\_ALL**, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$ .
  - 5300 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$ .

5301 The internal vector and mask are checked for dimension compatibility. The following conditions  
 5302 must hold:

- 5303 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5304 2.  $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$ .

5305 If any compatibility rule above is violated, execution of **GrB\_assign** ends and the dimension mis-  
 5306 match error listed above is returned.

5307 From this point forward, in **GrB\_NONBLOCKING** mode, the method can optionally exit with  
 5308 **GrB\_SUCCESS** return code and defer any computation and/or execution error codes.

5309 We are now ready to carry out the assign and any additional associated operations. We describe  
 5310 this in terms of two intermediate vectors:

- 5311 •  $\tilde{\mathbf{t}}$ : The vector holding the copies of the scalar, either **val** or  $\tilde{s}$ , in their destination locations  
 5312 relative to  $\tilde{\mathbf{w}}$ .

5313 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5314 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows. If a non-opaque scalar  $\mathbf{val}$  is provided:

$$5315 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5316 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$5317 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5318 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$5319 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5320 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  
 5321 is not in the range  $[0, \mathbf{size}(\tilde{\mathbf{w}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds  
 5322 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a  
 5323 sequence-terminating `GrB_wait()` is called. Regardless, the result vector,  $\mathbf{w}$ , is invalid from this  
 5324 point forward in the sequence.

5325 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

5326 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5327 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5328 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
 5329 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  
 5330 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5331 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5332 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5333 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5334 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5336 where the difference operator refers to set difference. We note that in this case of assigning  
 5337 a constant,  $\{\tilde{\mathbf{I}}[k], \forall k\}$  and  $\mathbf{ind}(\tilde{\mathbf{t}})$  are identical.

5338 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5339 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5340 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5341 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5342 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$5343 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5344 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5347 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.



5348 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 5349 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 5350 mask which acts as a “write mask”.

- 5351 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 5352 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5353 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5354 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 5355 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 5356 mask are unchanged:

$$5357 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5358 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 5359 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 5360 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 5361 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 5362 sequence.

#### 5363 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5364 Assign the same value to a specified subset of matrix elements. With the use of GrB\_ALL, the  
 5365 entire destination matrix can be filled with the constant.

### 5366 C Syntax

```
5367     GrB_Info GrB_assign(GrB_Matrix      C,
5368                        const GrB_Matrix Mask,
5369                        const GrB_BinaryOp accum,
5370                        <type>          val,
5371                        const GrB_Index  *row_indices,
5372                        GrB_Index        nrows,
5373                        const GrB_Index  *col_indices,
5374                        GrB_Index        ncols,
5375                        const GrB_Descriptor desc);
```

```
5376     GrB_Info GrB_assign(GrB_Matrix      C,
5377                        const GrB_Matrix Mask,
5378                        const GrB_BinaryOp accum,
5379                        const GrB_Scalar  s,
5380                        const GrB_Index  *row_indices,
5381                        GrB_Index        nrows,
```

```

5382         const GrB_Index      *col_indices,
5383         GrB_Index             ncols,
5384         const GrB_Descriptor  desc);

```

## 5385 Parameters

5386       **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
5387       that may be accumulated with the result of the assign operation. On output, the  
5388       matrix holds the results of the operation.

5389       **Mask** (IN) An optional “write” mask that controls which results from this operation are  
5390       stored into the output matrix **C**. The mask dimensions must match those of the  
5391       matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5392       of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
5393       in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5394       dimensions of **C**), **GrB\_NULL** should be specified.

5395       **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
5396       entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5397       specified.

5398       **val** (IN) Scalar value to assign to (a subset of) **C**.

5399       **s** (IN) Scalar value to assign to (a subset of) **C**.

5400       **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
5401       that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,  
5402       then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
5403       this array may be manipulated by the caller after this operation returns without  
5404       affecting any deferred computations for this operation. Unlike other variants, if  
5405       there are duplicated values in this array the result is still defined.

5406       **nrows** (IN) The number of values in **row\_indices** array. Must be in the range: [0, **nrows**(**C**)].  
5407       If **nrows** is zero, the operation becomes a NO-OP.

5408       **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**  
5409       that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** – 1,  
5410       then **GrB\_ALL** should be specified. Regardless of execution mode and return value,  
5411       this array may be manipulated by the caller after this operation returns without  
5412       affecting any deferred computations for this operation. Unlike other variants, if  
5413       there are duplicated values in this array the result is still defined.

5414       **ncols** (IN) The number of values in **col\_indices** array. Must be in the range: [0, **ncols**(**C**)].  
5415       If **ncols** is zero, the operation becomes a NO-OP.

5416       **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5417       should be specified. Non-default field/value pairs are listed as follows:

5418

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`

5448 or  $C(\text{row\_indices}, \text{col\_indices}) = s$  is performed. If an optional binary accumulation operator  
 5449  $(\odot)$  is provided, then either  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot \text{val}$  or  
 5450  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot s$  is performed. More explicitly, if a  
 5451 non-opaque value  $\text{val}$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = \text{val}, \text{ or} \\ 5452 \quad & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5453 Correspondingly, if a `GrB_Scalar`  $s$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = s, \text{ or} \\ 5454 \quad & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5455 Logically, this operation occurs in three steps:

5456     Setup The internal vectors and mask used in the computation are formed and their domains  
 5457             and dimensions are tested for compatibility.

5458     Compute The indicated computations are carried out.

5459     Output The result is written into the output matrix, possibly under control of a mask.

5460 Up to two argument matrices are used in the `GrB_assign` operation:

- 5461     1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5462     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)

5463 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain  
 5464 compatibility as follows:

- 5465     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 5466         must be from one of the pre-defined types of Table 3.2.
- 5467     2.  $\mathbf{D}(C)$  must be compatible with either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5468         method.
- 5469     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5470         of the accumulation operator.
- 5471     4. If `accum` is not `GrB_NULL`, then either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5472         method, must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

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

From the arguments, the internal matrices, index arrays, and mask used in the computation are formed ( $\leftarrow$  denotes copy):

1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
  - (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - (b) If `Mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
3. Scalar  $\tilde{s} \leftarrow s$  (`GrB_Scalar` version only).
4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
  - (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
  - (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
  - (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

The internal matrix and mask are checked for dimension compatibility. The following conditions must hold:

1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
3.  $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$ .
4.  $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$ .

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

5505 From this point forward, in GrB\_NONBLOCKING mode, the method can optionally exit with  
 5506 GrB\_SUCCESS return code and defer any computation and/or execution error codes.

5507 We are now ready to carry out the assign and any additional associated operations. We describe  
 5508 this in terms of two intermediate matrices:

- 5509 •  $\tilde{\mathbf{T}}$ : The matrix holding the copies of the scalar, either  $\mathbf{val}$  or  $\tilde{s}$ , in their destination locations  
 5510 relative to  $\tilde{\mathbf{C}}$ .
- 5511 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5512 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows. If a non-opaque scalar  $\mathbf{val}$  is provided:

$$5513 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \mathbf{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5514 Correspondingly, if a non-empty GrB\_Scalar  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$5515 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \mathbf{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5516 Finally, if an empty GrB\_Scalar  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$5517 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5518 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  
 5519 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  
 5520  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of GrB\_assign ends and the index out-of-bounds error listed above is  
 5521 generated. In GrB\_NONBLOCKING mode, the error can be deferred until a sequence-terminating  
 5522 GrB\_wait() is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from this point forward in the  
 5523 sequence.

5524 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 5525 • If  $\mathbf{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}}$  is defined as

$$5526 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5527 \quad \{(i, j, Z_{ij}) \mid (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}}))\} \rangle.$$

5528 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 5529 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  
 5530 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

5531 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 5532 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5533 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5534 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \\ 5535$$

5536 where the difference operator refers to set difference. We note that, in this particular case of  
 5537 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.

5538 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5539 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5540 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5541 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5542 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5543 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5544 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5547 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5548 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5549 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5550 mask which acts as a “write mask”.

5551 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5552 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5553 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5554 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
5555 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
5556 mask are unchanged:

$$5557 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5558 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5559 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5560 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
5561 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5562 sequence.

#### 5563 4.3.8 apply: Apply a function to the elements of an object

5564 Computes the transformation of the values of the elements of a vector or a matrix using a unary  
5565 function, or a binary function where one argument is bound to a scalar.

##### 5566 4.3.8.1 apply: Vector variant

5567 Computes the transformation of the values of the elements of a vector using a unary function.

## 5568 C Syntax

```

5569         GrB_Info GrB_apply(GrB_Vector          w,
5570                             const GrB_Vector    mask,
5571                             const GrB_BinaryOp    accum,
5572                             const GrB_UnaryOp     op,
5573                             const GrB_Vector     u,
5574                             const GrB_Descriptor  desc);

```

## 5575 Parameters

5576     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5577     that may be accumulated with the result of the apply operation. On output, this  
5578     vector holds the results of the operation.

5579     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5580     stored into the output vector **w**. The mask dimensions must match those of the  
5581     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5582     of the mask vector must be of type **bool** or any of the predefined “built-in” types  
5583     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5584     dimensions of **w**), **GrB\_NULL** should be specified.

5585     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5586     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5587     specified.

5588     **op** (IN) A unary operator applied to each element of input vector **u**.

5589     **u** (IN) The GraphBLAS vector to which the unary function is applied.

5590     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5591     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> .

## 5594 Return Values

5595     **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5596     blocking mode, this indicates that the compatibility tests on di-  
5597     mensions and domains for the input arguments passed successfully.



5598 Either way, output vector  $w$  is ready to be used in the next method  
 5599 of the sequence.

5600 **GrB\_PANIC** Unknown internal error.

5601 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 5602 GraphBLAS objects (input or output) is in an invalid state caused  
 5603 by a previous execution error. Call **GrB\_error()** to access any error  
 5604 messages generated by the implementation.

5605 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5606 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 5607 a call to **new** (or **dup** for vector parameters).

5608 **GrB\_DIMENSION\_MISMATCH**  $mask$ ,  $w$  and/or  $u$  dimensions are incompatible.

5609 **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the corre-  
 5610 sponding domains of the accumulation operator or unary function,  
 5611 or the mask's domain is not compatible with **bool** (in the case where  
 5612  $desc[GrB\_MASK].GrB\_STRUCTURE$  is not set).

## 5613 Description

5614 This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a  
 5615 GraphBLAS vector:  $w = f(u)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
 5616  $w = w \odot f(u)$ .

5617 Logically, this operation occurs in three steps:

5618 **Setup** The internal vectors and mask used in the computation are formed and their domains  
 5619 and dimensions are tested for compatibility.

5620 **Compute** The indicated computations are carried out.

5621 **Output** The result is written into the output vector, possibly under control of a mask.

5622 Up to three argument vectors are used in this **GrB\_apply** operation:

- 5623 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5624 2.  $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$  (optional)
- 5625 3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5626 The argument vectors, unary operator and the accumulation operator (if provided) are tested for  
 5627 domain compatibility as follows:

- 5628 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
5629 must be from one of the pre-defined types of Table 3.2.
- 5630 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5631 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5632 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5633  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5634 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in}(\text{op})$ .

5635 Two domains are compatible with each other if values from one domain can be cast to values in  
5636 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5637 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5638 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
5639 error listed above is returned.

5640 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
5641 denotes copy):

- 5642 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5643 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 5644 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 5645 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 5646 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 5647 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 5648 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5649 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5650 The internal vectors and masks are checked for dimension compatibility. The following conditions  
5651 must hold:

- 5652 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5653 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

5654 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5655 error listed above is returned.

5656 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5657 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5658 We are now ready to carry out the apply and any additional associated operations. We describe  
5659 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	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 <b>Mask</b> matrix. 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 A for the operation.

## 5720 Return Values

5721	<b>GrB_SUCCESS</b>	In blocking mode, the operation completed successfully. In non-
5722		blocking mode, this indicates that the compatibility tests on
5723		dimensions and domains for the input arguments passed suc-
5724		cessfully. Either way, output matrix C is ready to be used in the
5725		next method of the sequence.
5726	<b>GrB_PANIC</b>	Unknown internal error.
5727	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
5728		opaque GraphBLAS objects (input or output) is in an invalid
5729		state caused by a previous execution error. Call <b>GrB_error()</b> to
5730		access any error messages generated by the implementation.
5731	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
5732	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
5733		by a call to <b>new</b> (or <b>Matrix_dup</b> for matrix parameters).
5734	<b>GrB_DIMENSION_MISMATCH</b>	Mask and C dimensions are incompatible, <b>nrows</b> $\neq$ <b>nrows</b> (C), or
5735		<b>ncols</b> $\neq$ <b>ncols</b> (C).
5736	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various matrices are incompatible with the
5737		corresponding domains of the accumulation operator or unary
5738		function, or the mask's domain is not compatible with <b>bool</b> (in
5739		the case where <b>desc</b> [ <b>GrB_MASK</b> ]. <b>GrB_STRUCTURE</b> is not set).

## 5740 Description

5741 This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a  
 5742 GraphBLAS matrix:  $C = f(A)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
 5743  $C = C \odot f(A)$ .

5744 Logically, this operation occurs in three steps:

5745     **Setup** The internal matrices and mask used in the computation are formed and their domains  
 5746     and dimensions are tested for compatibility.

5747     **Compute** The indicated computations are carried out.

5748     **Output** The result is written into the output matrix, possibly under control of a mask.

5749 Up to three argument matrices are used in the **GrB\_apply** operation:

- 5750 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5751 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)

5752 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5753 The argument matrices, unary operator and the accumulation operator (if provided) are tested for  
5754 domain compatibility as follows:

- 5755 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
5756 must be from one of the pre-defined types of Table 3.2.
- 5757 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5758 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5759 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5760  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5761 4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in}(\text{op})$  of the unary operator.

5762 Two domains are compatible with each other if values from one domain can be cast to values in  
5763 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5764 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5765 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
5766 error listed above is returned.

5767 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
5768 are formed ( $\leftarrow$  denotes copy):

- 5769 1. Matrix  $\tilde{C} \leftarrow C$ .
- 5770 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument **Mask** as follows:
  - 5771 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
5772  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 5773 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 5774 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
5775  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 5776 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
5777  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 5778 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .
- 5779 3. Matrix  $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$ .

5780 The internal matrices and mask are checked for dimension compatibility. The following conditions  
5781 must hold:

- 5782 1.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$ .
- 5783 2.  $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$ .
- 5784 3.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$ .

5785 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

5786 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5787 error listed above is returned.

5788 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5789 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5790 We are now ready to carry out the apply and any additional associated operations. We describe  
5791 this in terms of two intermediate matrices:

- 5792 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the unary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 5793 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5794 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$5795 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5796 where  $f = \mathbf{f}(\mathbf{op})$ .

5797 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 5798 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 5799 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5800 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5801 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5802 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 5803 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 5804 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5805 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5806 \quad & \\ 5807 \end{aligned}$$

5808 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5809 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5810 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5811 mask which acts as a “write mask”.

- 5812 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5813 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5814 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 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

```

5832 // bind-first + scalar value
5833 GrB_Info GrB_apply(GrB_Vector          w,
5834                   const GrB_Vector     mask,
5835                   const GrB_BinaryOp    accum,
5836                   const GrB_BinaryOp    op,
5837                   <type>                val,
5838                   const GrB_Vector     u,
5839                   const GrB_Descriptor  desc);

5840 // bind-first + GraphBLAS scalar
5841 GrB_Info GrB_apply(GrB_Vector          w,
5842                   const GrB_Vector     mask,
5843                   const GrB_BinaryOp    accum,
5844                   const GrB_BinaryOp    op,
5845                   const GrB_Scalar      s,
5846                   const GrB_Vector     u,
5847                   const GrB_Descriptor  desc);

5848 // bind-second + scalar value
5849 GrB_Info GrB_apply(GrB_Vector          w,
5850                   const GrB_Vector     mask,
```



```

5851         const GrB_BinaryOp      accum,
5852         const GrB_BinaryOp      op,
5853         const GrB_Vector        u,
5854         <type>                  val,
5855         const GrB_Descriptor    desc);

5856 // bind-second + GraphBLAS scalar
5857 GrB_Info GrB_apply(GrB_Vector      w,
5858                   const GrB_Vector mask,
5859                   const GrB_BinaryOp accum,
5860                   const GrB_BinaryOp op,
5861                   const GrB_Vector u,
5862                   const GrB_Scalar s,
5863                   const GrB_Descriptor desc);

```

## 5864 Parameters

5865     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5866     that may be accumulated with the result of the apply operation. On output, this  
5867     vector holds the results of the operation.

5868     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5869     stored into the output vector **w**. The mask dimensions must match those of the  
5870     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5871     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5872     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5873     dimensions of **w**), **GrB\_NULL** should be specified.

5874     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5875     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5876     specified.

5877     **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar  
5878     value, **val**.

5879     **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as  
5880     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
5881     argument in the *bind-second* variant.

5882     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
5883     argument in the *bind-first* variant, or the right-hand (second) argument in the  
5884     *bind-second* variant.

5885     **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand  
5886     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
5887     the *bind-second* variant. It must not be empty.

5888 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
5889 should be specified. Non-default field/value pairs are listed as follows:

5890

	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.
5891	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> .

## 5892 Return Values

5893 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
5894 blocking mode, this indicates that the compatibility tests on di-  
5895 mensions and domains for the input arguments passed successfully.  
5896 Either way, output vector **w** is ready to be used in the next method  
5897 of the sequence.

5898 GrB\_PANIC Unknown internal error.

5899 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
5900 GraphBLAS objects (input or output) is in an invalid state caused  
5901 by a previous execution error. Call GrB\_error() to access any error  
5902 messages generated by the implementation.

5903 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

5904 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
5905 a call to new (or dup for vector parameters).

5906 GrB\_DIMENSION\_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

5907 GrB\_DOMAIN\_MISMATCH The domains of the various vectors and scalar are incompatible with  
5908 the corresponding domains of the binary operator or accumulation  
5909 operator, or the **mask**'s domain is not compatible with **bool** (in the  
5910 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

5911 GrB\_EMPTY\_OBJECT The GrB\_Scalar **s** used in the call is empty (**nvals(s) = 0**) and  
5912 therefore a value cannot be passed to the binary operator.

## 5913 Description

5914 This variant of GrB\_apply computes the result of applying a binary operator to the elements of a  
5915 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5916                   bind-first:      $w = f(\text{val}, u)$  or  $w = f(s, u)$

5917                   bind-second:     $w = f(u, \text{val})$  or  $w = f(u, s)$ ,

5918 or if an optional binary accumulation operator ( $\odot$ ) is provided:

5919                   bind-first:      $w = w \odot f(\text{val}, u)$  or  $w = w \odot f(s, u)$

5920                   bind-second:     $w = w \odot f(u, \text{val})$  or  $w = w \odot f(u, s)$ .

5921 Logically, this operation occurs in three steps:

5922     **Setup** The internal vectors and mask used in the computation are formed and their domains  
5923             and dimensions are tested for compatibility.

5924     **Compute** The indicated computations are carried out.

5925     **Output** The result is written into the output vector, possibly under control of a mask.

5926 Up to three argument vectors are used in this GrB\_apply operation:

- 5927     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$   
5928     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)  
5929     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5930 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are  
5931 tested for domain compatibility as follows:

- 5932     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
5933         must be from one of the pre-defined types of Table 3.2.
- 5934     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.
- 5935     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5936         of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
5937          $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5938     4.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.
- 5939     5. If bind-first:
- 5940         (a)  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.
- 5941         (b) If the non-opaque scalar **val** is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
5942             of the binary operator.
- 5943         (c) If the GrB\_Scalar **s** is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
5944             binary operator.

- 5945 6. If bind-second:
- 5946 (a)  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the binary operator.
- 5947 (b) If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$
- 5948 of the binary operator.
- 5949 (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
- 5950 binary operator.

5951 Two domains are compatible with each other if values from one domain can be cast to values in

5952 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5953 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5954 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5955 error listed above is returned.

5956 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$

5957 denotes copy):

- 5958 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5959 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 5960 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 5961 (b) If `mask  $\neq$  GrB_NULL`,
- 5962 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 5963 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 5964 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5965 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 5966 4. Scalar  $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$  (GraphBLAS scalar case).

5967 The internal vectors and masks are checked for dimension compatibility. The following conditions

5968 must hold:

- 5969 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5970 2.  $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$ .

5971 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5972 error listed above is returned.

5973 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5974 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5975 If an empty `GrB_Scalar`  $\tilde{\mathbf{s}}$  is provided ( $\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.

5976 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

5977 `val` with the same domain as  $\tilde{\mathbf{s}}$  and set `val = val( $\tilde{\mathbf{s}}$ )`.

5978 We are now ready to carry out the apply and any additional associated operations. We describe

5979 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  $\text{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- If  $\text{accum}$  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  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  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  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  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.

#### 6013 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

6014 Computes the transformation of the values of the stored elements of a matrix using a binary  
6015 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the  
6016 first argument to the binary operator and stored elements of the matrix are passed as the second  
6017 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument  
6018 and the specified scalar value is passed as the second argument. The scalar can be passed either as  
6019 a non-opaque variable or as a GrB\_Scalar object.

#### 6020 C Syntax

```
6021 // bind-first + scalar value
6022 GrB_Info GrB_apply(GrB_Matrix      C,
6023                   const GrB_Matrix Mask,
6024                   const GrB_BinaryOp accum,
6025                   const GrB_BinaryOp op,
6026                   <type>           val,
6027                   const GrB_Matrix A,
6028                   const GrB_Descriptor desc);
```

```
6029 // bind-first + GraphBLAS scalar
6030 GrB_Info GrB_apply(GrB_Matrix      C,
6031                   const GrB_Matrix Mask,
6032                   const GrB_BinaryOp accum,
6033                   const GrB_BinaryOp op,
6034                   const GrB_Scalar s,
6035                   const GrB_Matrix A,
6036                   const GrB_Descriptor desc);
```

```
6037 // bind-second + scalar value
6038 GrB_Info GrB_apply(GrB_Matrix      C,
6039                   const GrB_Matrix Mask,
6040                   const GrB_BinaryOp accum,
6041                   const GrB_BinaryOp op,
6042                   const GrB_Matrix A,
6043                   <type>           val,
6044                   const GrB_Descriptor desc);
```

```
6045 // bind-second + GraphBLAS scalar
6046 GrB_Info GrB_apply(GrB_Matrix      C,
6047                   const GrB_Matrix Mask,
6048                   const GrB_BinaryOp accum,
6049                   const GrB_BinaryOp op,
6050                   const GrB_Matrix A,
```

```

6051         const GrB_Scalar      s,
6052         const GrB_Descriptor desc);

```

## 6053 Parameters

6054     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6055     that may be accumulated with the result of the apply operation. On output, the  
6056     matrix holds the results of the operation.

6057     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6058     stored into the output matrix C. The mask dimensions must match those of the  
6059     matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
6060     of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
6061     in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
6062     dimensions of C), `GrB_NULL` should be specified.

6063     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
6064     entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
6065     specified.

6066     **op** (IN) A binary operator applied to each element of input matrix, A, with the element  
6067     of the input matrix used as the left-hand argument, and the scalar value, `val`, used  
6068     as the right-hand argument.

6069     **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as  
6070     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
6071     argument in the *bind-second* variant.

6072     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
6073     argument in the *bind-first* variant, or the right-hand (second) argument in the  
6074     *bind-second* variant.

6075     **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand  
6076     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
6077     the *bind-second* variant. It must not be empty.

6078     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6079     should be specified. Non-default field/value pairs are listed as follows:  
6080

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.



## 6108 Description

6109 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a  
6110 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6111                   bind-first:      $C = f(\text{val}, A)$  or  $C = f(s, A)$

6112                   bind-second:     $C = f(A, \text{val})$  or  $C = f(A, s)$ ,

6113 or if an optional binary accumulation operator ( $\odot$ ) is provided:

6114                   bind-first:      $C = C \odot f(\text{val}, A)$  or  $C = C \odot f(s, A)$

6115                   bind-second:     $C = C \odot f(A, \text{val})$  or  $C = C \odot f(A, s)$ .

6116 Logically, this operation occurs in three steps:

6117       **Setup** The internal matrices and mask used in the computation are formed and their domains  
6118               and dimensions are tested for compatibility.

6119       **Compute** The indicated computations are carried out.

6120       **Output** The result is written into the output matrix, possibly under control of a mask.

6121 Up to three argument matrices are used in the `GrB_apply` operation:

6122     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

6123     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)

6124     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6125 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are  
6126 tested for domain compatibility as follows:

6127     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
6128       must be from one of the pre-defined types of Table 3.2.

6129     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.

6130     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6131       of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
6132        $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

6133     4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

6134     5. If bind-first:

6135       (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

6136 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
 6137 of the binary operator.

6138 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
 6139 binary operator.

6140 6. If `bind-second`:

6141 (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

6142 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
 6143 of the binary operator.

6144 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the  
 6145 binary operator.

6146 Two domains are compatible with each other if values from one domain can be cast to values in  
 6147 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6148 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6149 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
 6150 error listed above is returned.

6151 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
 6152 are formed ( $\leftarrow$  denotes copy):

6153 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .

6154 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:

6155 (a) If `Mask` = `GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 6156  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .

6157 (b) If `Mask`  $\neq$  `GrB_NULL`,

6158 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
 6159  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,

6160 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
 6161  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .

6162 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .

6163 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument `A` as follows:

6164 `bind-first:`  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? A^T : A$

6165 `bind-second:`  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$

6166 4. Scalar  $\tilde{s} \leftarrow s$  (`GraphBLAS` scalar case).

6167 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 6168 must hold:

6169 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .

6170 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .

6171 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6172 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6173 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6174 error listed above is returned.

6175 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6176 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6177 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6178 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6179  $\mathbf{val}$  with the same domain as  $\tilde{s}$  and set  $\mathbf{val} = \mathbf{val}(\tilde{s})$ .

6180 We are now ready to carry out the apply and any additional associated operations. We describe  
6181 this in terms of two intermediate matrices:

- 6182 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the binary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 6183 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6184 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as one of the following:

6185 bind-first:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$ ,

6186 bind-second:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$ ,

6187 where  $f = \mathbf{f}(\mathbf{op})$ .

6188 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6189 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6190 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6191 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6192 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6193 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6194 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6195 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6196 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6197 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6200 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 6201 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 6202 mask which acts as a “write mask”.

- 6203 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 6204 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6205 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6206 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 6207 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 6208 mask are unchanged:

$$6209 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6210 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 6211 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 6212 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 6213 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 6214 sequence.

#### 6215 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6216 Computes the transformation of the values of the stored elements of a vector using an index unary  
 6217 operator that is a function of the stored value, its location indices, and an user provided scalar  
 6218 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6219 C Syntax

```
6220      GrB_Info GrB_apply(GrB_Vector          w,
6221                        const GrB_Vector    mask,
6222                        const GrB_BinaryOp   accum,
6223                        const GrB_IndexUnaryOp op,
6224                        const GrB_Vector    u,
6225                        <type>              val,
6226                        const GrB_Descriptor desc);
```

```
6227      GrB_Info GrB_apply(GrB_Vector          w,
6228                        const GrB_Vector    mask,
6229                        const GrB_BinaryOp   accum,
6230                        const GrB_IndexUnaryOp op,
6231                        const GrB_Vector    u,
6232                        const GrB_Scalar    s,
6233                        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.

6265                   GrB\_PANIC   Unknown internal error.

6266           GrB\_INVALID\_OBJECT   This is returned in any execution mode whenever one of the  
6267                                   opaque GraphBLAS objects (input or output) is in an invalid  
6268                                   state caused by a previous execution error. Call `GrB_error()` to  
6269                                   access any error messages generated by the implementation.

6270           GrB\_OUT\_OF\_MEMORY   Not enough memory available for operation.

6271   GrB\_UNINITIALIZED\_OBJECT   One or more of the GraphBLAS objects has not been initialized  
6272                                   by a call to `new` (or another constructor).

6273   GrB\_DIMENSION\_MISMATCH   `mask`, `w` and/or `u` dimensions are incompatible.

6274   GrB\_DOMAIN\_MISMATCH   The domains of the various vectors are incompatible with the cor-  
6275                                   responding domains of the accumulation operator or index unary  
6276                                   operator, or the mask's domain is not compatible with `bool` (in  
6277                                   the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6278           GrB\_EMPTY\_OBJECT   The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
6279                                   therefore a value cannot be passed to the index unary operator.

## 6280 Description

6281   This variant of `GrB_apply` computes the result of applying an index unary operator to the elements  
6282   of a GraphBLAS vector each composed with the element's index and a scalar constant, `val` or `s`:

$$6283 \quad w = f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = f_i(u, \mathbf{ind}(u), 0, s),$$

6284   or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6285 \quad w = w \odot f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = w \odot f_i(u, \mathbf{ind}(u), 0, s).$$

6286   Logically, this operation occurs in three steps:

6287       **Setup**   The internal vectors and mask used in the computation are formed and their domains  
6288                   and dimensions are tested for compatibility.

6289       **Compute**   The indicated computations are carried out.

6290       **Output**   The result is written into the output vector, possibly under control of a mask.

6291   Up to three argument vectors are used in this `GrB_apply` operation:

- 6292   1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6293   2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)

6294 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6295 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6296 are tested for domain compatibility as follows:

- 6297 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
6298 must be from one of the pre-defined types of Table 3.2.
- 6299 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the index unary operator.
- 6300 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6301 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be compatible  
6302 with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6303 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6304 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  
6305 the index unary operator.
- 6306 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  
6307 unary operator.

6308 Two domains are compatible with each other if values from one domain can be cast to values in  
6309 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6310 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6311 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
6312 error listed above is returned.

6313 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6314 denotes copy):

- 6315 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6316 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 6317 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 6318 (b) If `mask  $\neq$  GrB_NULL`,
    - 6319 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
    - 6320 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 6321 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6322 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6323 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6324 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6325 must hold:

6326 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6327 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

6328 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6329 error listed above is returned.

6330 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6331 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6332 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.

6333 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided ( $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable `val`  
6334 with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6335 We are now ready to carry out the apply and any additional associated operations. We describe  
6336 this in terms of two intermediate vectors:

- 6337 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6338  $\tilde{\mathbf{u}}$ .
- 6339 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6340 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6341 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6342 where  $f_i = \mathbf{f}(\text{op})$ .

6343 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6344 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 6345 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6346 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6347 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6348 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6349 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6350 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6351 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6352 \quad & \\ 6353 \end{aligned}$$

6354 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6355 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6356 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6357 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.

#### 6370 4.3.8.6 apply: Matrix index unary operator variant[Scott: NEW CONTENT]

6371 Computes the transformation of the values of the stored elements of a matrix using an index unary  
6372 operator that is a function of the stored value, its location indices, and an user provided scalar  
6373 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6374 C Syntax

```
6375     GrB_Info GrB_apply(GrB_Matrix      C,
6376                       const GrB_Matrix Mask,
6377                       const GrB_BinaryOp accum,
6378                       const GrB_IndexUnaryOp op,
6379                       const GrB_Matrix A,
6380                       <type>          val,
6381                       const GrB_Descriptor desc);
```

```
6382     GrB_Info GrB_apply(GrB_Matrix      C,
6383                       const GrB_Matrix Mask,
6384                       const GrB_BinaryOp accum,
6385                       const GrB_IndexUnaryOp op,
6386                       const GrB_Matrix A,
6387                       const GrB_Scalar s,
6388                       const GrB_Descriptor desc);
```

#### 6389 Parameters

6390 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6391 that may be accumulated with the result of the apply operation. On output, the  
6392 matrix holds the results of the operation.

6393 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6394 stored into the output matrix **C**. The mask dimensions must match those of the  
6395 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6396 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6397 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6398 dimensions of **C**), **GrB\_NULL** should be specified.

6399 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6400 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6401 specified.

6402 **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  
6403 to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6404 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6405 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6406 ator.

6407 **val** (IN) An additional scalar value that is passed to the index unary operator.

6408 **s** (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6409 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6410 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.

## 6413 Return Values

6414 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
6415 blocking mode, this indicates that the compatibility tests on di-  
6416 mensions and domains for the input arguments passed successfully.  
6417 Either way, output matrix **C** is ready to be used in the next method  
6418 of the sequence.

6419 **GrB\_PANIC** Unknown internal error.

6420 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
6421 GraphBLAS objects (input or output) is in an invalid state caused

6422 by a previous execution error. Call `GrB_error()` to access any error  
 6423 messages generated by the implementation.

6424 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6425 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 6426 a call to `new` (or another constructor).

6427 **GrB\_DIMENSION\_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6428 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the  
 6429 corresponding domains of the accumulation operator or index unary  
 6430 operator, or the mask's domain is not compatible with `bool` (in the  
 6431 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6432 **GrB\_EMPTY\_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
 6433 therefore a value cannot be passed to the index unary operator.

## 6434 Description

6435 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements  
 6436 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar  
 6437 constant, `val` or `s`:

$$6438 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6439 or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6440 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6441 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
 6442 indices, respectively.

6443 Logically, this operation occurs in three steps:

6444 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6445 and dimensions are tested for compatibility.

6446 **Compute** The indicated computations are carried out.

6447 **Output** The result is written into the output matrix, possibly under control of a mask.

6448 Up to three argument matrices are used in the `GrB_apply` operation:

- 6449 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6450 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)

6451 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6452 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6453 are tested for domain compatibility as follows:

- 6454 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{Mask})$   
6455 must be from one of the pre-defined types of Table 3.2.
- 6456 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator.
- 6457 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6458 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be compatible  
6459 with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 6460 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6461 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  
6462 the index unary operator.
- 6463 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  
6464 unary operator.

6465 Two domains are compatible with each other if values from one domain can be cast to values in  
6466 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6467 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6468 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
6469 error listed above is returned.

6470 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation  
6471 are formed ( $\leftarrow$  denotes copy):

- 6472 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6473 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument **Mask** as follows:
  - 6474 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
6475  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 6476 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 6477 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
6478  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 6479 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
6480  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 6481 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6482 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument **A** as follows:
  - 6483  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6484 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6485 The internal matrices and mask are checked for dimension compatibility. The following conditions  
6486 must hold:

- 6487 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6488 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6489 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6490 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6491 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6492 error listed above is returned.

6493 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6494 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6495 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6496 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6497 `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6498 We are now ready to carry out the apply and any additional associated operations. We describe  
6499 this in terms of two intermediate matrices:

- 6500 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6501  $\tilde{\mathbf{A}}$ .
- 6502 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6503 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6504 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6505 where  $f_i = \mathbf{f}(\mathbf{op})$ .

6506 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6507 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6508 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6509 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6510 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6511 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 6512 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6513 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6514 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6515 \end{aligned}$$

6516 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6518 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6519 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6520 mask which acts as a “write mask”.

- 6521 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
6522 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6523 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6524 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6525 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6526 mask are unchanged:

$$6527 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6528 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
6529 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
6530 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
6531 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6532 sequence.

#### 6533 4.3.9 select:

6534 Apply a select operator to the stored elements of an object to determine whether or not to keep  
6535 them.

##### 6536 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6537 Apply a select operator (an index unary operator) to the elements of a vector.

#### 6538 C Syntax

```
6539 // scalar value variant
6540 GrB_Info GrB_select(GrB_Vector          w,
6541                    const GrB_Vector     mask,
6542                    const GrB_BinaryOp   accum,
6543                    const GrB_IndexUnaryOp op,
6544                    const GrB_Vector     u,
6545                    <type>               val,
6546                    const GrB_Descriptor  desc);
6547
6548 // GraphBLAS scalar variant
6549 GrB_Info GrB_select(GrB_Vector          w,
6550                    const GrB_Vector     mask,
```

```

6551         const GrB_BinaryOp      accum,
6552         const GrB_IndexUnaryOp  op,
6553         const GrB_Vector        u,
6554         const GrB_Scalar        s,
6555         const GrB_Descriptor    desc);
6556

```

## 6557 Parameters

6558     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6559     that may be accumulated with the result of the select operation. On output, this  
6560     vector holds the results of the operation.

6561     **mask** (IN) An optional “write” mask that controls which results from this operation are  
6562     stored into the output vector **w**. The mask dimensions must match those of the  
6563     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6564     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6565     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6566     dimensions of **w**), **GrB\_NULL** should be specified.

6567     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6568     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6569     specified.

6570     **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  
6571     to each element stored in the input vector, **u**. It is a function of the stored element’s  
6572     value, its location index, and a user supplied scalar value (either **s** or **val**).

6573     **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-  
6574     ator.

6575     **val** (IN) An additional scalar value that is passed to the index unary operator.

6576     **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6577     not be empty.

6578     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6579     should be specified. Non-default field/value pairs are listed as follows:

6580

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> .

6581

## 6582 Return Values

6583           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 6584                           blocking mode, this indicates that the compatibility tests on di-  
 6585                           mensions and domains for the input arguments passed success-  
 6586                           fully. Either way, output vector **w** is ready to be used in the next  
 6587                           method of the sequence.

6588           **GrB\_PANIC** Unknown internal error.

6589           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 6590                           opaque GraphBLAS objects (input or output) is in an invalid  
 6591                           state caused by a previous execution error. Call **GrB\_error()** to  
 6592                           access any error messages generated by the implementation.

6593           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6594           **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
 6595                           by a call to one of its constructors.

6596           **GrB\_DIMENSION\_MISMATCH** **mask**, **w** and/or **u** dimensions are incompatible.

6597           **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the cor-  
 6598                           responding domains of the accumulation operator or index unary  
 6599                           operator, or the mask's domain is not compatible with **bool** (in  
 6600                           the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

6601           **GrB\_EMPTY\_OBJECT** The **GrB\_Scalar s** used in the call is empty (**nvals(s) = 0**) and  
 6602                           therefore a value cannot be passed to the index unary operator.

## 6603 Description

6604 This variant of **GrB\_select** computes the result of applying a index unary operator to select the  
 6605 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored  
 6606 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding  
 6607 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.  
 6608 This acts like a functional mask on the input vector as follows:

$$6609 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6610 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6611 Correspondingly, if a **GrB\_Scalar s**, is provided:

$$6612 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6613 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$



6614 Logically, this operation occurs in three steps:

6615     **Setup** The internal vectors and mask used in the computation are formed and their domains  
6616             and dimensions are tested for compatibility.

6617     **Compute** The indicated computations are carried out.

6618     **Output** The result is written into the output vector, possibly under control of a mask.

6619 Up to three argument vectors are used in this `GrB_select` operation:

- 6620     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6621     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 6622     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6623 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6624 are tested for domain compatibility as follows:

- 6625     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
6626         must be from one of the pre-defined types of Table 3.2.
- 6627     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 6628     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6629         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
6630         mulation operator.
- 6631     4.  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6632         i.e., castable to `bool`.
- 6633     5.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6634     6.  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(\mathbf{s})$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$   
6635         of the index unary operator.

6636 Two domains are compatible with each other if values from one domain can be cast to values in  
6637 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6638 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6639 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch  
6640 error listed above is returned.

6641 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6642 denotes copy):

- 6643     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6644     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

6645 (a) If  $\text{mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .  
6646 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
6647 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
6648 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
6649 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
6650 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .  
6651 4. Scalar  $\widetilde{s} \leftarrow s$  (GrB\_Scalar version only).

6652 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6653 must hold:

- 6654 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6655 2.  $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$ .

6656 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch  
6657 error listed above is returned.

6658 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6659 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6660 If an empty `GrB_Scalar`  $\widetilde{s}$  is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6661 If a non-empty `GrB_Scalar`,  $\widetilde{s}$ , is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 1$ ), we then create an internal variable  
6662 `val` with the same domain as  $\widetilde{s}$  and set  $\text{val} = \text{val}(\widetilde{s})$ .

6663 We are now ready to carry out the `select` and any additional associated operations. We describe  
6664 this in terms of two intermediate vectors:

- 6665 •  $\widetilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6666  $\widetilde{\mathbf{u}}$ .
- 6667 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6668 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$6669 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6670 where  $f_i = \mathbf{f}(\text{op})$ .

6671 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6672 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 6673 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$6674 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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);
```

```

6710 // GraphBLAS scalar variant
6711 GrB_Info GrB_select(GrB_Matrix          C,
6712                    const GrB_Matrix     Mask,
6713                    const GrB_BinaryOp    accum,
6714                    const GrB_IndexUnaryOp op,
6715                    const GrB_Matrix      A,
6716                    const GrB_Scalar      s,
6717                    const GrB_Descriptor   desc);

```

## 6718 Parameters

6719 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6720 that may be accumulated with the result of the select operation. On output, the  
6721 matrix holds the results of the operation.

6722 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6723 stored into the output matrix **C**. The mask dimensions must match those of the  
6724 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6725 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6726 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6727 dimensions of **C**), **GrB\_NULL** should be specified.

6728 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6729 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6730 specified.

6731 **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  
6732 to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6733 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6734 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6735 ator.

6736 **val** (IN) An additional scalar value that is passed to the index unary operator.

6737 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6738 not be empty.

6739 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6740 should be specified. Non-default field/value pairs are listed as follows:

6741

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:

6771  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ , or  
6772  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ .

6773 Correspondingly, if a GrB\_Scalar,  $s$ , is provided:

6774  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ , or  
6775  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ .

6776 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
6777 indices, respectively.

6778 Logically, this operation occurs in three steps:

6779 **Setup** The internal matrices and mask used in the computation are formed and their domains  
6780 and dimensions are tested for compatibility.

6781 **Compute** The indicated computations are carried out.

6782 **Output** The result is written into the output matrix, possibly under control of a mask.

6783 Up to three argument matrices are used in the GrB\_select operation:

- 6784 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$   
6785 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)  
6786 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6787 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6788 are tested for domain compatibility as follows:

- 6789 1. If **Mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
6790 must be from one of the pre-defined types of Table 3.2.  
6791 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .  
6792 3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6793 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
6794 mulation operator.  
6795 4.  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6796 i.e., castable to **bool**.  
6797 5.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.  
6798 6.  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
6799 of the index unary operator.

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

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

1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - (b) If `Mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument `A` as follows:  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$
4. Scalar  $\tilde{s} \leftarrow s$  (`GrB_Scalar` version only).

The internal matrices and mask are checked for dimension compatibility. The following conditions must hold:

1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

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

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

If an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`. If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

We are now ready to carry out the `select` and any additional associated operations. We describe this in terms of two intermediate matrices:

- 6834 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6835  $\tilde{\mathbf{A}}$ .
- 6836 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6837 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6838 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6839 where  $f_i = \mathbf{f}(\text{op})$ .

6840 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6841 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6842 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6843 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6844 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6845 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6846 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6847 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6848 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6849 \quad 6850$$

6851 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6852 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6853 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6854 mask which acts as a “write mask”.

- 6855 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
6856 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6857 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6858 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6859 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6860 mask are unchanged:

$$6861 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6862 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
6863 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
6864 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
6865 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6866 sequence.



### 6867 4.3.10 reduce: Perform a reduction across the elements of an object

6868 Computes the reduction of the values of the elements of a vector or matrix.

#### 6869 4.3.10.1 reduce: Standard matrix to vector variant

6870 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns  
6871 is desired, the input matrix should be transposed using the descriptor.

### 6872 C Syntax

```
6873     GrB_Info GrB_reduce(GrB_Vector      w,  
6874                        const GrB_Vector mask,  
6875                        const GrB_BinaryOp accum,  
6876                        const GrB_Monoid op,  
6877                        const GrB_Matrix A,  
6878                        const GrB_Descriptor desc);  
6879  
6880     GrB_Info GrB_reduce(GrB_Vector      w,  
6881                        const GrB_Vector mask,  
6882                        const GrB_BinaryOp accum,  
6883                        const GrB_BinaryOp op,  
6884                        const GrB_Matrix A,  
6885                        const GrB_Descriptor desc);
```

### 6886 Parameters

6887 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6888 that may be accumulated with the result of the reduction operation. On output,  
6889 this vector holds the results of the operation.

6890 **mask** (IN) An optional “write” mask that controls which results from this operation are  
6891 stored into the output vector **w**. The mask dimensions must match those of the  
6892 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6893 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6894 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6895 dimensions of **w**), **GrB\_NULL** should be specified.

6896 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6897 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6898 specified.

6899 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.  
6900 Depending on which type is passed, the following defines the binary operator with  
6901 one domain,  $F_b = \langle D, D, D, \oplus \rangle$ , that is used:

6902  
6903  
6904  
  
6905  
6906  
6907  
  
6908  
  
6909  
6910  
6911  
  
6912  
  
  
  
  
  
  
  
  
  
6913  
  
6914  
6915  
6916  
6917  
6918  
  
6919  
  
6920  
6921  
6922  
6923  
  
6924  
  
6925  
6926  
  
6927  
  
6928  
6929  
6930

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 of the monoid is ignored.

If `op` is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in both cases  $\odot(\text{op})$  must be commutative and associative. Otherwise, the outcome of the operation is undefined.

`A` (IN) The GraphBLAS matrix on which reduction will be performed.

`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
<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.

## 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 vector parameters).

`GrB_DIMENSION_MISMATCH` `mask`, `w` and/or `u` dimensions are incompatible.

`GrB_DOMAIN_MISMATCH` Either the domains of the various vectors and matrices are incompatible with the corresponding domains of the accumulation operator or reduce function, or the domains of the GraphBLAS binary

6931 operator `op` are not all the same, or the mask's domain is not com-  
 6932 patible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE`  
 6933 is not set).

## 6934 Description

6935 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows  
 6936 of an input matrix:  $w(i) = \bigoplus A(i, :) \forall i$ ; or, if an optional binary accumulation operator is provided,  
 6937  $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$ , where  $\bigoplus = \odot(F_b)$  and  $\odot = \odot(\text{accum})$ .

6938 Logically, this operation occurs in three steps:

6939     **Setup** The internal vector, matrix and mask used in the computation are formed and their  
 6940 domains and dimensions are tested for compatibility.

6941     **Compute** The indicated computations are carried out.

6942     **Output** The result is written into the output vector, possibly under control of a mask.

6943 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6944 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6945 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 6946 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6947 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested  
 6948 for domain compatibility as follows:

- 6949 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 6950 must be from one of the pre-defined types of Table 3.2.
- 6951 2.  $\mathbf{D}(w)$  must be compatible with the domain of the reduction binary operator,  $\mathbf{D}(F_b)$ .
- 6952 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 6953 of the accumulation operator and  $\mathbf{D}(F_b)$ , must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 6954 mulation operator.
- 6955 4.  $\mathbf{D}(A)$  must be compatible with the domain of the binary reduction operator,  $\mathbf{D}(F_b)$ .

6956 Two domains are compatible with each other if values from one domain can be cast to values in  
 6957 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6958 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6959 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
 6960 error listed above is returned.

6961 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 6962 denotes copy):

- 6963 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6964 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 6965 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 6966 (b) If `mask  $\neq$  GrB_NULL`,
- 6967 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 6968 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 6969 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6970 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

6971 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6972 must hold:

- 6973 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6974 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6975 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-  
6976 match error listed above is returned.

6977 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6978 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6979 We carry out the reduce and any additional associated operations. We describe this in terms of  
6980 two intermediate vectors:

- 6981 •  $\tilde{\mathbf{t}}$ : The vector holding the result from reducing along the rows of input matrix  $\tilde{\mathbf{A}}$ .
- 6982 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6983 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6984 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6985 The value of each of its elements is computed by

$$6986 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6987 where  $\bigoplus = \odot(F_b)$ .

6988 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6989 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

6990 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6991 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6992 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6993 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6994 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 6995 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6996 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

6999 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7000 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
7001 using what is called a *standard vector mask and replace*. This is carried out under control of the  
7002 mask which acts as a “write mask”.

7003 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
7004 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$7005 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7006 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
7007 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
7008 mask are unchanged:

$$7009 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7010 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
7011 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
7012 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
7013 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7014 sequence.

#### 7015 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

7016 Reduce all stored values into a single scalar.

### 7017 C Syntax

```
7018 // scalar value + monoid (only)
7019 GrB_Info GrB_reduce(<type>          *val,
7020                      const GrB_BinaryOp accum,
7021                      const GrB_Monoid op,
7022                      const GrB_Vector u,
```

```

7023             const GrB_Descriptor desc);
7024
7025 // GraphBLAS Scalar + monoid
7026 GrB_Info GrB_reduce(GrB_Scalar      s,
7027                    const GrB_BinaryOp accum,
7028                    const GrB_Monoid op,
7029                    const GrB_Vector u,
7030                    const GrB_Descriptor desc);
7031
7032 // GraphBLAS Scalar + binary operator
7033 GrB_Info GrB_reduce(GrB_Scalar      s,
7034                    const GrB_BinaryOp accum,
7035                    const GrB_BinaryOp op,
7036                    const GrB_Vector u,
7037                    const GrB_Descriptor desc);

```

## 7038 Parameters

7039 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7040 a value that may be accumulated (optionally) with the result of the reduction  
7041 operation. On output, this scalar holds the results of the operation.

7042 **accum** (IN) An optional binary operator used for accumulating entries into an exist-  
7043 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,  
7044 **GrB\_NULL** should be specified.

7045 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7046 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7047 otherwise, the outcome of the operation is undefined.

7048 **u** (IN) The GraphBLAS vector on which reduction will be performed.

7049 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7050 should be specified. Non-default field/value pairs are listed as follows:

7052 Param	Field	Value	Description
------------	-------	-------	-------------

7053 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7054 tions. There are currently no non-default field/value pairs that can be set for this  
7055 operation.

## 7056 Return Values

7057 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
7058 cessfully, and the output scalar (**s** or **val**) is ready to be used in the  
7059 next method of the sequence.

7060                   GrB\_PANIC Unknown internal error.

7061           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7062                   GraphBLAS objects (input or output) is in an invalid state caused  
7063                   by a previous execution error. Call GrB\_error() to access any error  
7064                   messages generated by the implementation.

7065           GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7066 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7067                   a call to a respective constructor.

7068           GrB\_NULL\_POINTER val pointer is NULL.

7069           GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
7070                   the corresponding domains of the accumulation operator, or reduce  
7071                   operator.

## 7072 Description

7073 This variant of GrB\_reduce computes the result of performing a reduction across all of the stored  
7074 elements of an input vector storing the result into either s or val. This corresponds to (shown here  
7075 for the scalar value case only):

$$7076 \quad \text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7077 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7078 Logically, this operation occurs in three steps:

7079       **Setup** The internal vector used in the computation is formed and its domain is tested for  
7080                   compatibility.

7081       **Compute** The indicated computations are carried out.

7082       **Output** The result is written into the output scalar.

7083 One vector argument is used in this GrB\_reduce operation:

- 7084       1.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7085 The output scalar, argument vector, reduction operator and accumulation operator (if provided)  
7086 are tested for domain compatibility as follows:

- 7087       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  
7088            $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

- 7089 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  
7090  $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
7091 be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 7092 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  $F_b$ ).

7093 Two domains are compatible with each other if values from one domain can be cast to values in  
7094 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
7095 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
7096 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
7097 error listed above is returned.

7098 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one  
7099 of the following occurs depending on the output variant:

$$7100 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7101 or

$$7102 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7103 where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7104 For all other cases, the internal vector and scalar used in the computation is formed ( $\leftarrow$  denotes  
7105 copy):

7106 1. Vector  $\tilde{\mathbf{u}} \leftarrow \text{u}$ .

7107 2. Scalar  $\tilde{s} \leftarrow \text{s}$  (GraphBLAS scalar case).

7108 We are now ready to carry out the reduction and any additional associated operations. An inter-  
7109 mediate scalar result  $t$  is computed as follows:

$$7110 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

7111 where  $\oplus = \odot(\text{op})$ .

7112 The final reduction value is computed as follows:

$$7113 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7114 or

$$7115 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB\_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$



7116 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
7117 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7118 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7119 Reduce all stored values into a single scalar.

### 7120 C Syntax

```
7121 // scalar value + monoid (only)
7122 GrB_Info GrB_reduce(<type>          *val,
7123                   const GrB_BinaryOp accum,
7124                   const GrB_Monoid   op,
7125                   const GrB_Matrix   A,
7126                   const GrB_Descriptor desc);
7127
7128 // GraphBLAS Scalar + monoid
7129 GrB_Info GrB_reduce(GrB_Scalar      s,
7130                   const GrB_BinaryOp accum,
7131                   const GrB_Monoid   op,
7132                   const GrB_Matrix   A,
7133                   const GrB_Descriptor desc);
7134
7135 // GraphBLAS Scalar + binary operator
7136 GrB_Info GrB_reduce(GrB_Scalar      s,
7137                   const GrB_BinaryOp accum,
7138                   const GrB_BinaryOp op,
7139                   const GrB_Matrix   A,
7140                   const GrB_Descriptor desc);
```

### 7141 Parameters

7142 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7143 a value that may be accumulated (optionally) with the result of the reduction  
7144 operation. On output, this scalar holds the results of the operation.

7145 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or  
7146 **val**) value. If assignment rather than accumulation is desired, GrB\_NULL should  
7147 be specified.

7148 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7149 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7150 otherwise, the outcome of the operation is undefined.

7151 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

7152 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 7153 should be specified. Non-default field/value pairs are listed as follows:

7154

Param	Field	Value	Description
-------	-------	-------	-------------

7156 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
 7157 tions. There are currently no non-default field/value pairs that can be set for this  
 7158 operation.

## 7159 Return Values

7160 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 7161 cessfully, and the output scalar (s or val) is ready to be used in the  
 7162 next method of the sequence.

7163 GrB\_PANIC Unknown internal error.

7164 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 7165 GraphBLAS objects (input or output) is in an invalid state caused  
 7166 by a previous execution error. Call GrB\_error() to access any error  
 7167 messages generated by the implementation.

7168 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7169 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 7170 a call to a respective constructor.

7171 GrB\_NULL\_POINTER val pointer is NULL.

7172 GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
 7173 the corresponding domains of the accumulation operator, or reduce  
 7174 operator.

## 7175 Description

7176 This variant of GrB\_reduce computes the result of performing a reduction across all of the stored  
 7177 elements of an input matrix storing the result into either s or val. This corresponds to (shown here  
 7178 for the scalar value case only):

$$7179 \quad \text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7180 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7181 Logically, this operation occurs in three steps:

7182       **Setup** The internal matrix used in the computation is formed and its domain is tested for  
 7183       compatibility.

7184       **Compute** The indicated computations are carried out.

7185       **Output** The result is written into the output scalar.

7186   One matrix argument is used in this GrB\_reduce operation:

7187       1.  $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7188   The output scalar, argument matrix, reduction operator and accumulation operator (if provided)  
 7189   are tested for domain compatibility as follows:

7190       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  
 7191        $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

7192       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  
 7193        $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
 7194       be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

7195       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  $F_b$ ).

7196   Two domains are compatible with each other if values from one domain can be cast to values in  
 7197   the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7198   compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7199   any compatibility rule above is violated, execution of GrB\_reduce ends and the domain mismatch  
 7200   error listed above is returned.

7201   The number of values stored in the input,  $A$ , is checked. If there are no stored values in  $A$ , then  
 7202   one of the following occurs depending on the output variant:

$$7203 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7204   or

$$7205 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7206   where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with GrB\_SUCCESS.

7207   For all other cases, the internal matrix and scalar used in the computation is formed ( $\leftarrow$  denotes  
 7208   copy):

7209       1. Matrix  $\tilde{A} \leftarrow A$ .

7210       2. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

7211 We are now ready to carry out the reduce and any additional associated operations. An intermediate  
 7212 scalar result  $t$  is computed as follows:

$$7213 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7214 where  $\oplus = \odot(\text{op})$ .

7215 The final reduction value is computed as follows:

$$7216 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7217 or

$$7218 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB\_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7219 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
 7220 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7221 4.3.11 transpose: Transpose rows and columns of a matrix

7222 This version computes a new matrix that is the transpose of the source matrix.

#### 7223 C Syntax

```
7224      GrB_Info GrB_transpose(GrB_Matrix      C,
7225                           const GrB_Matrix Mask,
7226                           const GrB_BinaryOp accum,
7227                           const GrB_Matrix A,
7228                           const GrB_Descriptor desc);
```

#### 7229 Parameters

7230 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 7231 that may be accumulated with the result of the transpose operation. On output,  
 7232 the matrix holds the results of the operation.

7233 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 7234 stored into the output matrix C. The mask dimensions must match those of the  
 7235 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
 7236 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
 7237 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
 7238 dimensions of C), GrB\_NULL should be specified.

7239 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
 7240 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 7241 specified.

7242 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7243 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 7244 should be specified. Non-default field/value pairs are listed as follows:  
 7245

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.

## 7247 Return Values

7248 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 7249 blocking mode, this indicates that the compatibility tests on di-  
 7250 mensions and domains for the input arguments passed successfully.  
 7251 Either way, output matrix C is ready to be used in the next method  
 7252 of the sequence.

7253 **GrB\_PANIC** Unknown internal error.

7254 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 7255 GraphBLAS objects (input or output) is in an invalid state caused  
 7256 by a previous execution error. Call **GrB\_error()** to access any error  
 7257 messages generated by the implementation.

7258 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

7259 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 7260 a call to **new** (or **Matrix\_dup** for matrix parameters).

7261 **GrB\_DIMENSION\_MISMATCH** mask, C and/or A dimensions are incompatible.

7262 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the cor-  
 7263 responding domains of the accumulation operator, or the mask's do-  
 7264 main is not compatible with **bool** (in the case where desc[GrB\_MASK].GrB\_STRUCT  
 7265 is not set).

## 7266 Description

7267 GrB\_transpose computes the result of performing a transpose of the input matrix:  $C = A^T$ ; or, if an  
 7268 optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A^T$ . We note that the input matrix  
 7269 A can itself be optionally transposed before the operation, which would cause either an assignment  
 7270 from A to C or an accumulation of A into C.

7271 Logically, this operation occurs in three steps:

7272     **Setup** The internal matrix and mask used in the computation are formed and their domains  
 7273             and dimensions are tested for compatibility.

7274     **Compute** The indicated computations are carried out.

7275     **Output** The result is written into the output matrix, possibly under control of a mask.

7276 Up to three matrix arguments are used in this GrB\_transpose operation:

- 7277     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7278     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)
- 7279     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7280 The argument matrices and accumulation operator (if provided) are tested for domain compatibility  
 7281 as follows:

- 7282     1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 7283         must be from one of the pre-defined types of Table 3.2.
- 7284     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$  of the input matrix.
- 7285     3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 7286         of the accumulation operator and  $\mathbf{D}(A)$  of the input matrix must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 7287         of the accumulation operator.

7288 Two domains are compatible with each other if values from one domain can be cast to values in  
 7289 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7290 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7291 any compatibility rule above is violated, execution of GrB\_transpose ends and the domain mismatch  
 7292 error listed above is returned.

7293 From the argument matrices, the internal matrices and mask used in the computation are formed  
 7294 ( $\leftarrow$  denotes copy):

- 7295     1. Matrix  $\tilde{C} \leftarrow C$ .
- 7296     2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument Mask as follows:

- 7297 (a) If  $\text{Mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
7298  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 7299 (b) If  $\text{Mask} \neq \text{GrB\_NULL}$ ,
- 7300 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
7301  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
- 7302 ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
7303  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
- 7304 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$ .
- 7305 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

7306 The internal matrices and masks are checked for dimension compatibility. The following conditions  
7307 must hold:

- 7308 1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$ .
- 7309 2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$ .
- 7310 3.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$ .
- 7311 4.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$ .

7312 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension  
7313 mismatch error listed above is returned.

7314 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
7315 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7316 We are now ready to carry out the matrix transposition and any additional associated operations.  
7317 We describe this in terms of two intermediate matrices:

- 7318 •  $\widetilde{\mathbf{T}}$ : The matrix holding the transpose of  $\widetilde{\mathbf{A}}$ .
- 7319 •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

7320 The intermediate matrix

$$7321 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7322 is created.

7323 The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7324 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 7325 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$7326 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7327 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 7328 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned}
 7329 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 7330 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7331 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7332 \quad & \\
 7333 \quad &
 \end{aligned}$$

7334 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7335 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 7336 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 7337 mask which acts as a “write mask”.

- 7338 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 7339 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7340 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7341 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 7342 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 7343 mask are unchanged:

$$7344 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7345 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 7346 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 7347 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 7348 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 7349 sequence.

#### 7350 4.3.12 kronecker: Kronecker product of two matrices

7351 Computes the Kronecker product of two matrices. The result is a matrix.

#### 7352 C Syntax

```

7353      GrB_Info GrB_kronecker(GrB_Matrix      C,
7354                             const GrB_Matrix  Mask,
7355                             const GrB_BinaryOp accum,
7356                             const GrB_Semiring op,
7357                             const GrB_Matrix  A,
7358                             const GrB_Matrix  B,
7359                             const GrB_Descriptor desc);
7360
  
```



```

7361      GrB_Info GrB_kronecker(GrB_Matrix      C,
7362                             const GrB_Matrix Mask,
7363                             const GrB_BinaryOp accum,
7364                             const GrB_Monoid op,
7365                             const GrB_Matrix A,
7366                             const GrB_Matrix B,
7367                             const GrB_Descriptor desc);
7368
7369      GrB_Info GrB_kronecker(GrB_Matrix      C,
7370                             const GrB_Matrix Mask,
7371                             const GrB_BinaryOp accum,
7372                             const GrB_BinaryOp op,
7373                             const GrB_Matrix A,
7374                             const GrB_Matrix B,
7375                             const GrB_Descriptor desc);

```

## 7376 Parameters

7377 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
7378 that may be accumulated with the result of the Kronecker product. On output,  
7379 the matrix holds the results of the operation.

7380 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
7381 stored into the output matrix C. The mask dimensions must match those of the  
7382 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
7383 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
7384 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
7385 dimensions of C), GrB\_NULL should be specified.

7386 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7387 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
7388 specified.

7389 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
7390 operation. Depending on which type is passed, the following defines the binary  
7391 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

7392 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

7393 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-  
7394 nored.

7395 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  
7396 is ignored.

7397 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
7398 product.

7399           B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
7400           product.

7401       desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7402           should be specified. Non-default field/value pairs are listed as follows:  
7403

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.

## 7405 Return Values

7406           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
7407           blocking mode, this indicates that the compatibility tests on di-  
7408           mensions and domains for the input arguments passed successfully.  
7409           Either way, output matrix C is ready to be used in the next method  
7410           of the sequence.

7411           GrB\_PANIC Unknown internal error.

7412           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7413           GraphBLAS objects (input or output) is in an invalid state caused  
7414           by a previous execution error. Call GrB\_error() to access any error  
7415           messages generated by the implementation.

7416           GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7417           GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7418           a call to new (or Matrix\_dup for matrix parameters).

7419           GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

7420           GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
7421           corresponding domains of the binary operator (op) or accumulation  
7422           operator, or the mask's domain is not compatible with bool (in the  
7423           case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 7424 Description

7425       GrB\_kronecker computes the Kronecker product  $C = A \otimes B$  or, if an optional binary accumulation  
7426       operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$  (where matrices A and B can be optionally transposed).

7427 The Kronecker product is defined as follows:

7428

$$7429 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

7430 where  $A : \mathbb{S}^{m_A \times n_A}$ ,  $B : \mathbb{S}^{m_B \times n_B}$ , and  $C : \mathbb{S}^{m_A m_B \times n_A n_B}$ . More explicitly, the elements of the  
7431 Kronecker product are defined as

$$7432 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7433 where  $\otimes$  is the multiplicative operator specified by the **op** parameter.

7434 Logically, this operation occurs in three steps:

7435     **Setup** The internal matrices and mask used in the computation are formed and their domains  
7436     and dimensions are tested for compatibility.

7437     **Compute** The indicated computations are carried out.

7438     **Output** The result is written into the output matrix, possibly under control of a mask.

7439 Up to four argument matrices are used in the **GrB\_kronecker** operation:

- 7440 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7441 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)
- 7442 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 7443 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

7444 The argument matrices, the "product" operator (**op**), and the accumulation operator (if provided)  
7445 are tested for domain compatibility as follows:

- 7446 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
7447     must be from one of the pre-defined types of Table 3.2.
- 7448 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 7449 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 7450 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 7451 5. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
7452     of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of **op** must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
7453     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.

7487 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$   
7488  $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$  is created. The value of  
7489 each of its elements is computed by

$$7490 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7491 where  $\otimes$  is the multiplicative operator specified by the `op` parameter.

7492 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7493 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 7494 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$7495 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7496 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
7497 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$7498 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7499 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7500 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7501 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7502 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7503 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7504 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
7505 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
7506 mask which acts as a “write mask”.

- 7507 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
7508 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7509 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7510 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
7511 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
7512 mask are unchanged:

$$7513 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7514 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
7515 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
7516 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
7517 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7518 sequence. s



## Chapter 5

# Nonpolymorphic interface[Scott: NEW CONTENT]

Each polymorphic GraphBLAS method (those with multiple parameter signatures under the same name) has a corresponding set of long-name forms that are specific to each parameter signature. That is show in Tables 5.1 through 5.11.

Table 5.1: Long-name, nonpolymorphic form of GraphBLAS methods.

Polymorphic signature	Nonpolymorphic signature
GrB_Monoid_new(GrB_Monoid*,...,bool)	GrB_Monoid_new_BOOL(GrB_Monoid*,GrB_BinaryOp,bool)
GrB_Monoid_new(GrB_Monoid*,...,int8_t)	GrB_Monoid_new_INT8(GrB_Monoid*,GrB_BinaryOp,int8_t)
GrB_Monoid_new(GrB_Monoid*,...,uint8_t)	GrB_Monoid_new_UINT8(GrB_Monoid*,GrB_BinaryOp,uint8_t)
GrB_Monoid_new(GrB_Monoid*,...,int16_t)	GrB_Monoid_new_INT16(GrB_Monoid*,GrB_BinaryOp,int16_t)
GrB_Monoid_new(GrB_Monoid*,...,uint16_t)	GrB_Monoid_new_UINT16(GrB_Monoid*,GrB_BinaryOp,uint16_t)
GrB_Monoid_new(GrB_Monoid*,...,int32_t)	GrB_Monoid_new_INT32(GrB_Monoid*,GrB_BinaryOp,int32_t)
GrB_Monoid_new(GrB_Monoid*,...,uint32_t)	GrB_Monoid_new_UINT32(GrB_Monoid*,GrB_BinaryOp,uint32_t)
GrB_Monoid_new(GrB_Monoid*,...,int64_t)	GrB_Monoid_new_INT64(GrB_Monoid*,GrB_BinaryOp,int64_t)
GrB_Monoid_new(GrB_Monoid*,...,uint64_t)	GrB_Monoid_new_UINT64(GrB_Monoid*,GrB_BinaryOp,uint64_t)
GrB_Monoid_new(GrB_Monoid*,...,float)	GrB_Monoid_new_FP32(GrB_Monoid*,GrB_BinaryOp,float)
GrB_Monoid_new(GrB_Monoid*,...,double)	GrB_Monoid_new_FP64(GrB_Monoid*,GrB_BinaryOp,double)
GrB_Monoid_new(GrB_Monoid*,...,other)	GrB_Monoid_new_UDT(GrB_Monoid*,GrB_BinaryOp,void*)

Table 5.2: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_setElement(..., bool,...)	GrB_Scalar_setElement_BOOL(..., bool,...)
GrB_Scalar_setElement(..., int8_t,...)	GrB_Scalar_setElement_INT8(..., int8_t,...)
GrB_Scalar_setElement(..., uint8_t,...)	GrB_Scalar_setElement_UINT8(..., uint8_t,...)
GrB_Scalar_setElement(..., int16_t,...)	GrB_Scalar_setElement_INT16(..., int16_t,...)
GrB_Scalar_setElement(..., uint16_t,...)	GrB_Scalar_setElement_UINT16(..., uint16_t,...)
GrB_Scalar_setElement(..., int32_t,...)	GrB_Scalar_setElement_INT32(..., int32_t,...)
GrB_Scalar_setElement(..., uint32_t,...)	GrB_Scalar_setElement_UINT32(..., uint32_t,...)
GrB_Scalar_setElement(..., int64_t,...)	GrB_Scalar_setElement_INT64(..., int64_t,...)
GrB_Scalar_setElement(..., uint64_t,...)	GrB_Scalar_setElement_UINT64(..., uint64_t,...)
GrB_Scalar_setElement(..., float,...)	GrB_Scalar_setElement_FP32(..., float,...)
GrB_Scalar_setElement(..., double,...)	GrB_Scalar_setElement_FP64(..., double,...)
GrB_Scalar_setElement(..., <i>other</i> ,...)	GrB_Scalar_setElement_UDT(..., const void*,...)
GrB_Scalar_extractElement(bool*,...)	GrB_Scalar_extractElement_BOOL(bool*,...)
GrB_Scalar_extractElement(int8_t*,...)	GrB_Scalar_extractElement_INT8(int8_t*,...)
GrB_Scalar_extractElement(uint8_t*,...)	GrB_Scalar_extractElement_UINT8(uint8_t*,...)
GrB_Scalar_extractElement(int16_t*,...)	GrB_Scalar_extractElement_INT16(int16_t*,...)
GrB_Scalar_extractElement(uint16_t*,...)	GrB_Scalar_extractElement_UINT16(uint16_t*,...)
GrB_Scalar_extractElement(int32_t*,...)	GrB_Scalar_extractElement_INT32(int32_t*,...)
GrB_Scalar_extractElement(uint32_t*,...)	GrB_Scalar_extractElement_UINT32(uint32_t*,...)
GrB_Scalar_extractElement(int64_t*,...)	GrB_Scalar_extractElement_INT64(int64_t*,...)
GrB_Scalar_extractElement(uint64_t*,...)	GrB_Scalar_extractElement_UINT64(uint64_t*,...)
GrB_Scalar_extractElement(float*,...)	GrB_Scalar_extractElement_FP32(float*,...)
GrB_Scalar_extractElement(double*,...)	GrB_Scalar_extractElement_FP64(double*,...)
GrB_Scalar_extractElement( <i>other</i> *,...)	GrB_Scalar_extractElement_UDT(void*,...)



Table 5.3: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Vector_build(...,const bool*,...)	GrB_Vector_build_BOOL(...,const bool*,...)
GrB_Vector_build(...,const int8_t*,...)	GrB_Vector_build_INT8(...,const int8_t*,...)
GrB_Vector_build(...,const uint8_t*,...)	GrB_Vector_build_UINT8(...,const uint8_t*,...)
GrB_Vector_build(...,const int16_t*,...)	GrB_Vector_build_INT16(...,const int16_t*,...)
GrB_Vector_build(...,const uint16_t*,...)	GrB_Vector_build_UINT16(...,const uint16_t*,...)
GrB_Vector_build(...,const int32_t*,...)	GrB_Vector_build_INT32(...,const int32_t*,...)
GrB_Vector_build(...,const uint32_t*,...)	GrB_Vector_build_UINT32(...,const uint32_t*,...)
GrB_Vector_build(...,const int64_t*,...)	GrB_Vector_build_INT64(...,const int64_t*,...)
GrB_Vector_build(...,const uint64_t*,...)	GrB_Vector_build_UINT64(...,const uint64_t*,...)
GrB_Vector_build(...,const float*,...)	GrB_Vector_build_FP32(...,const float*,...)
GrB_Vector_build(...,const double*,...)	GrB_Vector_build_FP64(...,const double*,...)
GrB_Vector_build(...,const <i>other</i> *,...)	GrB_Vector_build_UDT(...,const void*,...)
GrB_Vector_setElement(...,GrB_Scalar,...)	GrB_Vector_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Vector_setElement(...,bool,...)	GrB_Vector_setElement_BOOL(..., bool,...)
GrB_Vector_setElement(...,int8_t,...)	GrB_Vector_setElement_INT8(..., int8_t,...)
GrB_Vector_setElement(...,uint8_t,...)	GrB_Vector_setElement_UINT8(..., uint8_t,...)
GrB_Vector_setElement(...,int16_t,...)	GrB_Vector_setElement_INT16(..., int16_t,...)
GrB_Vector_setElement(...,uint16_t,...)	GrB_Vector_setElement_UINT16(..., uint16_t,...)
GrB_Vector_setElement(...,int32_t,...)	GrB_Vector_setElement_INT32(..., int32_t,...)
GrB_Vector_setElement(...,uint32_t,...)	GrB_Vector_setElement_UINT32(..., uint32_t,...)
GrB_Vector_setElement(...,int64_t,...)	GrB_Vector_setElement_INT64(..., int64_t,...)
GrB_Vector_setElement(...,uint64_t,...)	GrB_Vector_setElement_UINT64(..., uint64_t,...)
GrB_Vector_setElement(...,float,...)	GrB_Vector_setElement_FP32(..., float,...)
GrB_Vector_setElement(...,double,...)	GrB_Vector_setElement_FP64(..., double,...)
GrB_Vector_setElement(..., <i>other</i> ,...)	GrB_Vector_setElement_UDT(...,const void*,...)
GrB_Vector_extractElement(GrB_Scalar,...)	GrB_Vector_extractElement_Scalar(GrB_Scalar,...)
GrB_Vector_extractElement(bool*,...)	GrB_Vector_extractElement_BOOL(bool*,...)
GrB_Vector_extractElement(int8_t*,...)	GrB_Vector_extractElement_INT8(int8_t*,...)
GrB_Vector_extractElement(uint8_t*,...)	GrB_Vector_extractElement_UINT8(uint8_t*,...)
GrB_Vector_extractElement(int16_t*,...)	GrB_Vector_extractElement_INT16(int16_t*,...)
GrB_Vector_extractElement(uint16_t*,...)	GrB_Vector_extractElement_UINT16(uint16_t*,...)
GrB_Vector_extractElement(int32_t*,...)	GrB_Vector_extractElement_INT32(int32_t*,...)
GrB_Vector_extractElement(uint32_t*,...)	GrB_Vector_extractElement_UINT32(uint32_t*,...)
GrB_Vector_extractElement(int64_t*,...)	GrB_Vector_extractElement_INT64(int64_t*,...)
GrB_Vector_extractElement(uint64_t*,...)	GrB_Vector_extractElement_UINT64(uint64_t*,...)
GrB_Vector_extractElement(float*,...)	GrB_Vector_extractElement_FP32(float*,...)
GrB_Vector_extractElement(double*,...)	GrB_Vector_extractElement_FP64(double*,...)
GrB_Vector_extractElement( <i>other</i> *,...)	GrB_Vector_extractElement_UDT(void*,...)
GrB_Vector_extractTuples(...,bool*,...)	GrB_Vector_extractTuples_BOOL(..., bool*,...)
GrB_Vector_extractTuples(...,int8_t*,...)	GrB_Vector_extractTuples_INT8(..., int8_t*,...)
GrB_Vector_extractTuples(...,uint8_t*,...)	GrB_Vector_extractTuples_UINT8(..., uint8_t*,...)
GrB_Vector_extractTuples(...,int16_t*,...)	GrB_Vector_extractTuples_INT16(..., int16_t*,...)
GrB_Vector_extractTuples(...,uint16_t*,...)	GrB_Vector_extractTuples_UINT16(..., uint16_t*,...)
GrB_Vector_extractTuples(...,int32_t*,...)	GrB_Vector_extractTuples_INT32(..., int32_t*,...)
GrB_Vector_extractTuples(...,uint32_t*,...)	GrB_Vector_extractTuples_UINT32(..., uint32_t*,...)
GrB_Vector_extractTuples(...,int64_t*,...)	GrB_Vector_extractTuples_INT64(..., int64_t*,...)
GrB_Vector_extractTuples(...,uint64_t*,...)	GrB_Vector_extractTuples_UINT64(..., uint64_t*,...)
GrB_Vector_extractTuples(...,float*,...)	GrB_Vector_extractTuples_FP32(..., float*,...)
GrB_Vector_extractTuples(...,double*,...)	GrB_Vector_extractTuples_FP64(..., double*,...)
GrB_Vector_extractTuples(..., <i>other</i> *,...)	GrB_Vector_extractTuples_UDT(..., void*,...)

Table 5.4: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_build(...,const bool*,...)	GrB_Matrix_build_BOOL(...,const bool*,...)
GrB_Matrix_build(...,const int8_t*,...)	GrB_Matrix_build_INT8(...,const int8_t*,...)
GrB_Matrix_build(...,const uint8_t*,...)	GrB_Matrix_build_UINT8(...,const uint8_t*,...)
GrB_Matrix_build(...,const int16_t*,...)	GrB_Matrix_build_INT16(...,const int16_t*,...)
GrB_Matrix_build(...,const uint16_t*,...)	GrB_Matrix_build_UINT16(...,const uint16_t*,...)
GrB_Matrix_build(...,const int32_t*,...)	GrB_Matrix_build_INT32(...,const int32_t*,...)
GrB_Matrix_build(...,const uint32_t*,...)	GrB_Matrix_build_UINT32(...,const uint32_t*,...)
GrB_Matrix_build(...,const int64_t*,...)	GrB_Matrix_build_INT64(...,const int64_t*,...)
GrB_Matrix_build(...,const uint64_t*,...)	GrB_Matrix_build_UINT64(...,const uint64_t*,...)
GrB_Matrix_build(...,const float*,...)	GrB_Matrix_build_FP32(...,const float*,...)
GrB_Matrix_build(...,const double*,...)	GrB_Matrix_build_FP64(...,const double*,...)
GrB_Matrix_build(...,const <i>other</i> *,...)	GrB_Matrix_build_UDT(...,const void*,...)
GrB_Matrix_setElement(...,GrB_Scalar,...)	GrB_Matrix_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Matrix_setElement(...,bool,...)	GrB_Matrix_setElement_BOOL(..., bool,...)
GrB_Matrix_setElement(...,int8_t,...)	GrB_Matrix_setElement_INT8(..., int8_t,...)
GrB_Matrix_setElement(...,uint8_t,...)	GrB_Matrix_setElement_UINT8(..., uint8_t,...)
GrB_Matrix_setElement(...,int16_t,...)	GrB_Matrix_setElement_INT16(..., int16_t,...)
GrB_Matrix_setElement(...,uint16_t,...)	GrB_Matrix_setElement_UINT16(..., uint16_t,...)
GrB_Matrix_setElement(...,int32_t,...)	GrB_Matrix_setElement_INT32(..., int32_t,...)
GrB_Matrix_setElement(...,uint32_t,...)	GrB_Matrix_setElement_UINT32(..., uint32_t,...)
GrB_Matrix_setElement(...,int64_t,...)	GrB_Matrix_setElement_INT64(..., int64_t,...)
GrB_Matrix_setElement(...,uint64_t,...)	GrB_Matrix_setElement_UINT64(..., uint64_t,...)
GrB_Matrix_setElement(...,float,...)	GrB_Matrix_setElement_FP32(..., float,...)
GrB_Matrix_setElement(...,double,...)	GrB_Matrix_setElement_FP64(..., double,...)
GrB_Matrix_setElement(..., <i>other</i> ,...)	GrB_Matrix_setElement_UDT(...,const void*,...)
GrB_Matrix_extractElement(GrB_Scalar,...)	GrB_Matrix_extractElement_Scalar(GrB_Scalar,...)
GrB_Matrix_extractElement(bool*,...)	GrB_Matrix_extractElement_BOOL(bool*,...)
GrB_Matrix_extractElement(int8_t*,...)	GrB_Matrix_extractElement_INT8(int8_t*,...)
GrB_Matrix_extractElement(uint8_t*,...)	GrB_Matrix_extractElement_UINT8(uint8_t*,...)
GrB_Matrix_extractElement(int16_t*,...)	GrB_Matrix_extractElement_INT16(int16_t*,...)
GrB_Matrix_extractElement(uint16_t*,...)	GrB_Matrix_extractElement_UINT16(uint16_t*,...)
GrB_Matrix_extractElement(int32_t*,...)	GrB_Matrix_extractElement_INT32(int32_t*,...)
GrB_Matrix_extractElement(uint32_t*,...)	GrB_Matrix_extractElement_UINT32(uint32_t*,...)
GrB_Matrix_extractElement(int64_t*,...)	GrB_Matrix_extractElement_INT64(int64_t*,...)
GrB_Matrix_extractElement(uint64_t*,...)	GrB_Matrix_extractElement_UINT64(uint64_t*,...)
GrB_Matrix_extractElement(float*,...)	GrB_Matrix_extractElement_FP32(float*,...)
GrB_Matrix_extractElement(double*,...)	GrB_Matrix_extractElement_FP64(double*,...)
GrB_Matrix_extractElement( <i>other</i> ,...)	GrB_Matrix_extractElement_UDT(void*,...)
GrB_Matrix_extractTuples(..., bool*,...)	GrB_Matrix_extractTuples_BOOL(..., bool*,...)
GrB_Matrix_extractTuples(..., int8_t*,...)	GrB_Matrix_extractTuples_INT8(..., int8_t*,...)
GrB_Matrix_extractTuples(..., uint8_t*,...)	GrB_Matrix_extractTuples_UINT8(..., uint8_t*,...)
GrB_Matrix_extractTuples(..., int16_t*,...)	GrB_Matrix_extractTuples_INT16(..., int16_t*,...)
GrB_Matrix_extractTuples(..., uint16_t*,...)	GrB_Matrix_extractTuples_UINT16(..., uint16_t*,...)
GrB_Matrix_extractTuples(..., int32_t*,...)	GrB_Matrix_extractTuples_INT32(..., int32_t*,...)
GrB_Matrix_extractTuples(..., uint32_t*,...)	GrB_Matrix_extractTuples_UINT32(..., uint32_t*,...)
GrB_Matrix_extractTuples(..., int64_t*,...)	GrB_Matrix_extractTuples_INT64(..., int64_t*,...)
GrB_Matrix_extractTuples(..., uint64_t*,...)	GrB_Matrix_extractTuples_UINT64(..., uint64_t*,...)
GrB_Matrix_extractTuples(..., float*,...)	GrB_Matrix_extractTuples_FP32(..., float*,...)
GrB_Matrix_extractTuples(..., double*,...)	GrB_Matrix_extractTuples_FP64(..., double*,...)
GrB_Matrix_extractTuples(..., <i>other</i> *,...)	GrB_Matrix_extractTuples_UDT(..., void*,...)

Table 5.5: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_import(...,const bool*,...)	GrB_Matrix_import_BOOL(...,const bool*,...)
GrB_Matrix_import(...,const int8_t*,...)	GrB_Matrix_import_INT8(...,const int8_t*,...)
GrB_Matrix_import(...,const uint8_t*,...)	GrB_Matrix_import_UINT8(...,const uint8_t*,...)
GrB_Matrix_import(...,const int16_t*,...)	GrB_Matrix_import_INT16(...,const int16_t*,...)
GrB_Matrix_import(...,const uint16_t*,...)	GrB_Matrix_import_UINT16(...,const uint16_t*,...)
GrB_Matrix_import(...,const int32_t*,...)	GrB_Matrix_import_INT32(...,const int32_t*,...)
GrB_Matrix_import(...,const uint32_t*,...)	GrB_Matrix_import_UINT32(...,const uint32_t*,...)
GrB_Matrix_import(...,const int64_t*,...)	GrB_Matrix_import_INT64(...,const int64_t*,...)
GrB_Matrix_import(...,const uint64_t*,...)	GrB_Matrix_import_UINT64(...,const uint64_t*,...)
GrB_Matrix_import(...,const float*,...)	GrB_Matrix_import_FP32(...,const float*,...)
GrB_Matrix_import(...,const double*,...)	GrB_Matrix_import_FP64(...,const double*,...)
GrB_Matrix_import(...,const other,...)	GrB_Matrix_import_UDT(...,const void*,...)
GrB_Matrix_export(...,bool*,...)	GrB_Matrix_export_BOOL(...,bool*,...)
GrB_Matrix_export(...,int8_t*,...)	GrB_Matrix_export_INT8(...,int8_t*,...)
GrB_Matrix_export(...,uint8_t*,...)	GrB_Matrix_export_UINT8(...,uint8_t*,...)
GrB_Matrix_export(...,int16_t*,...)	GrB_Matrix_export_INT16(...,int16_t*,...)
GrB_Matrix_export(...,uint16_t*,...)	GrB_Matrix_export_UINT16(...,uint16_t*,...)
GrB_Matrix_export(...,int32_t*,...)	GrB_Matrix_export_INT32(...,int32_t*,...)
GrB_Matrix_export(...,uint32_t*,...)	GrB_Matrix_export_UINT32(...,uint32_t*,...)
GrB_Matrix_export(...,int64_t*,...)	GrB_Matrix_export_INT64(...,int64_t*,...)
GrB_Matrix_export(...,uint64_t*,...)	GrB_Matrix_export_UINT64(...,uint64_t*,...)
GrB_Matrix_export(...,float*,...)	GrB_Matrix_export_FP32(...,float*,...)
GrB_Matrix_export(...,double*,...)	GrB_Matrix_export_FP64(...,double*,...)
GrB_Matrix_export(...,other,...)	GrB_Matrix_export_UDT(...,void*,...)
GrB_free(GrB_Type*)	GrB_Type_free(GrB_Type*)
GrB_free(GrB_UnaryOp*)	GrB_UnaryOp_free(GrB_UnaryOp*)
GrB_free(GrB_IndexUnaryOp*)	GrB_IndexUnaryOp_free(GrB_IndexUnaryOp*)
GrB_free(GrB_BinaryOp*)	GrB_BinaryOp_free(GrB_BinaryOp*)
GrB_free(GrB_Monoid*)	GrB_Monoid_free(GrB_Monoid*)
GrB_free(GrB_Semiring*)	GrB_Semiring_free(GrB_Semiring*)
GrB_free(GrB_Scalar*)	GrB_Scalar_free(GrB_Scalar*)
GrB_free(GrB_Vector*)	GrB_Vector_free(GrB_Vector*)
GrB_free(GrB_Matrix*)	GrB_Matrix_free(GrB_Matrix*)
GrB_free(GrB_Descriptor*)	GrB_Descriptor_free(GrB_Descriptor*)
GrB_wait(GrB_Type, GrB_WaitMode)	GrB_Type_wait(GrB_Type, GrB_WaitMode)
GrB_wait(GrB_UnaryOp, GrB_WaitMode)	GrB_UnaryOp_wait(GrB_UnaryOp, GrB_WaitMode)
GrB_wait(GrB_IndexUnaryOp, GrB_WaitMode)	GrB_IndexUnaryOp_wait(GrB_IndexUnaryOp, GrB_WaitMode)
GrB_wait(GrB_BinaryOp, GrB_WaitMode)	GrB_BinaryOp_wait(GrB_BinaryOp, GrB_WaitMode)
GrB_wait(GrB_Monoid, GrB_WaitMode)	GrB_Monoid_wait(GrB_Monoid, GrB_WaitMode)
GrB_wait(GrB_Semiring, GrB_WaitMode)	GrB_Semiring_wait(GrB_Semiring, GrB_WaitMode)
GrB_wait(GrB_Scalar, GrB_WaitMode)	GrB_Scalar_wait(GrB_Scalar, GrB_WaitMode)
GrB_wait(GrB_Vector, GrB_WaitMode)	GrB_Vector_wait(GrB_Vector, GrB_WaitMode)
GrB_wait(GrB_Matrix, GrB_WaitMode)	GrB_Matrix_wait(GrB_Matrix, GrB_WaitMode)
GrB_wait(GrB_Descriptor, GrB_WaitMode)	GrB_Descriptor_wait(GrB_Descriptor, GrB_WaitMode)
GrB_error(const char**, const GrB_Type)	GrB_Type_error(const char**, const GrB_Type)
GrB_error(const char**, const GrB_UnaryOp)	GrB_UnaryOp_error(const char**, const GrB_UnaryOp)
GrB_error(const char**, const GrB_IndexUnaryOp)	GrB_IndexUnaryOp_error(const char**, const GrB_IndexUnaryOp)
GrB_error(const char**, const GrB_BinaryOp)	GrB_BinaryOp_error(const char**, const GrB_BinaryOp)
GrB_error(const char**, const GrB_Monoid)	GrB_Monoid_error(const char**, const GrB_Monoid)
GrB_error(const char**, const GrB_Semiring)	GrB_Semiring_error(const char**, const GrB_Semiring)
GrB_error(const char**, const GrB_Scalar)	GrB_Scalar_error(const char**, const GrB_Scalar)
GrB_error(const char**, const GrB_Vector)	GrB_Vector_error(const char**, const GrB_Vector)
GrB_error(const char**, const GrB_Matrix)	GrB_Matrix_error(const char**, const GrB_Matrix)
GrB_error(const char**, const GrB_Descriptor)	GrB_Descriptor_error(const char**, const GrB_Descriptor)

Table 5.6: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_eWiseMult(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseMult_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseMult_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseMult_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseMult_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseMult_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseMult_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseAdd_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseAdd_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseAdd_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseAdd_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseAdd_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseAdd_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_extract(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_extract(GrB_Vector,...,GrB_Vector,...)
GrB_extract(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_extract(GrB_Matrix,...,GrB_Matrix,...)
GrB_extract(GrB_Vector,...,GrB_Matrix,...)	GrB_Col_extract(GrB_Vector,...,GrB_Matrix,...)
GrB_assign(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_assign(GrB_Vector,...,GrB_Vector,...)
GrB_assign(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_assign(GrB_Matrix,...,GrB_Matrix,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)	GrB_Col_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)	GrB_Row_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)
GrB_assign(GrB_Vector,...,GrB_Scalar,...)	GrB_Vector_assign_Scalar(GrB_Vector,...,const GrB_Scalar,...)
GrB_assign(GrB_Vector,...,bool,...)	GrB_Vector_assign_BOOL(GrB_Vector,..., bool,...)
GrB_assign(GrB_Vector,...,int8_t,...)	GrB_Vector_assign_INT8(GrB_Vector,..., int8_t,...)
GrB_assign(GrB_Vector,...,uint8_t,...)	GrB_Vector_assign_UINT8(GrB_Vector,..., uint8_t,...)
GrB_assign(GrB_Vector,...,int16_t,...)	GrB_Vector_assign_INT16(GrB_Vector,..., int16_t,...)
GrB_assign(GrB_Vector,...,uint16_t,...)	GrB_Vector_assign_UINT16(GrB_Vector,..., uint16_t,...)
GrB_assign(GrB_Vector,...,int32_t,...)	GrB_Vector_assign_INT32(GrB_Vector,..., int32_t,...)
GrB_assign(GrB_Vector,...,uint32_t,...)	GrB_Vector_assign_UINT32(GrB_Vector,..., uint32_t,...)
GrB_assign(GrB_Vector,...,int64_t,...)	GrB_Vector_assign_INT64(GrB_Vector,..., int64_t,...)
GrB_assign(GrB_Vector,...,uint64_t,...)	GrB_Vector_assign_UINT64(GrB_Vector,..., uint64_t,...)
GrB_assign(GrB_Vector,...,float,...)	GrB_Vector_assign_FP32(GrB_Vector,..., float,...)
GrB_assign(GrB_Vector,...,double,...)	GrB_Vector_assign_FP64(GrB_Vector,..., double,...)
GrB_assign(GrB_Vector,...,other,...)	GrB_Vector_assign_UDT(GrB_Vector,...,const void*,...)
GrB_assign(GrB_Matrix,...,GrB_Scalar,...)	GrB_Matrix_assign_Scalar(GrB_Matrix,...,const GrB_Scalar,...)
GrB_assign(GrB_Matrix,...,bool,...)	GrB_Matrix_assign_BOOL(GrB_Matrix,..., bool,...)
GrB_assign(GrB_Matrix,...,int8_t,...)	GrB_Matrix_assign_INT8(GrB_Matrix,..., int8_t,...)
GrB_assign(GrB_Matrix,...,uint8_t,...)	GrB_Matrix_assign_UINT8(GrB_Matrix,..., uint8_t,...)
GrB_assign(GrB_Matrix,...,int16_t,...)	GrB_Matrix_assign_INT16(GrB_Matrix,..., int16_t,...)
GrB_assign(GrB_Matrix,...,uint16_t,...)	GrB_Matrix_assign_UINT16(GrB_Matrix,..., uint16_t,...)
GrB_assign(GrB_Matrix,...,int32_t,...)	GrB_Matrix_assign_INT32(GrB_Matrix,..., int32_t,...)
GrB_assign(GrB_Matrix,...,uint32_t,...)	GrB_Matrix_assign_UINT32(GrB_Matrix,..., uint32_t,...)
GrB_assign(GrB_Matrix,...,int64_t,...)	GrB_Matrix_assign_INT64(GrB_Matrix,..., int64_t,...)
GrB_assign(GrB_Matrix,...,uint64_t,...)	GrB_Matrix_assign_UINT64(GrB_Matrix,..., uint64_t,...)
GrB_assign(GrB_Matrix,...,float,...)	GrB_Matrix_assign_FP32(GrB_Matrix,..., float,...)
GrB_assign(GrB_Matrix,...,double,...)	GrB_Matrix_assign_FP64(GrB_Matrix,..., double,...)
GrB_assign(GrB_Matrix,...,other,...)	GrB_Matrix_assign_UDT(GrB_Matrix,...,const void*,...)

Table 5.7: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)	GrB_Vector_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)
GrB_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)	GrB_Matrix_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_BOOL(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT8(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT8(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT16(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT16(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT32(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT32(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT64(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT64(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP32(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP64(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp, <i>other</i> ,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UDT(GrB_Vector,...,GrB_BinaryOp,const void*,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_BinaryOp2nd_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_BinaryOp2nd_BOOL(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_BinaryOp2nd_INT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_BinaryOp2nd_INT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_BinaryOp2nd_INT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_BinaryOp2nd_INT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)	GrB_Vector_apply_BinaryOp2nd_FP32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)	GrB_Vector_apply_BinaryOp2nd_FP64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_BinaryOp2nd_UDT(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,const void*,...)

Table 5.8: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_BOOL(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT8(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT8(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT16(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT16(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT32(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT32(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT64(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT64(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP32(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP64(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp, <i>other</i> ,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UDT(GrB_Matrix,...,GrB_BinaryOp,const void*,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_BinaryOp2nd_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_BinaryOp2nd_BOOL(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_BinaryOp2nd_FP32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_BinaryOp2nd_FP64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_BinaryOp2nd_UDT(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,const void*,...)

Table 5.9: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_IndexOp_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_IndexOp_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_IndexOp_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_IndexOp_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_IndexOp_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_IndexOp_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_IndexOp_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_IndexOp_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_IndexOp_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_IndexOp_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)	GrB_Vector_apply_IndexOp_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)	GrB_Vector_apply_IndexOp_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_IndexOp_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_IndexOp_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_IndexOp_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_IndexOp_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_IndexOp_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_IndexOp_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_IndexOp_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_IndexOp_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_IndexOp_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_IndexOp_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_IndexOp_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_IndexOp_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_IndexOp_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_IndexOp_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)

Table 5.10: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>	<code>GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>	<code>GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>	<code>GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>	<code>GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>	<code>GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>	<code>GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>	<code>GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>	<code>GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>	<code>GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>	<code>GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>	<code>GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>	<code>GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)</code>	<code>GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>	<code>GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>	<code>GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>	<code>GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>	<code>GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>	<code>GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>	<code>GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>	<code>GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>	<code>GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>	<code>GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>	<code>GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>	<code>GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>	<code>GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)</code>	<code>GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)</code>



Table 5.11: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_reduce(GrB_Vector,...,GrB_Monoid,...)	GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Vector,...)	GrB_Vector_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Vector,...)	GrB_Vector_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(bool*,...,GrB_Vector,...)	GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)
GrB_reduce(int8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)
GrB_reduce(uint8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)
GrB_reduce(int16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)
GrB_reduce(uint16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)
GrB_reduce(int32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)
GrB_reduce(uint32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)
GrB_reduce(int64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)
GrB_reduce(uint64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)
GrB_reduce(float*,...,GrB_Vector,...)	GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)
GrB_reduce(double*,...,GrB_Vector,...)	GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)
GrB_reduce( <i>other</i> *,...,GrB_Vector,...)	GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)	GrB_Matrix_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)	GrB_Matrix_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)
GrB_reduce(bool*,...,GrB_Matrix,...)	GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)
GrB_reduce(int8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)
GrB_reduce(uint8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)
GrB_reduce(int16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)
GrB_reduce(uint16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)
GrB_reduce(int32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)
GrB_reduce(uint32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)
GrB_reduce(int64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)
GrB_reduce(uint64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)
GrB_reduce(float*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)
GrB_reduce(double*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)
GrB_reduce( <i>other</i> *,...,GrB_Matrix,...)	GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)
GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)

Table 5.12: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_get(GrB_Scalar,GrB_Scalar,GrB_Field)	GrB_Scalar_get_Scalar(GrB_Scalar,GrB_Scalar,GrB_Field)
GrB_get(GrB_Scalar,char*,GrB_Field)	GrB_Scalar_get_String(GrB_Scalar,char*,GrB_Field)
GrB_get(GrB_Scalar,int32_t*,GrB_Field)	GrB_Scalar_get_INT32(GrB_Scalar,int32_t*,GrB_Field)
GrB_get(GrB_Scalar,size_t*,GrB_Field)	GrB_Scalar_get_SIZE(GrB_Scalar,size_t*,GrB_Field)
GrB_get(GrB_Scalar,void*,GrB_Field)	GrB_Scalar_get_VOID(GrB_Scalar,void*,GrB_Field)
GrB_get(GrB_Vector,GrB_Scalar,GrB_Field)	GrB_Vector_get_Scalar(GrB_Vector,GrB_Scalar,GrB_Field)
GrB_get(GrB_Vector,char*,GrB_Field)	GrB_Vector_get_String(GrB_Vector,char*,GrB_Field)
GrB_get(GrB_Vector,int32_t*,GrB_Field)	GrB_Vector_get_INT32(GrB_Vector,int32_t*,GrB_Field)
GrB_get(GrB_Vector,size_t*,GrB_Field)	GrB_Vector_get_SIZE(GrB_Vector,size_t*,GrB_Field)
GrB_get(GrB_Vector,void*,GrB_Field)	GrB_Vector_get_VOID(GrB_Vector,void*,GrB_Field)
GrB_get(GrB_Matrix,GrB_Scalar,GrB_Field)	GrB_Matrix_get_Scalar(GrB_Matrix,GrB_Scalar,GrB_Field)
GrB_get(GrB_Matrix,char*,GrB_Field)	GrB_Matrix_get_String(GrB_Matrix,char*,GrB_Field)
GrB_get(GrB_Matrix,int32_t*,GrB_Field)	GrB_Matrix_get_INT32(GrB_Matrix,int32_t*,GrB_Field)
GrB_get(GrB_Matrix,size_t*,GrB_Field)	GrB_Matrix_get_SIZE(GrB_Matrix,size_t*,GrB_Field)
GrB_get(GrB_Matrix,void*,GrB_Field)	GrB_Matrix_get_VOID(GrB_Matrix,void*,GrB_Field)
GrB_get(GrB_UnaryOp,GrB_Scalar,GrB_Field)	GrB_UnaryOp_get_Scalar(GrB_UnaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_UnaryOp,char*,GrB_Field)	GrB_UnaryOp_get_String(GrB_UnaryOp,char*,GrB_Field)
GrB_get(GrB_UnaryOp,int32_t*,GrB_Field)	GrB_UnaryOp_get_INT32(GrB_UnaryOp,int32_t*,GrB_Field)
GrB_get(GrB_UnaryOp,size_t*,GrB_Field)	GrB_UnaryOp_get_SIZE(GrB_UnaryOp,size_t*,GrB_Field)
GrB_get(GrB_UnaryOp,void*,GrB_Field)	GrB_UnaryOp_get_VOID(GrB_UnaryOp,void*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)	GrB_IndexUnaryOp_get_Scalar(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_IndexUnaryOp,char*,GrB_Field)	GrB_IndexUnaryOp_get_String(GrB_IndexUnaryOp,char*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,int32_t*,GrB_Field)	GrB_IndexUnaryOp_get_INT32(GrB_IndexUnaryOp,int32_t*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,size_t*,GrB_Field)	GrB_IndexUnaryOp_get_SIZE(GrB_IndexUnaryOp,size_t*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,void*,GrB_Field)	GrB_IndexUnaryOp_get_VOID(GrB_IndexUnaryOp,void*,GrB_Field)
GrB_get(GrB_BinaryOp,GrB_Scalar,GrB_Field)	GrB_BinaryOp_get_Scalar(GrB_BinaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_BinaryOp,char*,GrB_Field)	GrB_BinaryOp_get_String(GrB_BinaryOp,char*,GrB_Field)
GrB_get(GrB_BinaryOp,int32_t*,GrB_Field)	GrB_BinaryOp_get_INT32(GrB_BinaryOp,int32_t*,GrB_Field)
GrB_get(GrB_BinaryOp,size_t*,GrB_Field)	GrB_BinaryOp_get_SIZE(GrB_BinaryOp,size_t*,GrB_Field)
GrB_get(GrB_BinaryOp,void*,GrB_Field)	GrB_BinaryOp_get_VOID(GrB_BinaryOp,void*,GrB_Field)
GrB_get(GrB_Monoid,GrB_Scalar,GrB_Field)	GrB_Monoid_get_Scalar(GrB_Monoid,GrB_Scalar,GrB_Field)
GrB_get(GrB_Monoid,char*,GrB_Field)	GrB_Monoid_get_String(GrB_Monoid,char*,GrB_Field)
GrB_get(GrB_Monoid,int32_t*,GrB_Field)	GrB_Monoid_get_INT32(GrB_Monoid,int32_t*,GrB_Field)
GrB_get(GrB_Monoid,size_t*,GrB_Field)	GrB_Monoid_get_SIZE(GrB_Monoid,size_t*,GrB_Field)
GrB_get(GrB_Monoid,void*,GrB_Field)	GrB_Monoid_get_VOID(GrB_Monoid,void*,GrB_Field)
GrB_get(GrB_Semiring,GrB_Scalar,GrB_Field)	GrB_Semiring_get_Scalar(GrB_Semiring,GrB_Scalar,GrB_Field)
GrB_get(GrB_Semiring,char*,GrB_Field)	GrB_Semiring_get_String(GrB_Semiring,char*,GrB_Field)
GrB_get(GrB_Semiring,int32_t*,GrB_Field)	GrB_Semiring_get_INT32(GrB_Semiring,int32_t*,GrB_Field)
GrB_get(GrB_Semiring,size_t*,GrB_Field)	GrB_Semiring_get_SIZE(GrB_Semiring,size_t*,GrB_Field)
GrB_get(GrB_Semiring,void*,GrB_Field)	GrB_Semiring_get_VOID(GrB_Semiring,void*,GrB_Field)
GrB_get(GrB_Descriptor,GrB_Scalar,GrB_Field)	GrB_Descriptor_get_Scalar(GrB_Descriptor,GrB_Scalar,GrB_Field)
GrB_get(GrB_Descriptor,char*,GrB_Field)	GrB_Descriptor_get_String(GrB_Descriptor,char*,GrB_Field)
GrB_get(GrB_Descriptor,int32_t*,GrB_Field)	GrB_Descriptor_get_INT32(GrB_Descriptor,int32_t*,GrB_Field)
GrB_get(GrB_Descriptor,size_t*,GrB_Field)	GrB_Descriptor_get_SIZE(GrB_Descriptor,size_t*,GrB_Field)
GrB_get(GrB_Descriptor,void*,GrB_Field)	GrB_Descriptor_get_VOID(GrB_Descriptor,void*,GrB_Field)
GrB_get(GrB_Type,GrB_Scalar,GrB_Field)	GrB_Type_get_Scalar(GrB_Type,GrB_Scalar,GrB_Field)
GrB_get(GrB_Type,char*,GrB_Field)	GrB_Type_get_String(GrB_Type,char*,GrB_Field)
GrB_get(GrB_Type,int32_t*,GrB_Field)	GrB_Type_get_INT32(GrB_Type,int32_t*,GrB_Field)
GrB_get(GrB_Type,size_t*,GrB_Field)	GrB_Type_get_SIZE(GrB_Type,size_t*,GrB_Field)
GrB_get(GrB_Type,void*,GrB_Field)	GrB_Type_get_VOID(GrB_Type,void*,GrB_Field)
GrB_get(GrB_Global,GrB_Scalar,GrB_Field)	GrB_Global_get_Scalar(GrB_Global,GrB_Scalar,GrB_Field)
GrB_get(GrB_Global,char*,GrB_Field)	GrB_Global_get_String(GrB_Global,char*,GrB_Field)
GrB_get(GrB_Global,int32_t*,GrB_Field)	GrB_Global_get_INT32(GrB_Global,int32_t*,GrB_Field)
GrB_get(GrB_Global,size_t*,GrB_Field)	GrB_Global_get_SIZE(GrB_Global,size_t*,GrB_Field)
GrB_get(GrB_Global,void*,GrB_Field)	GrB_Global_get_VOID(GrB_Global,void*,GrB_Field)

Table 5.13: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_set(GrB_Scalar,GrB_Scalar,GrB_Field)	GrB_Scalar_set_Scalar(GrB_Scalar,GrB_Scalar,GrB_Field)
GrB_set(GrB_Scalar,char*,GrB_Field)	GrB_Scalar_set_String(GrB_Scalar,char*,GrB_Field)
GrB_set(GrB_Scalar,int32_t,GrB_Field)	GrB_Scalar_set_INT32(GrB_Scalar,int32_t,GrB_Field)
GrB_set(GrB_Scalar,void*,GrB_Field,size_t)	GrB_Scalar_set_VOID(GrB_Scalar,void*,GrB_Field,size_t)
GrB_set(GrB_Vector,GrB_Scalar,GrB_Field)	GrB_Vector_set_Scalar(GrB_Vector,GrB_Scalar,GrB_Field)
GrB_set(GrB_Vector,char*,GrB_Field)	GrB_Vector_set_String(GrB_Vector,char*,GrB_Field)
GrB_set(GrB_Vector,int32_t,GrB_Field)	GrB_Vector_set_INT32(GrB_Vector,int32_t,GrB_Field)
GrB_set(GrB_Vector,void*,GrB_Field,size_t)	GrB_Vector_set_VOID(GrB_Vector,void*,GrB_Field,size_t)
GrB_set(GrB_Matrix,GrB_Scalar,GrB_Field)	GrB_Matrix_set_Scalar(GrB_Matrix,GrB_Scalar,GrB_Field)
GrB_set(GrB_Matrix,char*,GrB_Field)	GrB_Matrix_set_String(GrB_Matrix,char*,GrB_Field)
GrB_set(GrB_Matrix,int32_t,GrB_Field)	GrB_Matrix_set_INT32(GrB_Matrix,int32_t,GrB_Field)
GrB_set(GrB_Matrix,void*,GrB_Field,size_t)	GrB_Matrix_set_VOID(GrB_Matrix,void*,GrB_Field,size_t)
GrB_set(GrB_UnaryOp,GrB_Scalar,GrB_Field)	GrB_UnaryOp_set_Scalar(GrB_UnaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_UnaryOp,char*,GrB_Field)	GrB_UnaryOp_set_String(GrB_UnaryOp,char*,GrB_Field)
GrB_set(GrB_UnaryOp,int32_t,GrB_Field)	GrB_UnaryOp_set_INT32(GrB_UnaryOp,int32_t,GrB_Field)
GrB_set(GrB_UnaryOp,void*,GrB_Field,size_t)	GrB_UnaryOp_set_VOID(GrB_UnaryOp,void*,GrB_Field,size_t)
GrB_set(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)	GrB_IndexUnaryOp_set_Scalar(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_IndexUnaryOp,char*,GrB_Field)	GrB_IndexUnaryOp_set_String(GrB_IndexUnaryOp,char*,GrB_Field)
GrB_set(GrB_IndexUnaryOp,int32_t,GrB_Field)	GrB_IndexUnaryOp_set_INT32(GrB_IndexUnaryOp,int32_t,GrB_Field)
GrB_set(GrB_IndexUnaryOp,void*,GrB_Field,size_t)	GrB_IndexUnaryOp_set_VOID(GrB_IndexUnaryOp,void*,GrB_Field,size_t)
GrB_set(GrB_BinaryOp,GrB_Scalar,GrB_Field)	GrB_BinaryOp_set_Scalar(GrB_BinaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_BinaryOp,char*,GrB_Field)	GrB_BinaryOp_set_String(GrB_BinaryOp,char*,GrB_Field)
GrB_set(GrB_BinaryOp,int32_t,GrB_Field)	GrB_BinaryOp_set_INT32(GrB_BinaryOp,int32_t,GrB_Field)
GrB_set(GrB_BinaryOp,void*,GrB_Field,size_t)	GrB_BinaryOp_set_VOID(GrB_BinaryOp,void*,GrB_Field,size_t)
GrB_set(GrB_Monoid,GrB_Scalar,GrB_Field)	GrB_Monoid_set_Scalar(GrB_Monoid,GrB_Scalar,GrB_Field)
GrB_set(GrB_Monoid,char*,GrB_Field)	GrB_Monoid_set_String(GrB_Monoid,char*,GrB_Field)
GrB_set(GrB_Monoid,int32_t,GrB_Field)	GrB_Monoid_set_INT32(GrB_Monoid,int32_t,GrB_Field)
GrB_set(GrB_Monoid,void*,GrB_Field,size_t)	GrB_Monoid_set_VOID(GrB_Monoid,void*,GrB_Field,size_t)
GrB_set(GrB_Semiring,GrB_Scalar,GrB_Field)	GrB_Semiring_set_Scalar(GrB_Semiring,GrB_Scalar,GrB_Field)
GrB_set(GrB_Semiring,char*,GrB_Field)	GrB_Semiring_set_String(GrB_Semiring,char*,GrB_Field)
GrB_set(GrB_Semiring,int32_t,GrB_Field)	GrB_Semiring_set_INT32(GrB_Semiring,int32_t,GrB_Field)
GrB_set(GrB_Semiring,void*,GrB_Field,size_t)	GrB_Semiring_set_VOID(GrB_Semiring,void*,GrB_Field,size_t)
GrB_set(GrB_Descriptor,GrB_Scalar,GrB_Field)	GrB_Descriptor_set_Scalar(GrB_Descriptor,GrB_Scalar,GrB_Field)
GrB_set(GrB_Descriptor,char*,GrB_Field)	GrB_Descriptor_set_String(GrB_Descriptor,char*,GrB_Field)
GrB_set(GrB_Descriptor,int32_t,GrB_Field)	GrB_Descriptor_set_INT32(GrB_Descriptor,int32_t,GrB_Field)
GrB_set(GrB_Descriptor,void*,GrB_Field,size_t)	GrB_Descriptor_set_VOID(GrB_Descriptor,void*,GrB_Field,size_t)
GrB_set(GrB_Type,GrB_Scalar,GrB_Field)	GrB_Type_set_Scalar(GrB_Type,GrB_Scalar,GrB_Field)
GrB_set(GrB_Type,char*,GrB_Field)	GrB_Type_set_String(GrB_Type,char*,GrB_Field)
GrB_set(GrB_Type,int32_t,GrB_Field)	GrB_Type_set_INT32(GrB_Type,int32_t,GrB_Field)
GrB_set(GrB_Type,void*,GrB_Field,size_t)	GrB_Type_set_VOID(GrB_Type,void*,GrB_Field,size_t)
GrB_set(GrB_Global,GrB_Scalar,GrB_Field)	GrB_Global_set_Scalar(GrB_Global,GrB_Scalar,GrB_Field)
GrB_set(GrB_Global,char*,GrB_Field)	GrB_Global_set_String(GrB_Global,char*,GrB_Field)
GrB_set(GrB_Global,int32_t,GrB_Field)	GrB_Global_set_INT32(GrB_Global,int32_t,GrB_Field)
GrB_set(GrB_Global,void*,GrB_Field,size_t)	GrB_Global_set_VOID(GrB_Global,void*,GrB_Field,size_t)



# 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.

- 7549 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified  
7550 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats  
7551 have been deferred.
- 7552 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7553 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return  
7554 error if duplicate locations encountered.
- 7555 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7556 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7557 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type T (not to  
7558 be confused with the proposed unary operator).
- 7559 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added  
7560 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to  
7561 a method that is not supported by the implementation.
- 7562 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque  
7563 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7564 • (Issue BB-45) Removed language about annihilators.
- 7565 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7566 • Updated a number algorithms in the appendix to use new operations and methods.
- 7567 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the  
7568 specification.
- 7569 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and  
7570 `GRB_SUBVERSION`.
- 7571 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7572 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection  
7573 fixed to read `eWiseAdd`/set union.
- 7574 • Typographical error (`NEQ`  $\rightarrow$  `NE`) in Description of Table 3.8.

7575 Changes in 1.3.0 (Released: 25 September 2019):

- 7576 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque  
7577 GraphBLAS object as an argument.
- 7578 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7579 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a  
7580 scalar.

7581  
7582  
7583  
7584  
7585  
7586  
7587  
7588  
7589  
7590  
7591  
7592  
7593  
7594  
7595  
7596  
7597  
7598  
7599  
7600  
7601  
7602  
7603  
7604  
7605  
7606  
7607  
7608  
7609  
7610  
7611  
7612  
7613

- (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.

- 7614 • Fixed miscellaneous typographical errors (such as  $\otimes$ ,  $\oplus$ ).
- 7615 Changes in 1.2.0:
- 7616 • Removed "provisional" clause.
- 7617 Changes in 1.1.0:
- 7618 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
  - 7619 `assign` operations.
  - 7620 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
  - 7621 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
  - 7622 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
  - 7623 from the domains of the binary operator provided.
  - 7624 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
  - 7625 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
  - 7626 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
  - 7627 which is returned when the capacities of the output arrays are insufficient to hold all of the
  - 7628 tuples.
  - 7629 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
  - 7630 face.
  - 7631 • Added replace flag (`z`) notation to Table 4.1.
  - 7632 • Updated the “Mathematical Description” of the `assign` operation in Table 4.1.
  - 7633 • Added triangle counting example.
  - 7634 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
  - 7635 sections of GraphBLAS operations when the respective text was the “standard” text (i.e.,
  - 7636 identical in a majority of the operations).
  - 7637 • Fixed typographical errors.
- 7638 Changes in 1.0.2:
- 7639 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
  - 7640 matrices and avoid casting issues in certain implementations.
  - 7641 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
  - 7642 erased outside the assigned area.
  - 7643 • Changes non-polymorphic interface:
    - 7644 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.



- 7645           – Renamed GrB\_Vector\_reduce\_BinaryOp to GrB\_Matrix\_reduce\_BinaryOp.
- 7646           – Renamed GrB\_Vector\_reduce\_Monoid to GrB\_Matrix\_reduce\_Monoid.
- 7647       • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7648       • 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.

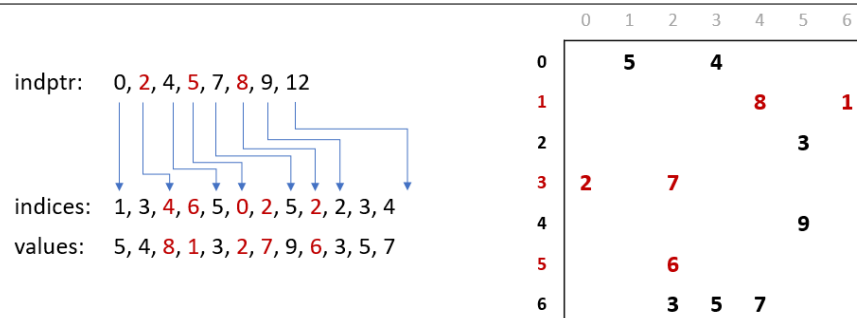


Figure B.1: Data layout for CSR format.

### 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.

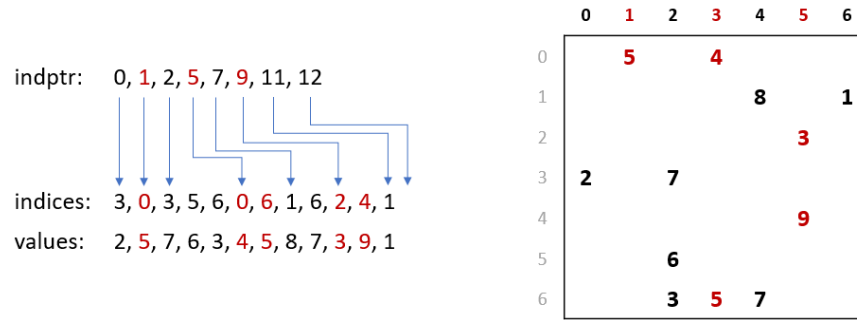


Figure B.2: Data layout for CSC format.

### 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.

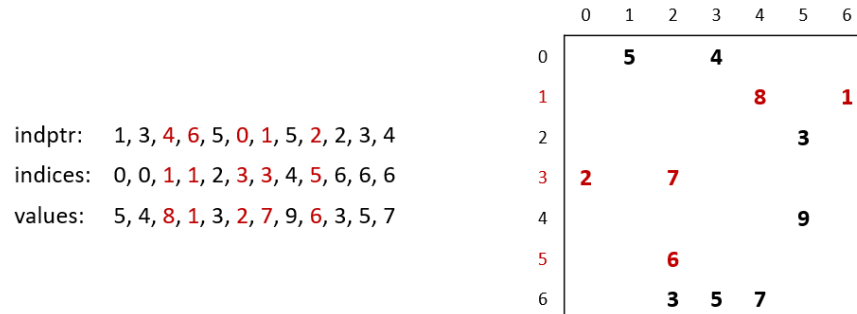


Figure B.3: Data layout for COO format.

7681 **Appendix C**

7682 **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);    //  $q[!v] = q \parallel A$ ; finds all the
35                                      // unvisited successors from current  $q$ 
36          GrB_Vector_nvals(&nvals, q);
37      } while (nvals);                 // if there is no successor in  $q$ , we are done.
38
39      GrB_free(&q);                    //  $q$  vector no longer needed
40
41      return GrB_SUCCESS;
42  }

```

## C.3 Example: Parent BFS in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a binary  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS
9   * traversal of the graph and sets  $parents[i]$  to the index of vertex  $i$ 's parent.
10  * The parent of the root vertex,  $s$ , will be set to itself ( $parents[s] = s$ ). If
11  * vertex  $i$  is not reachable from  $s$ ,  $parents[i]$  will not contain a stored value.
12  */
13  GrB_Info BFS(GrB_Vector *parents, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index N;
16      GrB_Matrix_nrows(&N, A);           //  $N = \#$  vertices
17
18      GrB_Vector_new(parents, GrB_UINT64, N);
19      GrB_Vector_setElement(*parents, s, s);           //  $parents[s] = s$ 
20
21      GrB_Vector wavefront;
22      GrB_Vector_new(&wavefront, GrB_UINT64, N);
23      GrB_Vector_setElement(wavefront, 1UL, s);       //  $wavefront[s] = 1$ 
24
25      /*
26       * BFS traversal and label the vertices.
27       */
28      GrB_Index nvals;
29      GrB_Vector_nvals(&nvals, wavefront);
30
31      while (nvals > 0)
32      {
33          // convert all stored values in wavefront to their 0-based index
34          GrB_apply(wavefront, GrB_NULL, GrB_NULL, GrB_ROWINDEX_INT64,
35                  wavefront, 0UL, GrB_NULL);
36
37          // "FIRST" because left-multiplying wavefront rows. Masking out the parent
38          // list ensures wavefront values do not overwrite parents already stored.
39          GrB_vxm(wavefront, *parents, GrB_NULL, GrB_MIN_FIRST_SEMIRING_UINT64,
40                  wavefront, A, GrB_DESC_RSC);
41
42          // Don't need to mask here since we did it in vxm. Merges new parents in
43          // current wavefront with existing parents:  $parents += wavefront$ 
44          GrB_apply(*parents, GrB_NULL, GrB_PLUS_UINT64,
45                  GrB_IDENTITY_UINT64, wavefront, GrB_NULL);
46
47          GrB_Vector_nvals(&nvals, wavefront);
48      }
49
50      GrB_free(&wavefront);
51
52      return GrB_SUCCESS;
53  }

```



## C.4 Example: Betweenness centrality (BC) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ ,
9   * compute the BC-metric vector  $\delta$ , which should be empty on input.
10  */
11  GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
12  {
13      GrB_Index n;
14      GrB_Matrix_nrows(&n, A);                      //  $n = \#$  of vertices in graph
15
16      GrB_Vector_new(delta, GrB_FP32, n);            // Vector<float>  $\delta(n)$ 
17
18      GrB_Matrix sigma;
19      GrB_Matrix_new(&sigma, GrB_INT32, n, n);        // Matrix<int32_t>  $\sigma(n, n)$ 
20                                                         //  $\sigma[d, k] = \#$  shortest paths to node  $k$  at level  $d$ 
21
22      GrB_Vector q;
23      GrB_Vector_new(&q, GrB_INT32, n);              // Vector<int32_t>  $q(n)$  of path counts
24                                                         //  $q[s] = 1$ 
25
26      GrB_Vector p;
27      GrB_Vector_dup(&p, q);                          // Vector<int32_t>  $p(n)$  shortest path counts so far
28                                                         //  $p = q$ 
29
30      GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
31              q, A, GrB_DESC_RC);                    // get the first set of out neighbors
32
33      /*
34       * BFS phase
35       */
36      GrB_Index d = 0;                                // BFS level number
37      int32_t sum = 0;                                // sum == 0 when BFS phase is complete
38
39      do {
40          GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\sigma[d, :] = q$ 
41          GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
42          GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
43                  q, A, GrB_DESC_RC);                //  $q = \#$  paths to nodes reachable
44                                                         // from current level
45          GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
46          ++d;
47      } while (sum);
48
49      /*
50       * BC computation phase
51       * ( $t1, t2, t3, t4$ ) are temporary vectors
52       */
53      GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
54      GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
55      GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
56      GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
57
58      for (int i=d-1; i>0; i--)
59      {
60          GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t1 = 1 + \delta$ 
61          GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, t1, *delta, GrB_NULL);
62          GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t2 = \sigma[i, :]$ 
63          GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t2 = (1 + \delta) / \sigma[i, :]$ 
64          GrB_mvx(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
65                  // add contributions made by

```

```

63         A, t2, GrB_NULL);
64     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, GrB_DESC_T0); // t4 = sigma[i-1,:]
65     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, GrB_TIMES_FP32, t4, t3, GrB_NULL); // t4 = sigma[i-1,:]*t3
66     GrB_eWiseAdd(delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, delta, t4, GrB_NULL); // accumulate into delta
67 }
68
69 GrB_free(&sigma);
70 GrB_free(&q); GrB_free(&p);
71 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
72
73 return GrB_SUCCESS;
74 }

```

## C.5 Example: Batched BC in GraphBLAS

```

1  #include <stdlib.h>
2  #include "GraphBLAS.h" // in addition to other required C headers
3
4  // Compute partial BC metric for a subset of source vertices, s, in graph A
5  GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
6  {
7      GrB_Index n;
8      GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
9      GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
10
11     // index and value arrays needed to build numsp
12     GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
13     int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
14     for(int i=0; i<nsver; ++i) {
15         i_nsver[i] = i;
16         ones[i] = 1;
17     }
18
19     // numsp: structure holds the number of shortest paths for each node and starting vertex
20     // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
21     GrB_Matrix numsp;
22     GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
23     GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
24     free(i_nsver); free(ones);
25
26     // frontier: Holds the current frontier where values are path counts.
27     // Initialized to out vertices of each source node in s.
28     GrB_Matrix frontier;
29     GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
30     GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, GrB_DESC_RCT0);
31
32     // sigma: stores frontier information for each level of BFS phase. The memory
33     // for an entry in sigmas is only allocated within the do-while loop if needed.
34     // n is an upper bound on diameter.
35     GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n);
36
37     int32_t d = 0; // BFS level number
38     GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
39
40     // ----- The BFS phase (forward sweep) -----
41     do {
42         // sigmas[d](:,s) = dth level frontier from source vertex s
43         GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
44
45         GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
46                 GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
47         GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL, GrB_PLUS_INT32,
48                     numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
49         GrB_mxm(frontier, numsp, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
50                 A, frontier, GrB_DESC_RCT0); // f<!numsp> = A' +.* f (update frontier)
51         GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level
52         d++;
53     } while (nvals);
54
55     // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
56     GrB_Matrix nspinv;
57     GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
58     GrB_apply(nspinv, GrB_NULL, GrB_NULL,
59              GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
60
61     // bcu: BC updates for each vertex for each starting vertex in s
62     GrB_Matrix bcu;

```

```

63 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
64 GrB_assign(bcu, GrB_NULL, GrB_NULL,
65           1.0f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
66
67 GrB_Matrix w; // temporary workspace matrix
68 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
69
70 // ----- Tally phase (backward sweep) -----
71 for (int i=d-1; i>0; i--) {
72     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
73                 GrB_TIMES_FP32, bcu, nspinv, GrB_DESC_R); // w<sigmas[i]>=(1 ./ nsp).*bcu
74
75     // add contributions by successors and mask with that BFS level's frontier
76     GrB_mxm(w, sigmas[i-1], GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
77            A, w, GrB_DESC_R); // w<sigmas[i-1]> = (A +.* w)
78     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32, GrB_TIMES_FP32,
79                 w, numsp, GrB_NULL); // bcu += w .* numsp
80 }
81
82 // row reduce bcu and subtract "nsver" from every entry to account
83 // for 1 extra value per bcu row element.
84 GrB_reduce(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, bcu, GrB_NULL);
85 GrB_apply(*delta, GrB_NULL, GrB_NULL, GrB_MINUS_FP32, *delta, (float)nsver, GrB_NULL);
86
87 // Release resources
88 for (int i=0; i<d; i++) {
89     GrB_free(&(sigmas[i]));
90 }
91 free(sigmas);
92
93 GrB_free(&frontier); GrB_free(&numsp);
94 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
95
96 return GrB_SUCCESS;
97 }

```

## C.6 Example: Maximal independent set (MIS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  // Assign a random number to each element scaled by the inverse of the node's degree.
8  // This will increase the probability that low degree nodes are selected and larger
9  // sets are selected.
10 void setRandom(void *out, const void *in)
11 {
12     uint32_t degree = *(uint32_t*)in;
13     *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
14 }
15
16 /*
17  * A variant of Luby's randomized algorithm [Luby 1985].
18  *
19  * Given a numeric n x n adjacency matrix A of an unweighted and undirected graph (where
20  * the value true represents an edge), compute a maximal set of independent vertices and
21  * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
22  * of the set (the iset vector should be uninitialized on input.)
23  */
24 GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
25 {
26     GrB_Index n;
27     GrB_Matrix_nrows(&n,A); // n = # of rows of A
28
29     GrB_Vector prob; // holds random probabilities for each node
30     GrB_Vector neighbor_max; // holds value of max neighbor probability
31     GrB_Vector new_members; // holds set of new members to iset
32     GrB_Vector new_neighbors; // holds set of new neighbors to new iset mbrs.
33     GrB_Vector candidates; // candidate members to iset
34
35     GrB_Vector_new(&prob,GrB_FP32,n);
36     GrB_Vector_new(&neighbor_max,GrB_FP32,n);
37     GrB_Vector_new(&new_members,GrB_BOOL,n);
38     GrB_Vector_new(&new_neighbors,GrB_BOOL,n);
39     GrB_Vector_new(&candidates,GrB_BOOL,n);
40     GrB_Vector_new(iset,GrB_BOOL,n); // Initialize independent set vector, bool
41
42     GrB_UnaryOp set_random;
43     GrB_UnaryOp_new(&set_random,setRandom,GrB_FP32,GrB_UINT32);
44
45     // compute the degree of each vertex.
46     GrB_Vector degrees;
47     GrB_Vector_new(&degrees,GrB_FP64,n);
48     GrB_reduce(degrees,GrB_NULL,GrB_NULL,GrB_PLUS_FP64,A,GrB_NULL);
49
50     // Isolated vertices are not candidates: candidates[degrees != 0] = true
51     GrB_assign(candidates,degrees,GrB_NULL,true,GrB_ALL,n,GrB_NULL);
52
53     // add all singletons to iset: iset[degree == 0] = 1
54     GrB_assign(*iset,degrees,GrB_NULL,true,GrB_ALL,n,GrB_DESC_RC) ;
55
56     // Iterate while there are candidates to check.
57     GrB_Index nvals;
58     GrB_Vector_nvals(&nvals, candidates);
59     while (nvals > 0) {
60         // compute a random probability scaled by inverse of degree
61         GrB_apply(prob,candidates,GrB_NULL,set_random,degrees,GrB_DESC_R);
62     }

```

```

63 // compute the max probability of all neighbors
64 GrB_mnv(neighbor_max, candidates, GrB_NULL, GrB_MAX_SECOND_SEMIRING_FP32, A, prob, GrB_DESC_R);
65
66 // select vertex if its probability is larger than all its active neighbors,
67 // and apply a "masked no-op" to remove stored falses
68 GrB_eWiseAdd(new_members, GrB_NULL, GrB_NULL, GrB_GT_FP64, prob, neighbor_max, GrB_NULL);
69 GrB_apply(new_members, new_members, GrB_NULL, GrB_IDENTITY_BOOL, new_members, GrB_DESC_R);
70
71 // add new members to independent set.
72 GrB_eWiseAdd(*iset, GrB_NULL, GrB_NULL, GrB_LOR, *iset, new_members, GrB_NULL);
73
74 // remove new members from set of candidates  $c = c \ominus !new$ 
75 GrB_eWiseMult(candidates, new_members, GrB_NULL,
76               GrB_LAND, candidates, candidates, GrB_DESC_RC);
77
78 GrB_Vector_nvals(&nvals, candidates);
79 if (nvals == 0) { break; } // early exit condition
80
81 // Neighbors of new members can also be removed from candidates
82 GrB_mnv(new_neighbors, candidates, GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL,
83         A, new_members, GrB_NULL);
84 GrB_eWiseMult(candidates, new_neighbors, GrB_NULL, GrB_LAND,
85               candidates, candidates, GrB_DESC_RC);
86
87 GrB_Vector_nvals(&nvals, candidates);
88 }
89
90 GrB_free(&neighbor_max); // free all objects "new'ed"
91 GrB_free(&new_members);
92 GrB_free(&new_neighbors);
93 GrB_free(&prob);
94 GrB_free(&candidates);
95 GrB_free(&set_random);
96 GrB_free(&degrees);
97
98 return GrB_SUCCESS;
99 }

```

## C.7 Example: Counting triangles in GraphBLAS

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given an  $n \times n$  boolean adjacency matrix,  $A$ , of an undirected graph, computes
9  * the number of triangles in the graph.
10 */
11 uint64_t triangle_count(GrB_Matrix A)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);           //  $n = \#$  of vertices
15
16     //  $L$ :  $N \times N$ , lower-triangular, bool
17     GrB_Matrix L;
18     GrB_Matrix_new(&L, GrB_BOOL, n, n);
19     GrB_select(L, GrB_NULL, GrB_NULL, GrB_TRIL, A, 0UL, GrB_NULL);
20
21     GrB_Matrix C;
22     GrB_Matrix_new(&C, GrB_UINT64, n, n);
23
24     GrB_mxm(C, L, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_UINT64, L, L, GrB_NULL); //  $C \langle L \rangle = L +.* L$ 
25
26     uint64_t count;
27     GrB_reduce(&count, GrB_NULL, GrB_PLUS_MONOID_UINT64, C, GrB_NULL); // 1-norm of  $C$ 
28
29     GrB_free(&C);
30     GrB_free(&L);
31
32     return count;
33 }
```