

The GraphBLAS C API Specification [†]:

Version 2.0.1

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

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

Generated on 2022/11/14 at 07:55:34 EDT

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

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

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

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

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

Contents

25

²⁶ **List of Tables**

²⁷ List of Figures

Acknowledgments

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

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

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

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

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

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

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

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

Chapter 1

Introduction

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

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

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

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

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 ?? 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 ?? when

88 it is not possible to express the signature of that method.

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

94 The remainder of this document is organized as follows:

- 95 • Chapter ??: Basic Concepts
- 96 • Chapter ??: Objects
- 97 • Chapter ??: Methods
- 98 • Chapter ??: Nonpolymorphic interface
- 99 • Appendix ??: Revision history
- 100 • Appendix ??: Non-opaque data format definitions
- 101 • Appendix ??: 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 ?? and (2) user-defined operators created using `GrB_UnaryOp_new()` or `GrB_BinaryOp_new()` (see Section ??).
- *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 ?? and (2) user-defined monoids created using `GrB_Monoid_new()` (see Section ??).
- *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 ?? and ??, and (2) user-defined semirings created using `GrB_Semiring_new()` (see Section ??).
- *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 ??), and user-defined operators created using `GrB_IndexUnaryOp_new` (see Section ??).

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*

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

- 278 • *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as param-
 279 eters to a GraphBLAS method are dimension (or shape) compatible if they have the correct
 280 number of dimensions and sizes for each dimension to satisfy the rules of the mathematical def-
 281 inition of the operation associated with the method. If any *dimension compatibility* rule above
 282 is violated, execution of the GraphBLAS method ends and the GrB_DIMENSION_MISMATCH
 283 error is returned.
- 284 • *domain compatible*: Two domains for which values from one domain can be cast to values in
 285 the other domain as per the rules of the C language. In particular, domains from Table ??
 286 are all compatible with each other, and a domain from a user-defined type is only compatible
 287 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS
 288 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 ??.
<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 ??.
<code>GrB_Object</code>	A method argument type referencing any of the GraphBLAS object types.
<code>GrB_NULL</code>	The GraphBLAS NULL.

2.3 Mathematical foundations

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

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

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

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

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

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

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

Table 2.1: Types of GraphBLAS opaque objects.

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

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

2.4 GraphBLAS opaque objects

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

A GraphBLAS opaque object is accessed through its *handle*. A handle is a variable that references an instance of one of the types from Table ???. 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 ??? 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 ??.

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 ??). 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 ?? 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 ??.

2.5.1 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 ??.) 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 ??(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 ??) and methods that produce nonopaque objects (e.g., `GrB_Matrix_extractTuples()`, Section ??) 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 ??(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 ??). 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	??
GrB_Vector_extractElement	??
GrB_Vector_extractTuples	??
GrB_Matrix_nvals	??
GrB_Matrix_extractElement	??
GrB_Matrix_extractTuples	??
GrB_reduce (vector-scalar value variant)	??
GrB_reduce (matrix-scalar value variant)	??

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 ???. 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 ???(a) are non-negative and include GrB_SUCCESS (a value of 0) and GrB_NO_VALUE.

An API error (listed in Table ???(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 ???(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 GraphBLAS method returns with a GrB_PANIC execution error, no guarantees can be made about

535 the state of any program data.

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

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

547 After a call to any GraphBLAS method that modifies an opaque object, the program can re-
548 trieve additional error information (beyond the error code returned by the method) through a
549 call to the function `GrB_error()`, passing the method's output object as described in Section ??.
550 The function returns a pointer to a NULL-terminated string, and the contents of that string are
551 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error
552 string. `GrB_error()` is a thread-safe function, in the sense that multiple threads can call it simul-
553 taneously and each will get its own error string back, referring to the object passed as an input
554 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 ?? 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 ??) and `GrB_Matrix_extractTuples` (Section ??) 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:

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

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

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

591 3.3 Types (domains)

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

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

The domains for elements that can be stored in collections and operated on through GraphBLAS methods are defined by GraphBLAS objects called GrB_Type. The predefined types and corresponding domains used in the GraphBLAS C API are shown in Table ??.

The Boolean type (bool) is defined in `stdbool.h`, the integral types (int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t) are defined in `stdint.h`, and the floating-point types (float, double) are native to the language and platform and in most cases defined by the IEEE-754 standard.

3.4 Algebraic objects, operators and associated functions

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

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

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

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

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

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

The GraphBLAS *algebraic objects* operators, monoids, and semirings are presented in this section. These objects can be used as input arguments to various GraphBLAS operations, as shown in Table ???. 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 ??.

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 ??.

3.4.1 Operators

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

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

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

(a) Properties of algebraic objects.

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

(b) Recipes for algebraic objects.

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

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

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

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

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

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

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

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

3.4.2 Monoids

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

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

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

3.4.3 Semirings

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

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

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

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

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

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

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

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

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

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

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

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

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

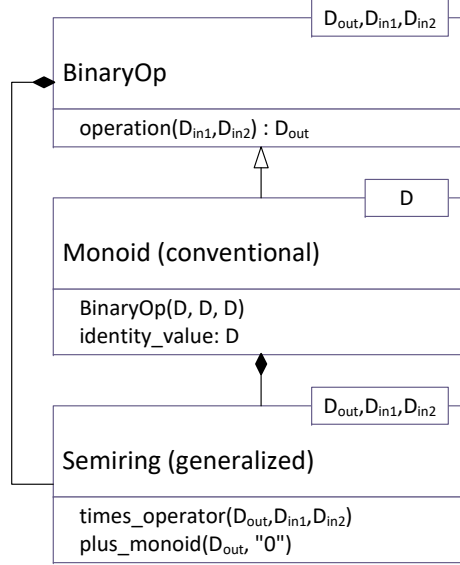


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

3.5 Collections

3.5.1 Scalars

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

3.5.2 Vectors

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

3.5.3 Matrices

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

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

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

3.5.3.1 External matrix formats

The specification also supports the export and import of matrices to/from a number of commonly used formats, such as COO, CSR, and CSC formats. When importing or exporting a matrix to or from a GraphBLAS object using `GrB_Matrix_import (§ ??)` or `GrB_Matrix_export (§ ??)`, 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 ???. A precise definition of the non-opaque data formats can be found in Appendix ??.

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

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

3.5.4 Masks

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

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

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

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

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

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

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

3.6 Fields

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

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

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

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

Table 3.11: Field values of type GrB_Field enumeration, corresponding types, and the objects which must implement that GrB_Field. Collection refers to GrB_Matrix, GrB_Vector, and GrB_Scalar, Algebraic refers to Operators, Monoids, and Semirings, while All refers to all GraphBLAS objects. Global fields are denoted by Global.

(a) Types used with GraphBLAS descriptors.

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

3.7 Descriptors

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

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

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 ??.

3.8 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 ??.

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

(a) Types used with GraphBLAS descriptors.

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

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

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

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

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

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

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

Table 3.14: Enumeration literals and corresponding values returned by GraphBLAS methods and operations.

(a) Informational return values

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

(b) API errors

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

(c) Execution errors

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

Chapter 4

Methods

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

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

4.1 Context methods

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

4.1.1 `init`: Initialize a GraphBLAS context

Creates and initializes a GraphBLAS C API context.

C Syntax

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

Parameters

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

819 **Return Values**

820 `GrB_SUCCESS` operation completed successfully.

821 `GrB_PANIC` unknown internal error.

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

823 **Description**

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

- 826 • `GrB_BLOCKING`: In this mode, each method in a sequence returns after its computations have
827 completed and output arguments are available to subsequent statements in an application.
828 When executing in `GrB_BLOCKING` mode, the methods execute in program order.
- 829 • `GrB_NONBLOCKING`: In this mode, methods in a sequence may return after arguments in
830 the method have been tested for dimension and domain compatibility within the method
831 but potentially before their computations complete. Output arguments are available to sub-
832 sequent GraphBLAS methods in an application. When executing in `GrB_NONBLOCKING`
833 mode, the methods in a sequence may execute in any order that preserves the mathematical
834 result defined by the sequence.

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

837 **4.1.2 finalize: Finalize a GraphBLAS context**

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

839 **C Syntax**

840 `GrB_Info GrB_finalize();`

841 **Return Values**

842 `GrB_SUCCESS` operation completed successfully.

843 `GrB_PANIC` unknown internal error.

Description

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

4.1.3 `getVersion`: Get the version number of the standard.

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

C Syntax

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

Parameters

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

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

Return Values

`GrB_SUCCESS` operation completed successfully.

`GrB_PANIC` unknown internal error.

Description

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

```
#define GRB_VERSION      2  
#define GRB_SUBVERSION  0
```

4.2 Object methods

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

4.2.1 Query methods

The methods in this section query and, depending on the field, set internal fields of many GraphBLAS objects.

4.2.1.1 get: Query the value of an object

C Syntax

```
GrB_Info GrB_<OBJ>_get(GrB_<OBJ> o, GrB_Field field, ...);

GrB_Info GrB_Scalar_get(GrB_Scalar s, GrB_Field field, ...);
GrB_Info GrB_Vector_get(GrB_Vector v, GrB_Field field, ...);
GrB_Info GrB_Matrix_get(GrB_Matrix A, GrB_Field field, ...);

GrB_Info GrB_UnaryOp_get(GrB_UnaryOp op, GrB_Field field, ...);
GrB_Info GrB_IndexUnaryOp_get(GrB_IndexUnaryOp op, GrB_Field field, ...);
GrB_Info GrB_BinaryOp_get(GrB_BinaryOp op, GrB_Field field, ...);
GrB_Info GrB_Monoid_get(GrB_Monoid op, GrB_Field field, ...);
GrB_Info GrB_Semiring_get(GrB_Semiring op, GrB_Field field, ...);

GrB_Info GrB_Descriptor_get(GrB_Descriptor op, GrB_Field field, ...);
GrB_Info GrB_Type_get(GrB_Type op, GrB_Field field, ...);

GrB_Info GrB_Global_get(GrB_Field field, ...);
```

Parameters

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

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

field (IN) The internal field being queried.

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

Return Value

GrB_SUCCESS The method completed successfully.

GrB_PANIC unknown internal error.

GrB_OUT_OF_MEMORY not enough memory available for operation.

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

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

903 Description

904 Queries a field of an existing GraphBLAS object.

905 4.2.1.2 Descriptor_set: Set content of descriptor

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

907 C Syntax

```
908 GrB_Info GrB_Descriptor_set(GrB_Descriptor desc,  
909                             GrB_Desc_Field field,  
910                             GrB_Desc_Value val);
```

911 Parameters

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

913 field (IN) The field being set.

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

915 Return Values

916 GrB_SUCCESS operation completed successfully.

917 GrB_PANIC unknown internal error.

918 GrB_OUT_OF_MEMORY not enough memory available for operation.

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

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

921 Description

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

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

925 GrB_MASK refers to the mask parameter of the operation.

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

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

928 Valid values for the val parameter are:

929 GrB_STRUCTURE Use only the structure of the stored values of the corresponding mask
930 (GrB_MASK) parameter.

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

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

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

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

944 4.2.2 Algebra methods

945 4.2.2.1 Type_new: Construct a new GraphBLAS (user-defined) type

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

948 C Syntax

```
949       GrB_Info GrB_Type_new(GrB_Type   *utype,
950                              size_t       sizeof(ctype));
```

951 Parameters

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

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

Return Values

`GrB_SUCCESS` operation completed successfully.

`GrB_PANIC` unknown internal error.

`GrB_OUT_OF_MEMORY` not enough memory available for operation.

`GrB_NULL_POINTER` `utype` pointer is `NULL`.

Description

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

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

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

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

4.2.2.2 UnaryOp_new: Construct a new GraphBLAS unary operator

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

C Syntax

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

Parameters

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

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

```

984         void func(void *out, const void *in);
985

```

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

989 **d_in** (IN) The GrB_Type of the input argument of the unary operator being created.
990 Should be one of the predefined GraphBLAS types in Table ??, or a user-defined
991 GraphBLAS type.

992 Return Values

993 GrB_SUCCESS operation completed successfully.

994 GrB_PANIC unknown internal error.

995 GrB_OUT_OF_MEMORY not enough memory available for operation.

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

998 GrB_NULL_POINTER unary_op or unary_func pointers are NULL.

999 Description

1000 The UnaryOp_new method creates a new GraphBLAS unary operator

1001 $f_u = \langle \mathbf{D}(d_out), \mathbf{D}(d_in), unary_func \rangle$

1002 and returns a handle to it in unary_op.

1003 The implementation of unary_func must be such that it works even if the d_out and d_in arguments
1004 are aliased. In other words, for all invocations of the function:

```

1005     unary_func(out, in);

```

1006 the value of out must be the same as if the following code was executed:

```

1007     D(d_in) *tmp = malloc(sizeof(D(d_in)));
1008     memcpy(tmp, in, sizeof(D(d_in)));
1009     unary_func(out, tmp);
1010     free(tmp);

```

1011 It is not an error to call this method more than once on the same variable; however, the handle to
1012 the previously created object will be overwritten.

1013 4.2.2.3 BinaryOp_new: Construct a new GraphBLAS binary operator

1014 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types
1015 (domains).

1016 C Syntax

```
1017     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
1018                               void          (*binary_func)(void*,  
1019                                                         const void*,  
1020                                                         const void*),  
1021                               GrB_Type      d_out,  
1022                               GrB_Type      d_in1,  
1023                               GrB_Type      d_in2);
```

1024 Parameters

1025 **binary_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1026 binary operator object.

1027 **binary_func** (IN) A pointer to a user-defined function that takes two input parameters of types
1028 d_in1 and d_in2 and returns a value of type d_out, all passed as void pointers.
1029 Specifically the signature of the function is expected to be of the form:

```
1030         void func(void *out, const void *in1, const void *in2);
```

1032 **d_out** (IN) The GrB_Type of the return value of the binary operator being created. Should
1033 be one of the predefined GraphBLAS types in Table ??, or a user-defined Graph-
1034 BLAS type.

1035 **d_in1** (IN) The GrB_Type of the left hand argument of the binary operator being created.
1036 Should be one of the predefined GraphBLAS types in Table ??, or a user-defined
1037 GraphBLAS type.

1038 **d_in2** (IN) The GrB_Type of the right hand argument of the binary operator being cre-
1039 ated. Should be one of the predefined GraphBLAS types in Table ??, or a user-
1040 defined GraphBLAS type.

1041 Return Values

1042 GrB_SUCCESS operation completed successfully.

1043 GrB_PANIC unknown internal error.

1044 GrB_OUT_OF_MEMORY not enough memory available for operation.

1045 GrB_UNINITIALIZED_OBJECT the GrB_Type (for user-defined types) has not been initialized by a
1046 call to GrB_Type_new.

1047 GrB_NULL_POINTER binary_op or binary_func pointer is NULL.

1048 Description

1049 The BinaryOp_new method creates a new GraphBLAS binary operator

1050 $f_b = \langle \mathbf{D}(d_out), \mathbf{D}(d_in1), \mathbf{D}(d_in2), \text{binary_func} \rangle$

1051 and returns a handle to it in binary_op.

1052 The implementation of binary_func must be such that it works even if any of the d_out, d_in1, and
1053 d_in2 arguments are aliased to each other. In other words, for all invocations of the function:

1054 `binary_func(out, in1, in2);`

1055 the value of out must be the same as if the following code was executed:

```
1056     D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));  
1057     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));  
1058     memcpy(tmp1, in1, sizeof(D(d_in1)));  
1059     memcpy(tmp2, in2, sizeof(D(d_in2)));  
1060     binary_func(out, tmp1, tmp2);  
1061     free(tmp2);  
1062     free(tmp1);
```

1063 It is not an error to call this method more than once on the same variable; however, the handle to
1064 the previously created object will be overwritten.

1065 4.2.2.4 Monoid_new: Construct a new GraphBLAS monoid

1066 Creates a new monoid with specified binary operator and identity value.

1067 C Syntax

```
1068     GrB_Info GrB_Monoid_new(GrB_Monoid *monoid,  
1069                             GrB_BinaryOp binary_op,  
1070                             <type> identity);
```


1071 Parameters

1072 **monoid** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1073 monoid object.

1074 **binary_op** (IN) An existing GraphBLAS associative binary operator whose input and output
1075 types are the same.

1076 **identity** (IN) The value of the identity element of the monoid. Must be the same type as
1077 the type used by the **binary_op** operator.

1078 Return Values

1079 **GrB_SUCCESS** operation completed successfully.

1080 **GrB_PANIC** unknown internal error.

1081 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1082 **GrB_UNINITIALIZED_OBJECT** the **GrB_BinaryOp** (for user-defined operators) has not been initial-
1083 ized by a call to **GrB_BinaryOp_new**.

1084 **GrB_NULL_POINTER** monoid pointer is NULL.

1085 **GrB_DOMAIN_MISMATCH** all three argument types of the binary operator and the type of the
1086 identity value are not the same.

1087 Description

1088 The **Monoid_new** method creates a new monoid $M = \langle \mathbf{D}(\text{binary_op}), \text{binary_op}, \text{identity} \rangle$ and re-
1089 turns a handle to it in **monoid**.

1090 If **binary_op** is not associative, the results of GraphBLAS operations that require associativity of
1091 this monoid will be undefined.

1092 It is not an error to call this method more than once on the same variable; however, the handle to
1093 the previously created object will be overwritten.

1094 4.2.2.5 Semiring_new: Construct a new GraphBLAS semiring

1095 Creates a new semiring with specified domain, operators, and elements.

1096 C Syntax

```
1097        GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1098                                    GrB_Monoid     add_op,  
1099                                    GrB_BinaryOp   mul_op);
```

1100 Parameters

- 1101 **semiring** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1102 semiring.
- 1103 **add_op** (IN) An existing GraphBLAS commutative monoid that specifies the addition op-
1104 erator and its identity.
- 1105 **mul_op** (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-
1106 plication operator. In addition, **mul_op**'s output domain, $\mathbf{D}_{out}(\text{mul_op})$, must be
1107 the same as the **add_op**'s domain $\mathbf{D}(\text{add_op})$.

1108 Return Values

- 1109 **GrB_SUCCESS** operation completed successfully.
- 1110 **GrB_PANIC** unknown internal error.
- 1111 **GrB_OUT_OF_MEMORY** not enough memory available for this method to complete.
- 1112 **GrB_UNINITIALIZED_OBJECT** the **add_op** (for user-define monoids) object has not been initialized
1113 with a call to **GrB_Monoid_new** or the **mul_op** (for user-defined
1114 operators) object has not been not been initialized by a call to
1115 **GrB_BinaryOp_new**.
- 1116 **GrB_NULL_POINTER** semiring pointer is NULL.
- 1117 **GrB_DOMAIN_MISMATCH** the output domain of **mul_op** does not match the domain of the
1118 **add_op** monoid.

1119 Description

1120 The **Semiring_new** method creates a new semiring:

$$1121 \quad S = \langle \mathbf{D}_{out}(\text{mul_op}), \mathbf{D}_{in_1}(\text{mul_op}), \mathbf{D}_{in_2}(\text{mul_op}), \text{add_op}, \text{mul_op}, \mathbf{0}(\text{add_op}) \rangle$$

1122 and returns a handle to it in **semiring**. Note that $\mathbf{D}_{out}(\text{mul_op})$ must be the same as $\mathbf{D}(\text{add_op})$.

1123 If **add_op** is not commutative, then GraphBLAS operations using this semiring will be undefined.

1124 It is not an error to call this method more than once on the same variable; however, the handle to
1125 the previously created object will be overwritten.

1126 4.2.2.6 IndexUnaryOp_new: Construct a new GraphBLAS index unary operator [Scott: 1127 NEW CONTENT]

1128 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its
1129 types (domains).

1130 C Syntax

```
1131     GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp  *index_unary_op,  
1132                                   void (*index_unary_func)(void*,  
1133                                                             const void*,  
1134                                                             GrB_Index,  
1135                                                             GrB_Index,  
1136                                                             const void*),  
1137                                   GrB_Type          d_out,  
1138                                   GrB_Type          d_in1,  
1139                                   GrB_Type          d_in2);
```

1140 Parameters

1141 **index_unary_op** (INOUT) On successful return, contains a handle to the newly created Graph-
1142 BLAS index unary operator object.

1143 **index_unary_func** (IN) A pointer to a user-defined function that takes input parameters of types
1144 d_in1, GrB_Index, GrB_Index and d_in2 and returns a value of type d_out. Ex-
1145 cept for the GrB_Index parameters, all are passed as void pointers. Specifically
1146 the signature of the function is expected to be of the form:

```
1147         void func(void      *out,  
1148                   const void *in1,  
1149                   GrB_Index  row_index,  
1150                   GrB_Index  col_index,  
1151                   const void *in2);  
1152
```

1153 **d_out** (IN) The GrB_Type of the return value of the index unary operator being created.
1154 Should be one of the predefined GraphBLAS types in Table ??, or a user-defined
1155 GraphBLAS type.

1156 **d_in1** (IN) The GrB_Type of the first input argument of the index unary operator being
1157 created and corresponds to the stored values of the GrB_Vector or GrB_Matrix be-
1158 ing operated on. Should be one of the predefined GraphBLAS types in Table ??,
1159 or a user-defined GraphBLAS type.

1160 **d_in2** (IN) The GrB_Type of the last input argument of the index unary operator be-
1161 ing created and corresponds to a scalar provided by the GraphBLAS operation
1162 that uses this operator. Should be one of the predefined GraphBLAS types in
1163 Table ??, or a user-defined GraphBLAS type.

1164 Return Values

1165 GrB_SUCCESS operation completed successfully.

1194 C Syntax

```
1195         GrB_Info GrB_Scalar_new(GrB_Scalar *s,  
1196                                 GrB_Type    d);
```

1197 Parameters

1198 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1199 scalar.

1200 **d** (IN) The type corresponding to the domain of the scalar being created. Can be
1201 one of the predefined GraphBLAS types in Table ??, or an existing user-defined
1202 GraphBLAS type.

1203 Return Values

1204 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1205 blocking mode, this indicates that the API checks for the input
1206 arguments passed successfully. Either way, output scalar **s** is ready
1207 to be used in the next method of the sequence.

1208 **GrB_PANIC** Unknown internal error.

1209 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1210 GraphBLAS objects (input or output) is in an invalid state caused
1211 by a previous execution error. Call **GrB_error()** to access any error
1212 messages generated by the implementation.

1213 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1214 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
1215 (needed for user-defined types).

1216 **GrB_NULL_POINTER** The **s** pointer is NULL.

1217 Description

1218 Creates a new GraphBLAS scalar **s** of domain **D(d)** and empty **L(s)**. The method returns a handle
1219 to the new scalar in **s**.

1220 It is not an error to call this method more than once on the same variable; however, the handle to
1221 the previously created object will be overwritten.

1222 4.2.3.2 Scalar_dup: Construct a copy of a GraphBLAS scalar

1223 Creates a new scalar with the same domain and contents as another scalar.

1224 C Syntax

```
1225         GrB_Info GrB_Scalar_dup(GrB_Scalar      *t,  
1226                                 const GrB_Scalar  s);
```

1227 Parameters

1228 **t** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1229 scalar.

1230 **s** (IN) The GraphBLAS scalar to be duplicated.

1231 Return Values

1232 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1233 blocking mode, this indicates that the API checks for the input
1234 arguments passed successfully. Either way, output scalar **t** is ready
1235 to be used in the next method of the sequence.

1236 **GrB_PANIC** Unknown internal error.

1237 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1238 GraphBLAS objects (input or output) is in an invalid state caused
1239 by a previous execution error. Call **GrB_error()** to access any error
1240 messages generated by the implementation.

1241 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1242 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to
1243 **Scalar_new** or **Scalar_dup**.

1244 **GrB_NULL_POINTER** The **t** pointer is NULL.

1245 Description

1246 Creates a new scalar *t* of domain **D(s)** and contents **L(s)**. The method returns a handle to the new
1247 scalar in **t**.

1248 It is not an error to call this method more than once with the same output variable; however, the
1249 handle to the previously created object will be overwritten.

1250 4.2.3.3 Scalar_clear: Clear/remove a stored value from a scalar

1251 Removes the stored value from a scalar.

1252 C Syntax

1253 GrB_Info GrB_Scalar_clear(GrB_Scalar s);

1254 Parameters

1255 s (INOUT) An existing GraphBLAS scalar to clear.

1256 Return Values

1257 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1258 blocking mode, this indicates that the API checks for the input
1259 arguments passed successfully. Either way, output scalar **s** is ready
1260 to be used in the next method of the sequence.

1261 GrB_PANIC Unknown internal error.

1262 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1263 GraphBLAS objects (input or output) is in an invalid state caused
1264 by a previous execution error. Call `GrB_error()` to access any error
1265 messages generated by the implementation.

1266 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1267 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, **s**, has not been initialized by a call to
1268 Scalar_new or Scalar_dup.

1269 Description

1270 Removes the stored value from an existing scalar. After the call, **L(s)** is empty. The size of the
1271 scalar does not change.

1272 4.2.3.4 Scalar_nvals: Number of stored elements in a scalar

1273 Retrieve the number of stored elements in a scalar (either zero or one).

1274 C Syntax

1275 GrB_Info GrB_Scalar_nvals(GrB_Index *nvals,
1276 const GrB_Scalar s);

1277

1278

1279

1280

1281

1282

1283

1285

1286

1287

1288

1280

1293

1294

1295

1296

1297

1298

1299

1300

1301

1302

1303

1304 Return Values

1305 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1306 blocking mode, this indicates that the compatibility tests on in-
1307 dex/dimensions and domains for the input arguments passed suc-
1308 cessfully. Either way, the output scalar `s` is ready to be used in the
1309 next method of the sequence.

1310 GrB_PANIC Unknown internal error.

1311 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1312 GraphBLAS objects (input or output) is in an invalid state caused
1313 by a previous execution error. Call `GrB_error()` to access any error
1314 messages generated by the implementation.

1315 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1316 GrB_UNINITIALIZED_OBJECT The GraphBLAS scalar, `s`, has not been initialized by a call to
1317 `Scalar_new` or `Scalar_dup`.

1318 GrB_DOMAIN_MISMATCH The domains of `s` and `val` are incompatible.

1319 Description

1320 First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: `D(val)` must
1321 be compatible with `D(s)`. Two domains are compatible with each other if values from one domain
1322 can be cast to values in the other domain as per the rules of the C language. In particular, domains
1323 from Table ?? are all compatible with each other. A domain from a user-defined type is only com-
1324 patible with itself. If any compatibility rule above is violated, execution of `GrB_Scalar_setElement`
1325 ends and the domain mismatch error listed above is returned.

1326 We are now ready to carry out the assignment `val`; that is:

$$1327 \qquad s(0) = \text{val}$$

1328 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1329 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1330 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1331 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be
1332 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1333 4.2.3.6 Scalar_extractElement: Extract a single element from a scalar.

1334 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

1335 C Syntax

```
1336      GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1337                                         const GrB_Scalar s);
```

1338 Parameters

1339 **val** (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain
1340 of scalar **s**. On successful return, **val** holds the result of the operation, and any
1341 previous value in **val** is overwritten.

1342 **s** (IN) The GraphBLAS scalar from which an element is extracted.

1343 Return Values

1344 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1345 cessfully. This indicates that the compatibility tests on dimensions
1346 and domains for the input arguments passed successfully, and the
1347 output scalar, **val**, has been computed and is ready to be used in
1348 the next method of the sequence.

1349 **GrB_PANIC** Unknown internal error.

1350 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1351 GraphBLAS objects (input or output) is in an invalid state caused
1352 by a previous execution error. Call **GrB_error()** to access any error
1353 messages generated by the implementation.

1354 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1355 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to
1356 **Scalar_new** or **Scalar_dup**.

1357 **GrB_NULL_POINTER** **val** pointer is NULL.

1358 **GrB_DOMAIN_MISMATCH** The domains of the scalar or scalar are incompatible.

1359 **GrB_NO_VALUE** There is no stored value in the scalar.

1360 Description

1361 First, **val** and input GraphBLAS scalar are tested for domain compatibility as follows: **D(val)** must
1362 be compatible with **D(s)**. Two domains are compatible with each other if values from one domain
1363 can be cast to values in the other domain as per the rules of the C language. In particular, domains
1364 from Table ?? are all compatible with each other. A domain from a user-defined type is only compat-
1365 ible with itself. If any compatibility rule above is violated, execution of **GrB_Scalar_extractElement**
1366 ends and the domain mismatch error listed above is returned.

1367 Then, if no value is currently stored in the GraphBLAS scalar, the method returns `GrB_NO_VALUE`
1368 and `val` remains unchanged.

1369 Finally the extract into the output argument, `val` can be performed; that is:

1370
$$\text{val} = \text{s}(0)$$

1371 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1372 `GrB_SUCCESS`, the new contents of `val` are as defined above.

1373 4.2.4 Vector methods

1374 4.2.4.1 Vector_new: Construct new vector

1375 Creates a new vector with specified domain and size.

1376 C Syntax

```
1377 GrB_Info GrB_Vector_new(GrB_Vector *v,  
1378                          GrB_Type    d,  
1379                          GrB_Index   nsize);
```

1380 Parameters

1381 `v` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1382 vector.

1383 `d` (IN) The type corresponding to the domain of the vector being created. Can be
1384 one of the predefined GraphBLAS types in Table ??, or an existing user-defined
1385 GraphBLAS type.

1386 `nsize` (IN) The size of the vector being created.

1387 Return Values

1388 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1389 blocking mode, this indicates that the API checks for the input
1390 arguments passed successfully. Either way, output vector `v` is ready
1391 to be used in the next method of the sequence.

1392 `GrB_PANIC` Unknown internal error.

1393 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1394 GraphBLAS objects (input or output) is in an invalid state caused
1395 by a previous execution error. Call `GrB_error()` to access any error
1396 messages generated by the implementation.

1426 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1427 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, **u**, has not been initialized by a call to
1428 Vector_new or Vector_dup.

1429 GrB_NULL_POINTER The **w** pointer is NULL.

1430 Description

1431 Creates a new vector **w** of domain **D(u)**, size **size(u)**, and contents **L(u)**. The method returns a
1432 handle to the new vector in **w**.

1433 It is not an error to call this method more than once on the same variable; however, the handle to
1434 the previously created object will be overwritten.

1435 4.2.4.3 Vector_resize: Resize a vector

1436 Changes the size of an existing vector.

1437 C Syntax

```
1438       GrB_Info GrB_Vector_resize(GrB_Vector w,  
1439                                   GrB_Index nsize);
```

1440 Parameters

1441 **w** (INOUT) An existing Vector object that is being resized.

1442 **nsize** (IN) The new size of the vector. It can be smaller or larger than the current size.

1443 Return Values

1444 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1445 blocking mode, this indicates that the API checks for the input
1446 arguments passed successfully. Either way, output vector **w** is ready
1447 to be used in the next method of the sequence.

1448 GrB_PANIC Unknown internal error.

1449 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1450 GraphBLAS objects (input or output) is in an invalid state caused
1451 by a previous execution error. Call GrB_error() to access any error
1452 messages generated by the implementation.

1453 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1454 GrB_NULL_POINTER The w pointer is NULL.

1455 GrB_INVALID_VALUE nsz is zero or outside the range of the type GrB_Index.

1456 Description

1457 Changes the size of w to nsz . The domain $\mathbf{D}(w)$ of vector w remains the same. The contents $\mathbf{L}(w)$
1458 are modified as described below.

1459 Let $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$ when the method is called. When the method returns, $w = \langle \mathbf{D}(w), nsz, \mathbf{L}'(w) \rangle$
1460 where $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < nsz)\}$. That is, all elements of w with index greater
1461 than or equal to the new vector size (nsz) are dropped.

1462 4.2.4.4 Vector_clear: Clear a vector

1463 Removes all the elements (tuples) from a vector.

1464 C Syntax

1465 GrB_Info GrB_Vector_clear(GrB_Vector v);

1466 Parameters

1467 v (INOUT) An existing GraphBLAS vector to clear.

1468 Return Values

1469 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1470 blocking mode, this indicates that the API checks for the input
1471 arguments passed successfully. Either way, output vector v is ready
1472 to be used in the next method of the sequence.

1473 GrB_PANIC Unknown internal error.

1474 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1475 GraphBLAS objects (input or output) is in an invalid state caused
1476 by a previous execution error. Call GrB_error() to access any error
1477 messages generated by the implementation.

1478 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1479 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v , has not been initialized by a call to
1480 Vector_new or Vector_dup.

1481 Description

1482 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,
1483 $L(v) = \emptyset$. The size of the vector does not change.

1484 4.2.4.5 Vector_size: Size of a vector

1485 Retrieve the size of a vector.

1486 C Syntax

```
1487         GrB_Info GrB_Vector_size(GrB_Index      *nsize,  
1488                                const GrB_Vector v);
```

1489 Parameters

1490 nsize (OUT) On successful return, is set to the size of the vector.

1491 v (IN) An existing GraphBLAS vector being queried.

1492 Return Values

1493 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1494 cessfully and the value of `nsize` has been set.

1495 GrB_PANIC Unknown internal error.

1496 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1497 GraphBLAS objects (input or output) is in an invalid state caused
1498 by a previous execution error. Call `GrB_error()` to access any error
1499 messages generated by the implementation.

1500 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to
1501 `Vector_new` or `Vector_dup`.

1502 GrB_NULL_POINTER `nsize` pointer is NULL.

1503 Description

1504 Return `size(v)` in `nsize`.

1505 4.2.4.6 Vector_nvals: Number of stored elements in a vector

1506 Retrieve the number of stored elements (tuples) in a vector.

1507 C Syntax

```
1508      GrB_Info GrB_Vector_nvals(GrB_Index      *nvals,  
1509                               const GrB_Vector v);
```

1510 Parameters

1511 **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)
1512 in the vector.

1513 **v** (IN) An existing GraphBLAS vector being queried.

1514 Return Values

1515 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1516 cessfully and the value of **nvals** has been set.

1517 **GrB_PANIC** Unknown internal error.

1518 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1519 GraphBLAS objects (input or output) is in an invalid state caused
1520 by a previous execution error. Call **GrB_error()** to access any error
1521 messages generated by the implementation.

1522 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1523 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to
1524 **Vector_new** or **Vector_dup**.

1525 **GrB_NULL_POINTER** The **nvals** pointer is **NULL**.

1526 Description

1527 Return **nvals(v)** in **nvals**. This is the number of stored elements in vector **v**, which is the size of
1528 **L(v)** (see Section ??).

1529 4.2.4.7 Vector_build: Store elements from tuples into a vector

1530 C Syntax

```
1531      GrB_Info GrB_Vector_build(GrB_Vector      w,  
1532                               const GrB_Index  *indices,  
1533                               const <type>     *values,  
1534                               GrB_Index        n,  
1535                               const GrB_BinaryOp dup);
```


1536 Parameters

- 1537 **w** (INOUT) An existing Vector object to store the result.
- 1538 **indices** (IN) Pointer to an array of indices.
- 1539 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of
1540 vector **w**.
- 1541 **n** (IN) The number of entries contained in each array (the same for **indices** and **values**).
- 1542 **dup** (IN) An associative and commutative binary operator to apply when duplicate
1543 values for the same location are present in the input arrays. All three domains of
1544 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,
1545 then duplicate locations will result in an error.

1546 Return Values

- 1547 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1548 blocking mode, this indicates that the API checks for the input
1549 arguments passed successfully. Either way, output vector **w** is
1550 ready to be used in the next method of the sequence.
- 1551 **GrB_PANIC** Unknown internal error.
- 1552 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
1553 opaque GraphBLAS objects (input or output) is in an invalid
1554 state caused by a previous execution error. Call **GrB_error()** to
1555 access any error messages generated by the implementation.
- 1556 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
- 1557 **GrB_UNINITIALIZED_OBJECT** Either **w** has not been initialized by a call to **GrB_Vector_new**
1558 or by **GrB_Vector_dup**, or **dup** has not been initialized by a call
1559 to **GrB_BinaryOp_new**.
- 1560 **GrB_NULL_POINTER** **indices** or **values** pointer is **NULL**.
- 1561 **GrB_INDEX_OUT_OF_BOUNDS** A value in **indices** is outside the allowed range for **w**.
- 1562 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are
1563 not all the same, or the domains of **values** and **w** are incompatible
1564 with each other or D_{dup} .
- 1565 **GrB_OUTPUT_NOT_EMPTY** Output vector **w** already contains valid tuples (elements). In
1566 other words, **GrB_Vector_nvals(C)** returns a positive value.
- 1567 **GrB_INVALID_VALUE** **indices** contains a duplicate location and **dup** is **GrB_NULL**.

Description

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

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

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

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

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

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

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

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

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

4.2.4.8 Vector_setElement: Set a single element in a vector

Set one element of a vector to a given value.

C Syntax

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

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

Parameters

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

1597 **val** or **s** (IN) Scalar assign. Its domain (type) must be compatible with the domain of **w**.
 1598 **index** (IN) The location of the element to be assigned.

1599 **Return Values**

1600 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 1601 blocking mode, this indicates that the compatibility tests on in-
 1602 dex/dimensions and domains for the input arguments passed suc-
 1603 cessfully. Either way, the output vector **w** is ready to be used in
 1604 the next method of the sequence.

1605 **GrB_PANIC** Unknown internal error.

1606 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 1607 GraphBLAS objects (input or output) is in an invalid state caused
 1608 by a previous execution error. Call **GrB_error()** to access any error
 1609 messages generated by the implementation.

1610 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1611 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been
 1612 initialized by a call to a respective constructor.

1613 **GrB_INVALID_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1614 **GrB_DOMAIN_MISMATCH** The domains of the vector and the scalar are incompatible.

1615 **Description**

1616 First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)**
 1617 must be compatible with **D(w)**. Two domains are compatible with each other if values from
 1618 one domain can be cast to values in the other domain as per the rules of the C language. In
 1619 particular, domains from Table ?? are all compatible with each other. A domain from a user-
 1620 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
 1621 **GrB_Vector_setElement** ends and the domain mismatch error listed above is returned.

1622 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1623 \qquad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1624 If this condition is violated, execution of **GrB_Vector_setElement** ends and the invalid index error
 1625 listed above is returned.

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

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

1626 In the case of a transparent scalar or if $\mathbf{L}(\mathbf{s})$ is not empty, then a value will be stored at the
 1627 specified location in \mathbf{w} , overwriting any value that may have been stored there before. In the case
 1628 of a GraphBLAS scalar, if $\mathbf{L}(\mathbf{s})$ is empty, then any value stored at the specified location in \mathbf{w} will
 1629 be removed.

1630 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
 1631 of \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
 1632 return value GrB_SUCCESS and the new contents of vector \mathbf{w} is as defined above but may not be
 1633 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1634 4.2.4.9 Vector_removeElement: Remove an element from a vector

1635 Remove (annihilate) one stored element from a vector.

1636 C Syntax

```
1637         GrB_Info GrB_Vector_removeElement(GrB_Vector  w,
1638                                         GrB_Index   index);
```

1639 Parameters

1640 \mathbf{w} (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1641 \mathbf{index} (IN) The location of the element to be removed.

1642 Return Values

1643 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 1644 blocking mode, this indicates that the compatibility tests on in-
 1645 dex/dimensions and domains for the input arguments passed suc-
 1646 cessfully. Either way, the output vector \mathbf{w} is ready to be used in
 1647 the next method of the sequence.

1648 GrB_PANIC Unknown internal error.

1649 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 1650 GraphBLAS objects (input or output) is in an invalid state caused
 1651 by a previous execution error. Call GrB_error() to access any error
 1652 messages generated by the implementation.

1653 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1654 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, \mathbf{w} , has not been initialized by a call to
 1655 Vector_new or Vector_dup.

1656 GrB_INVALID_INDEX \mathbf{index} specifies a location that is outside the dimensions of \mathbf{w} .

1657 Description

1658 First, the `index` parameter is checked for a valid value where the following condition must hold:

$$1659 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1660 If this condition is violated, execution of `GrB_Vector_removeElement` ends and the invalid index
1661 error listed above is returned.

1662 We are now ready to carry out the removal of a value that may be stored at the location specified
1663 by `index`. If a value does not exist at the specified location in \mathbf{w} , no error is reported and the
1664 operation has no effect on the state of \mathbf{w} . In either case, the following will be true on return from
1665 the method: `index` \notin `ind(\mathbf{w})`.

1666 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1667 of \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1668 return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not be
1669 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1670 4.2.4.10 Vector_extractElement: Extract a single element from a vector.

1671 Extract one element of a vector into a scalar.

1672 C Syntax

```
1673 // scalar value
1674 GrB_Info GrB_Vector_extractElement(<type>          *val,
1675                                     const GrB_Vector u,
1676                                     GrB_Index        index);
1677
1678 // GraphBLAS scalar
1679 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1680                                     const GrB_Vector u,
1681                                     GrB_Index        index);
```

1682 Parameters

1683 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector
1684 `u`. On successful return, this scalar holds the result of the extract. Any previous
1685 value stored in `val` or `s` is overwritten.

1686 `u` (IN) The GraphBLAS vector from which an element is extracted.

1687 `index` (IN) The location in `u` to extract.

1688 Return Values

- 1689 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1690 cessfully. This indicates that the compatibility tests on dimensions
1691 and domains for the input arguments passed successfully, and the
1692 output scalar, **val** or **s**, has been computed and is ready to be used
1693 in the next method of the sequence.
- 1694 **GrB_NO_VALUE** When using the transparent scalar, **val**, this is returned when there
1695 is no stored value at specified location.
- 1696 **GrB_PANIC** Unknown internal error.
- 1697 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1698 GraphBLAS objects (input or output) is in an invalid state caused
1699 by a previous execution error. Call **GrB_error()** to access any error
1700 messages generated by the implementation.
- 1701 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
- 1702 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **u**, or scalar, **s**, has not been initialized by
1703 a call to a corresponding constructor.
- 1704 **GrB_NULL_POINTER** **val** pointer is **NULL**.
- 1705 **GrB_INVALID_INDEX** **index** specifies a location that is outside the dimensions of **w**.
- 1706 **GrB_DOMAIN_MISMATCH** The domains of the vector and scalar are incompatible.

1707 Description

1708 First, the scalar and input vector are tested for domain compatibility as follows: **D(val)** or **D(s)** must
1709 be compatible with **D(u)**. Two domains are compatible with each other if values from one domain
1710 can be cast to values in the other domain as per the rules of the C language. In particular, domains
1711 from Table ?? are all compatible with each other. A domain from a user-defined type is only compat-
1712 ible with itself. If any compatibility rule above is violated, execution of **GrB_Vector_extractElement**
1713 ends and the domain mismatch error listed above is returned.

1714 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1715 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1716 If this condition is violated, execution of **GrB_Vector_extractElement** ends and the invalid index
1717 error listed above is returned.

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

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

1718 If $\text{index} \in \mathbf{ind}(u)$, then the corresponding value from u is copied into s or val with casting as
 1719 necessary. If $\text{index} \notin \mathbf{ind}(u)$, then one of the follow occurs depending on output scalar type:

- 1720 • The GraphBLAS scalar, s , is cleared and `GrB_SUCCESS` is returned.
- 1721 • The non-opaque scalar, val , is unchanged, and `GrB_NO_VALUE` is returned.

1722 When using the non-opaque scalar variant (val) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
 1723 mode, the new contents of val are as defined above if the method exits with return value `GrB_SUCCESS`
 1724 or `GrB_NO_VALUE`.

1725 When using the GraphBLAS scalar variant (s) with a `GrB_SUCCESS` return value, the method
 1726 exits and the new contents of s is as defined above and fully computed in `GrB_BLOCKING` mode.
 1727 In `GrB_NONBLOCKING` mode, the new contents of s is as defined above but may not be fully
 1728 computed; however, it can be used in the next GraphBLAS method call in a sequence.

1729 4.2.4.11 Vector_extractTuples: Extract tuples from a vector

1730 Extract the contents of a GraphBLAS vector into non-opaque data structures.

1731 C Syntax

```
1732      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,
1733                                     <type>          *values,
1734                                     GrB_Index        *n,
1735                                     const GrB_Vector  v);
1736
```

1737 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored
 1738 values' indices.

1739 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
 1740 the stored values whose type is compatible with $\mathbf{D}(v)$.

1741 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the
 1742 values and indices arrays can hold. Upon return, it will contain the number of
 1743 values written to the arrays.

1744 **v** (IN) An existing GraphBLAS vector.

1745 Return Values

1746 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
 1747 cessfully. This indicates that the compatibility tests on the input
 1748 argument passed successfully, and the output arrays, **indices** and
 1749 **values**, have been computed.

1750 GrB_PANIC Unknown internal error.

1751 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1752 GraphBLAS objects (input or output) is in an invalid state caused
1753 by a previous execution error. Call GrB_error() to access any error
1754 messages generated by the implementation.

1755 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1756 GrB_INSUFFICIENT_SPACE Not enough space in `indices` and `values` (as indicated by the `n` pa-
1757 rameter) to hold all of the tuples that will be extracted.

1758 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to
1759 Vector_new or Vector_dup.

1760 GrB_NULL_POINTER `indices`, `values`, or `n` pointer is NULL.

1761 GrB_DOMAIN_MISMATCH The domains of the `v` vector or `values` array are incompatible with
1762 one another.

1763 Description

1764 This method will extract all the tuples from the GraphBLAS vector `v`. The values associated
1765 with those tuples are placed in the `values` array and the indices are placed in the `indices` array.
1766 Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least
1767 GrB_Vector_nvals(`v`) elements before calling this function.

1768 Upon return of this function, `n` will be set to the number of values (and indices) copied. Also, the
1769 entries of `indices` are unique, but not necessarily sorted. Each tuple (i, v_i) in `v` is unzipped and
1770 copied into a distinct k th location in output vectors:

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

1771 where $0 \leq k < \text{GrB_Vector_nvals}(v)$. No gaps in output vectors are allowed; that is, if `indices`[k]
1772 and `values`[k] exist upon return, so does `indices`[j] and `values`[j] for all j such that $0 \leq j < k$.

1773 Note that if the value in `n` on input is less than the number of values contained in the vector `v`,
1774 then a GrB_INSUFFICIENT_SPACE error is returned because it is undefined which subset of values
1775 would be extracted otherwise.

1776 In both GrB_BLOCKING mode GrB_NONBLOCKING mode if the method exits with return value
1777 GrB_SUCCESS, the new contents of the arrays `indices` and `values` are as defined above.

1778 4.2.5 Matrix methods

1779 4.2.5.1 Matrix_new: Construct new matrix

1780 Creates a new matrix with specified domain and dimensions.

1781 C Syntax

```
1782         GrB_Info GrB_Matrix_new(GrB_Matrix *A,  
1783                                 GrB_Type      d,  
1784                                 GrB_Index     nrows,  
1785                                 GrB_Index     ncols);
```

1786 Parameters

1787 **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1788 matrix.

1789 **d** (IN) The type corresponding to the domain of the matrix being created. Can be
1790 one of the predefined GraphBLAS types in Table ??, or an existing user-defined
1791 GraphBLAS type.

1792 **nrows** (IN) The number of rows of the matrix being created.

1793 **ncols** (IN) The number of columns of the matrix being created.

1794 Return Values

1795 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1796 blocking mode, this indicates that the API checks for the input ar-
1797 guments passed successfully. Either way, output matrix **A** is ready
1798 to be used in the next method of the sequence.

1799 **GrB_PANIC** Unknown internal error.

1800 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1801 GraphBLAS objects (input or output) is in an invalid state caused
1802 by a previous execution error. Call **GrB_error()** to access any error
1803 messages generated by the implementation.

1804 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1805 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
1806 (needed for user-defined types).

1807 **GrB_NULL_POINTER** The **A** pointer is **NULL**.

1808 **GrB_INVALID_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB_Index**.

1809 Description

1810 Creates a new matrix **A** of domain **D**(**d**), size **nrows** \times **ncols**, and empty **L**(**A**). The method returns
1811 a handle to the new matrix in **A**.

1812 It is not an error to call this method more than once on the same variable; however, the handle to
1813 the previously created object will be overwritten.

1814 4.2.5.2 Matrix_dup: Construct a copy of a GraphBLAS matrix

1815 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

1816 C Syntax

```
1817         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
1818                                const GrB_Matrix A);
```

1819 Parameters

1820 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1821 matrix.

1822 A (IN) The GraphBLAS matrix to be duplicated.

1823 Return Values

1824 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1825 blocking mode, this indicates that the API checks for the input
1826 arguments passed successfully. Either way, output matrix C is ready
1827 to be used in the next method of the sequence.

1828 GrB_PANIC Unknown internal error.

1829 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1830 GraphBLAS objects (input or output) is in an invalid state caused
1831 by a previous execution error. Call GrB_error() to access any error
1832 messages generated by the implementation.

1833 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1834 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
1835 any matrix constructor.

1836 GrB_NULL_POINTER The C pointer is NULL.

1837 Description

1838 Creates a new matrix C of domain D(A), size nrows(A) × ncols(A), and contents L(A). It returns
1839 a handle to it in C.

1840 It is not an error to call this method more than once on the same variable; however, the handle to
1841 the previously created object will be overwritten.

1842 4.2.5.3 Matrix_diag: Construct a diagonal GraphBLAS matrix

1843 Creates a new matrix with the same domain and contents as a GrB_Vector, and square dimensions
1844 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

1845 C Syntax

```
1846         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
1847                                 const GrB_Vector  v,  
1848                                 int64_t           k);
```

1849 Parameters

1850 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1851 matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

1852 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the
1853 matrix.

1854 k (IN) The diagonal to which the vector is assigned. k = 0 represents the main
1855 diagonal, k > 0 is above the main diagonal, and k < 0 is below.

1856 Return Values

1857 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1858 blocking mode, this indicates that the API checks for the input
1859 arguments passed successfully. Either way, output matrix C is ready
1860 to be used in the next method of the sequence.

1861 GrB_PANIC Unknown internal error.

1862 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1863 GraphBLAS objects (input or output) is in an invalid state caused
1864 by a previous execution error. Call GrB_error() to access any error
1865 messages generated by the implementation.

1866 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

1867 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to
1868 Vector_new or Vector_dup.

1869 GrB_NULL_POINTER The C pointer is NULL.

1870 Description

1871 Creates a new matrix **C** of domain **D(v)**, size $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$, and contents

$$\begin{aligned} 1872 \quad \mathbf{L}(\mathbf{C}) &= \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or} \\ 1873 \quad \mathbf{L}(\mathbf{C}) &= \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0. \end{aligned}$$

1874 It returns a handle to it in **C**. It is not an error to call this method more than once on the same
1875 variable; however, the handle to the previously created object will be overwritten.

1876 4.2.5.4 Matrix_resize: Resize a matrix

1877 Changes the dimensions of an existing matrix.

1878 C Syntax

```
1879      GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
1880                               GrB_Index  nrows,  
1881                               GrB_Index  ncols);
```

1882 Parameters

1883 **C** (INOUT) An existing Matrix object that is being resized.

1884 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the
1885 current number of rows.

1886 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than
1887 the current number of columns.

1888 Return Values

1889 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1890 blocking mode, this indicates that the API checks for the input
1891 arguments passed successfully. Either way, output matrix **C** is ready
1892 to be used in the next method of the sequence.

1893 **GrB_PANIC** Unknown internal error.

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

1898 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1899 GrB_NULL_POINTER The C pointer is NULL.

1900 GrB_INVALID_VALUE nrows or ncols is zero or outside the range of the type GrB_Index.

1901 **Description**

1902 Changes the number of rows and columns of C to nrows and ncols, respectively. The domain $\mathbf{D}(\mathbf{C})$
1903 of matrix C remains the same. The contents $\mathbf{L}(\mathbf{C})$ are modified as described below.

1904 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
1905 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
1906 greater than or equal to ncols are dropped.
1907

1908 **4.2.5.5 Matrix_clear: Clear a matrix**

1909 Removes all elements (tuples) from a matrix.

1910 **C Syntax**

1911 GrB_Info GrB_Matrix_clear(GrB_Matrix A);

1912 **Parameters**

1913 A (IN) An existing GraphBLAS matrix to clear.

1914 **Return Values**

1915 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1916 blocking mode, this indicates that the API checks for the input ar-
1917 guments passed successfully. Either way, output matrix A is ready
1918 to be used in the next method of the sequence.

1919 GrB_PANIC Unknown internal error.

1920 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1921 GraphBLAS objects (input or output) is in an invalid state caused
1922 by a previous execution error. Call GrB_error() to access any error
1923 messages generated by the implementation.

1924 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1925 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
1926 any matrix constructor.

1927 **Description**

1928 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,
1929 $\mathbf{L}(\mathbf{A}) = \emptyset$. The dimensions of the matrix do not change.

1930 **4.2.5.6 Matrix_nrows: Number of rows in a matrix**

1931 Retrieve the number of rows in a matrix.

1932 **C Syntax**

```
1933         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
1934                                   const GrB_Matrix A);
```

1935 **Parameters**

1936 nrows (OUT) On successful return, contains the number of rows in the matrix.

1937 A (IN) An existing GraphBLAS matrix being queried.

1938 **Return Values**

1939 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1940 cessfully and the value of `nrows` has been set.

1941 GrB_PANIC Unknown internal error.

1942 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1943 GraphBLAS objects (input or output) is in an invalid state caused
1944 by a previous execution error. Call `GrB_error()` to access any error
1945 messages generated by the implementation.

1946 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
1947 any matrix constructor.

1948 GrB_NULL_POINTER `nrows` pointer is NULL.

1949 **Description**

1950 Return `nrows(A)` in `nrows` (the number of rows).

1951 **4.2.5.7 Matrix_ncols: Number of columns in a matrix**

1952 Retrieve the number of columns in a matrix.

1953 C Syntax

```
1954      GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
1955                               const GrB_Matrix A);
```

1956 Parameters

1957 ncols (OUT) On successful return, contains the number of columns in the matrix.

1958 A (IN) An existing GraphBLAS matrix being queried.

1959 Return Values

1960 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1961 cessfully and the value of ncols has been set.

1962 GrB_PANIC Unknown internal error.

1963 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1964 GraphBLAS objects (input or output) is in an invalid state caused
1965 by a previous execution error. Call GrB_error() to access any error
1966 messages generated by the implementation.

1967 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
1968 any matrix constructor.

1969 GrB_NULL_POINTER ncols pointer is NULL.

1970 Description

1971 Return **ncols(A)** in ncols (the number of columns).

1972 4.2.5.8 Matrix_nvals: Number of stored elements in a matrix

1973 Retrieve the number of stored elements (tuples) in a matrix.

1974 C Syntax

```
1975      GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
1976                               const GrB_Matrix A);
```

1977 **Parameters**

1978 **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in
1979 the matrix.

1980 **A** (IN) An existing GraphBLAS matrix being queried.

1981 **Return Values**

1982 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1983 cessfully and the value of **nvals** has been set.

1984 **GrB_PANIC** Unknown internal error.

1985 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1986 GraphBLAS objects (input or output) is in an invalid state caused
1987 by a previous execution error. Call **GrB_error()** to access any error
1988 messages generated by the implementation.

1989 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1990 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
1991 any matrix constructor.

1992 **GrB_NULL_POINTER** The **nvals** pointer is **NULL**.

1993 **Description**

1994 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of
1995 **L(A)** (see Section ??).

1996 **4.2.5.9 Matrix_build: Store elements from tuples into a matrix**

1997 **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);
```

1998 **Parameters**

1999 **C** (INOUT) An existing Matrix object to store the result.

2000 **row_indices** (IN) Pointer to an array of row indices.

2001 **col_indices** (IN) Pointer to an array of column indices.

2002 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of

2003 matrix, **C**.

2004 **n** (IN) The number of entries contained in each array (the same for **row_indices**,

2005 **col_indices**, and **values**).

2006 **dup** (IN) An associative and commutative binary operator to apply when duplicate

2007 values for the same location are present in the input arrays. All three domains of

2008 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$. If **dup** is **GrB_NULL**,

2009 then duplicate locations will result in an error.

2010 **Return Values**

2011 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-

2012 blocking mode, this indicates that the API checks for the input

2013 arguments passed successfully. Either way, output matrix **C** is

2014 ready to be used in the next method of the sequence.

2015 **GrB_PANIC** Unknown internal error.

2016 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the

2017 opaque GraphBLAS objects (input or output) is in an invalid

2018 state caused by a previous execution error. Call **GrB_error()** to

2019 access any error messages generated by the implementation.

2020 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2021 **GrB_UNINITIALIZED_OBJECT** Either **C** has not been initialized by a call to any matrix construc-

2022 tor, or **dup** has not been initialized by a call to **GrB_BinaryOp_new**.

2023 **GrB_NULL_POINTER** **row_indices**, **col_indices** or **values** pointer is **NULL**.

2024 **GrB_INDEX_OUT_OF_BOUNDS** A value in **row_indices** or **col_indices** is outside the allowed range

2025 for **C**.

2026 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are

2027 not all the same, or the domains of **values** and **C** are incompatible

2028 with each other or D_{dup} .

2029 **GrB_OUTPUT_NOT_EMPTY** Output matrix **C** already contains valid tuples (elements). In

2030 other words, **GrB_Matrix_nvals(C)** returns a positive value.

2031 **GrB_INVALID_VALUE** indices contains a duplicate location and **dup** is **GrB_NULL**.

Description

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

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

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

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

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

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

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

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

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

4.2.5.10 Matrix_setElement: Set a single element in matrix

Set one element of a matrix to a given value.

C Syntax

```
// scalar value
GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
                               <type>         val,
                               GrB_Index        row_index,
                               GrB_Index        col_index);

// GraphBLAS scalar
GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
                               const GrB_Scalar s,
                               GrB_Index        row_index,
                               GrB_Index        col_index);
```

2062 Parameters

2063 **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.
2064 **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of
2065 **C**.
2066 **row_index** (IN) Row index of element to be assigned
2067 **col_index** (IN) Column index of element to be assigned

2068 Return Values

2069 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2070 blocking mode, this indicates that the compatibility tests on in-
2071 dex/dimensions and domains for the input arguments passed suc-
2072 cessfully. Either way, the output matrix **C** is ready to be used in
2073 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**, or GraphBLAS scalar, **s**, has not been
2081 initialized by a call to a respective constructor.
2082 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e., not less
2083 than **nrows(C)** or **ncols(C)**, respectively).
2084 **GrB_DOMAIN_MISMATCH** The domains of the matrix and the scalar are incompatible.

2085 Description

2086 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or
2087 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from
2088 one domain can be cast to values in the other domain as per the rules of the C language. In
2089 particular, domains from Table ?? are all compatible with each other. A domain from a user-
2090 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
2091 **GrB_Matrix_setElement** ends and the domain mismatch error listed above is returned.

2092 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2093 \quad & 0 \leq \text{row_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2094 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid
 2095 index error listed above is returned.

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

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

2096 In the case of a transparent scalar or if $\mathbf{L}(s)$ is not empty, then a value will be stored at the
 2097 specified location in C , overwriting any value that may have been stored there before. In the case
 2098 of a GraphBLAS scalar and if $\mathbf{L}(s)$ is empty, then any value stored at the specified location in C
 2099 will be removed.

2100 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
 2101 of C is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
 2102 return value `GrB_SUCCESS` and the new content of vector C is as defined above but may not be
 2103 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2104 **4.2.5.11 Matrix_removeElement: Remove an element from a matrix**

2105 Remove (annihilate) one stored element from a matrix.

2106 **C Syntax**

```
2107      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2108                                         GrB_Index   row_index,
2109                                         GrB_Index   col_index);
```

2110 **Parameters**

2111 C (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2112 `row_index` (IN) Row index of element to be removed

2113 `col_index` (IN) Column index of element to be removed

2114 **Return Values**

2115 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 2116 blocking mode, this indicates that the compatibility tests on in-
 2117 dex/dimensions and domains for the input arguments passed suc-
 2118 cessfully. Either way, the output matrix C is ready to be used in
 2119 the next method of the sequence.

2120 `GrB_PANIC` Unknown internal error.

2121 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 2122 GraphBLAS objects (input or output) is in an invalid state caused
 2123 by a previous execution error. Call GrB_error() to access any error
 2124 messages generated by the implementation.

2125 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2126 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to
 2127 any matrix constructor.

2128 GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e., not less
 2129 than **nrows(C)** or **ncols(C)**, respectively).

2130 Description

2131 First, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2132 \quad & 0 \leq \text{row_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2133 If either of these conditions is violated, execution of GrB_Matrix_removeElement ends and the
 2134 invalid index error listed above is returned.

2135 We are now ready to carry out the removal of a value that may be stored at the location specified by
 2136 (row_index, col_index). If a value does not exist at the specified location in C, no error is reported
 2137 and the operation has no effect on the state of C. In either case, the following will be true on return
 2138 from this method: (row_index, col_index) \notin ind(C)

2139 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
 2140 of C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
 2141 return value GrB_SUCCESS and the new content of vector C is as defined above but may not be
 2142 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

2143 4.2.5.12 Matrix_extractElement: Extract a single element from a matrix

2144 Extract one element of a matrix into a scalar.

2145 C Syntax

```
2146 // scalar value
2147 GrB_Info GrB_Matrix_extractElement(<type>          *val,
2148                                   const GrB_Matrix A,
2149                                   GrB_Index         row_index,
2150                                   GrB_Index         col_index);
2151
2152 // GraphBLAS scalar
```

```

2153         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2154                                             const GrB_Matrix A,
2155                                             GrB_Index      row_index,
2156                                             GrB_Index      col_index);
2157

```

2158 Parameters

2159 **val or s (INOUT)** An existing scalar whose domain is compatible with the domain of matrix
2160 A. On successful return, this scalar holds the result of the extract. Any previous
2161 value stored in **val** or **s** is overwritten.

2162 **A (IN)** The GraphBLAS matrix from which an element is extracted.

2163 **row_index (IN)** The row index of location in **A** to extract.

2164 **col_index (IN)** The column index of location in **A** to extract.

2165 Return Values

2166 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2167 cessfully. This indicates that the compatibility tests on dimensions
2168 and domains for the input arguments passed successfully, and the
2169 output scalar, **val** or **s**, has been computed and is ready to be used
2170 in the next method of the sequence.

2171 **GrB_NO_VALUE** When using the transparent scalar, **val**, this is returned when there
2172 is no stored value at specified location.

2173 **GrB_PANIC** Unknown internal error.

2174 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2175 GraphBLAS objects (input or output) is in an invalid state caused
2176 by a previous execution error. Call **GrB_error()** to access any error
2177 messages generated by the implementation.

2178 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2179 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by
2180 a call to a corresponding constructor.

2181 **GrB_NULL_POINTER** **val** pointer is **NULL**.

2182 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e. less than
2183 zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-
2184 tively).

2185 **GrB_DOMAIN_MISMATCH** The domains of the matrix and scalar are incompatible.

2186 Description

2187 First, the scalar and input matrix are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ or $\mathbf{D}(\mathbf{s})$
 2188 must be compatible with $\mathbf{D}(\mathbf{A})$. Two domains are compatible with each other if values from
 2189 one domain can be cast to values in the other domain as per the rules of the C language. In
 2190 particular, domains from Table ?? are all compatible with each other. A domain from a user-
 2191 defined type is only compatible with itself. If any compatibility rule above is violated, execution of
 2192 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2193 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2194 \quad & 0 \leq \text{row_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2195 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index
 2196 error listed above is returned.

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

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

2197 If $(\text{row_index}, \text{col_index}) \in \mathbf{ind}(\mathbf{A})$, then the corresponding value from \mathbf{A} is copied into \mathbf{s} or val
 2198 with casting as necessary. If $(\text{row_index}, \text{col_index}) \notin \mathbf{ind}(\mathbf{A})$, then one of the follow occurs
 2199 depending on output scalar type:

- 2200 • The GraphBLAS scalar, \mathbf{s} , is cleared and `GrB_SUCCESS` is returned.
- 2201 • The non-opaque scalar, val , is unchanged, and `GrB_NO_VALUE` is returned.

2202 When using the non-opaque scalar variant (val) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`
 2203 mode, the new contents of val are as defined above if the method exits with return value `GrB_SUCCESS`
 2204 or `GrB_NO_VALUE`.

2205 When using the GraphBLAS scalar variant (\mathbf{s}) with a `GrB_SUCCESS` return value, the method
 2206 exits and the new contents of \mathbf{s} is as defined above and fully computed in `GrB_BLOCKING` mode.
 2207 In `GrB_NONBLOCKING` mode, the new contents of \mathbf{s} is as defined above but may not be fully
 2208 computed; however, it can be used in the next GraphBLAS method call in a sequence.

2209 4.2.5.13 Matrix_extractTuples: Extract tuples from a matrix

2210 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

2211 C Syntax

```
2212      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2213                                       GrB_Index      *col_indices,
```

```

2214                                     <type>          *values,
2215                                     GrB_Index        *n,
2216                                     const GrB_Matrix  A);

```

2217 Parameters

2218 **row_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the
2219 row indices.

2220 **col_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the
2221 column indices.

2222 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
2223 the stored values whose type is compatible with $\mathbf{D}(\mathbf{A})$.

2224 **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,
2225 **row_indices**, and **col_indices** arrays can hold. Upon return, it will contain the
2226 number of values written to the arrays.

2227 **A** (IN) An existing GraphBLAS matrix.

2228 Return Values

2229 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2230 cessfully. This indicates that the compatibility tests on the input
2231 argument passed successfully, and the output arrays, **indices** and
2232 **values**, have been computed.

2233 **GrB_PANIC** Unknown internal error.

2234 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2235 GraphBLAS objects (input or output) is in an invalid state caused
2236 by a previous execution error. Call **GrB_error()** to access any error
2237 messages generated by the implementation.

2238 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2239 **GrB_INSUFFICIENT_SPACE** Not enough space in **row_indices**, **col_indices**, and **values** (as indi-
2240 cated by the **n** parameter) to hold all of the tuples that will be
2241 extracted.

2242 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2243 any matrix constructor.

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

2245 **GrB_DOMAIN_MISMATCH** The domains of the **A** matrix and **values** array are incompatible
2246 with one another.

2247 Description

2248 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with
2249 those tuples are placed in the **values** array, the column indices are placed in the **col_indices** array,
2250 and the row indices are placed in the **row_indices** array. These output arrays are pre-allocated by
2251 the user before calling this function such that each output array has enough space to hold at least
2252 **GrB_Matrix_nvals(A)** elements.

2253 Upon return of this function, a pair of $\{\text{row_indices}[k], \text{col_indices}[k]\}$ are unique for every valid
2254 k , but they are not required to be sorted in any particular order. Each tuple (i, j, A_{ij}) in **A** is
2255 unzipped and copied into a distinct k th location in output vectors:

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

2256 where $0 \leq k < \text{GrB_Matrix_nvals}(v)$. No gaps in output vectors are allowed; that is, if **row_indices**[k],
2257 **col_indices**[k] and **values**[k] exist upon return, so does **row_indices**[j], **col_indices**[j] and **values**[j] for
2258 all j such that $0 \leq j < k$.

2259 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,
2260 then a **GrB_INSUFFICIENT_SPACE** error is returned since it is undefined which subset of values
2261 would be extracted.

2262 In both **GrB_BLOCKING** mode **GrB_NONBLOCKING** mode if the method exits with return value
2263 **GrB_SUCCESS**, the new contents of the arrays **row_indices**, **col_indices** and **values** are as defined
2264 above.

2265 **4.2.5.14 Matrix_exportHint: Provide a hint as to which storage format might be most**
2266 **efficient for exporting a matrix**

2267 C Syntax

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

2268 Parameters

2269 **hint** (OUT) Pointer to a value of type **GrB_Format**.

2270 **A** (IN) A GraphBLAS matrix object.

2271 Return Values

2272 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2273 cessfully and the value of **hint** has been set.

2274 **GrB_PANIC** Unknown internal error.

2275 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 2276 opaque GraphBLAS objects (input or output) is in an invalid
 2277 state caused by a previous execution error. Call GrB_error() to
 2278 access any error messages generated by the implementation.

2279 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2280 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
 2281 any matrix constructor.

2282 GrB_NULL_POINTER hint is NULL.

2283 GrB_NO_VALUE If the implementation does not have a preferred format, it may
 2284 return the value GrB_NO_VALUE.

2285 Description

2286 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for
 2287 exporting the matrix A. GraphBLAS implementations might return the current storage format of
 2288 the matrix, or the format to which it could most efficiently be exported. However, implementations
 2289 are free to return any value for format defined in Section ???. Note that an implementation is free
 2290 to refuse to provide a format hint, returning GrB_NO_VALUE.

2291 4.2.5.15 Matrix_exportSize: Return the array sizes necessary to export a GraphBLAS 2292 matrix object

2293 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);

```

2294 Parameters

2295 n_indptr (OUT) Pointer to a value of type GrB_Index.

2296 n_indices (OUT) Pointer to a value of type GrB_Index.

2297 n_values (OUT) Pointer to a value of type GrB_Index.

2298 format (IN) a value indicating the format in which the matrix will be exported, as defined
 2299 in Section ??.

2300 A (IN) A GraphBLAS matrix object.

2301 Return Values

2302 GrB_SUCCESS In blocking mode or non-blocking mode, the operation com-
 2303 pleted successfully. This indicates that the API checks for the
 2304 input arguments passed successfully, and the number of elements
 2305 necessary for the export buffers have been written to `n_indptr`,
 2306 `n_indices`, and `n_values`, respectively.

2307 GrB_PANIC Unknown internal error.

2308 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 2309 opaque GraphBLAS objects (input or output) is in an invalid
 2310 state caused by a previous execution error. Call `GrB_error()` to
 2311 access any error messages generated by the implementation.

2312 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2313 GrB_UNINITIALIZED_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to
 2314 any matrix constructor.

2315 GrB_NULL_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

2316 Description

2317 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to
 2318 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and
 2319 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be
 2320 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user
 2321 as one of the values defined in Section ??.

2322 4.2.5.16 Matrix_export: Export a GraphBLAS matrix to a pre-defined format

2323 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);
```

2324 Parameters

- 2325 **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-
2326 dices, depending on the value of **format**. It must be large enough to hold at
2327 least **n_indptr** elements of type **GrB_Index**, where **n_indices** was returned from
2328 **GrB_Matrix_exportSize()** method.
- 2329 **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements
2330 in **values**, depending on the value of **format**. It must be large enough to hold at
2331 least **n_indices** elements of type **GrB_Index**, where **n_indices** was returned from
2332 **GrB_Matrix_exportSize()** method.
- 2333 **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-
2334 ment must match the type of the values stored in **A**. It must be large enough
2335 to hold at least **n_values** elements of that type, where **n_values** was returned from
2336 **GrB_Matrix_exportSize**.
- 2337 **n_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**
2338 array can hold. Upon return, it will contain the number of elements written to the
2339 array.
- 2340 **n_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**
2341 array can hold. Upon return, it will contain the number of elements written to the
2342 array.
- 2343 **n_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**
2344 array can hold. Upon return, it will contain the number of elements written to the
2345 array.
- 2346 **format** (IN) a value indicating the format in which the matrix will be exported, as defined
2347 in Section ??.
- 2348 **A** (IN) A GraphBLAS matrix object.

2349 Return Values

- 2350 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2351 cessfully. This indicates that the compatibility tests on the input
2352 argument passed successfully, and the output arrays, **indptr**, **in-**
2353 **dices** and **values**, have been computed.
- 2354 **GrB_PANIC** Unknown internal error.
- 2355 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2356 opaque GraphBLAS objects (input or output) is in an invalid
2357 state caused by a previous execution error. Call **GrB_error()** to
2358 access any error messages generated by the implementation.
- 2359 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2360 GrB_INSUFFICIENT_SPACE Not enough space in `indptr`, `indices`, and/or `values` (as indicated
2361 by the corresponding `n_*` parameter) to hold all of the corre-
2362 sponding elements that will be extacted.

2363 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
2364 any matrix constructor.

2365 GrB_NULL_POINTER `indptr`, `indices`, `values` `n_indptr`, `n_indices`, `n_values` pointer is
2366 NULL.

2367 GrB_DOMAIN_MISMATCH The domain of `A` does not match with the type of `values`.

2368 Description

2369 Given a matrix `A`, this method exports the contents of the matrix into one of the pre-defined
2370 `GrB_Format` formats from Section ???. The user-allocated arrays pointed to by `indptr`, `indices`,
2371 and `values` must be at least large enough to hold the corresponding number of elements returned
2372 by calling `GrB_Matrix_exportSize`. The value of `format` can be chosen arbitrarily, but a call to
2373 `GrB_Matrix_exportHint` may suggest a format that results in the most efficient export. Details
2374 of the contents of `indptr`, `indices`, and `values` corresponding to each supported format is given in
2375 Appendix ??.

2376 4.2.5.17 Matrix_import: Import a matrix into a GraphBLAS object

2377 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);

```

2378 Parameters

2379 `A` (INOUT) On a successful return, contains a handle to the newly created Graph-
2380 BLAS matrix.

2381 `d` (IN) The type corresponding to the domain of the matrix being created. Can be
2382 one of the predefined GraphBLAS types in Table ??, or an existing user-defined
2383 GraphBLAS type.

2384 **nrows** (IN) Integer value holding the number of rows in the matrix.

2385 **ncols** (IN) Integer value holding the number of columns in the matrix.

2386 **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the
2387 value of **format**.

2388 **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending
2389 on the value of **format**.

2390 **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2391 **n_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2392 **n_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2393 **n_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2394 **format** (IN) a value indicating the format of the matrix being imported, as defined in
2395 Section ??.

2396 **Return Values**

2397 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2398 blocking mode, this indicates that the API checks for the input
2399 arguments passed successfully and the input arrays have been
2400 consumed. Either way, output matrix **A** is ready to be used in
2401 the next method of the sequence.

2402 **GrB_PANIC** Unknown internal error.

2403 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2404 **GrB_UNINITIALIZED_OBJECT** The **GrB_Type** object has not been initialized by a call to **GrB_Type_new**
2405 (needed for user-defined types).

2406 **GrB_NULL_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2407 **GrB_INDEX_OUT_OF_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices
2408 in **A** and or the size of **values**, **n_values**, depending on the value
2409 of **format**.

2410 **GrB_INVALID_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB_Index**.

2411 **GrB_DOMAIN_MISMATCH** The domain given in parameter **d** does not match the element
2412 type of **values**.

2413 Description

2414 Creates a new matrix **A** of domain **D**(d) and dimension **nrows** \times **ncols**. The new GraphBLAS
2415 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.
2416 The method returns a handle to the new matrix in **A**. The structure of the data being imported is
2417 defined by **format**, which must be equal to one of the values defined in Section ???. Details of the
2418 contents of **indptr**, **indices** and **values** for each supported format is given in Appendix ???.

2419 It is not an error to call this method more than once on the same output matrix; however, the
2420 handle to the previously created object will be overwritten.

2421 4.2.5.18 Matrix_serializeSize: Compute the serialize buffer size

2422 Compute the buffer size (in bytes) necessary to serialize a GrB_Matrix using GrB_Matrix_serialize.

2423 C Syntax

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

2424 Parameters

2425 **size** (OUT) Pointer to GrB_Index value where size in bytes of serialized object will be
2426 written.

2427 **A** (IN) A GraphBLAS matrix object.

2428 Return Values

2429 **GrB_SUCCESS** The operation completed successfully and the value pointed to
2430 by ***size** has been computed and is ready to use.

2431 **GrB_PANIC** Unknown internal error.

2432 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2433 **GrB_NULL_POINTER** **size** is NULL.

2434 Description

2435 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object **A**.
2436 Users may then allocate a buffer of **size** bytes to pass as a parameter to GrB_Matrix_serialize.

2437 **4.2.5.19 Matrix_serialize: Serialize a GraphBLAS matrix.**

2438 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2439 **C Syntax**

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

2440 **Parameters**

2441 **serialized_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be
2442 written.

2443 **serialized_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized_data**.
2444 On output, the number of bytes written to **serialized_data**.

2445 **A** (IN) A GraphBLAS matrix object.

2446 **Return Values**

2447 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
2448 cessfully. This indicates that the compatibility tests on the in-
2449 put argument passed successfully, and the output buffer **serial-
2450 ized_data** and **serialized_size**, have been computed and are ready
2451 to use.

2452 **GrB_PANIC** Unknown internal error.

2453 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
2454 opaque GraphBLAS objects (input or output) is in an invalid
2455 state caused by a previous execution error. Call **GrB_error()** to
2456 access any error messages generated by the implementation.

2457 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2458 **GrB_NULL_POINTER** **serialized_data** or **serialize_size** is NULL.

2459 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
2460 any matrix constructor.

2461 **GrB_INSUFFICIENT_SPACE** The size of the buffer **serialized_data** (provided as an input **seri-
2462 alized_size**) was not large enough.

2463 Description

2464 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,
2465 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must
2466 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the
2467 serialized matrix written to `serialized_data` is provided upon completion as an output written to
2468 `serialized_size`.

2469 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created
2470 with one library implementation is not necessarily valid for deserialization with another implemen-
2471 tation.

2472 4.2.5.20 Matrix_deserialize: Deserialize a GraphBLAS matrix.

2473 Construct a new GraphBLAS matrix from a serialized object.

2474 C Syntax

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

2475 Parameters

2476 A (INOUT) On a successful return, contains a handle to the newly created Graph-
2477 BLAS matrix.

2478 d (IN) the type of the matrix that was serialized in `serialized_data`.

2479 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2480 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

2481 Return Values

2482 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2483 blocking mode, this indicates that the API checks for the input
2484 arguments passed successfully. Either way, output matrix A is
2485 ready to be used in the next method of the sequence.

2486 GrB_PANIC Unknown internal error.

2487 GrB_INVALID_OBJECT This is returned if `serialized_data` is invalid or corrupted.

2488 GrB_OUT_OF_MEMORY Not enough memory available for operation.

2489 GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to GrB_Type_new
2490 (needed for user-defined types).

2491 GrB_NULL_POINTER serialized_data or A is NULL.

2492 GrB_DOMAIN_MISMATCH The type given in d does not match the type of the matrix
2493 serialized in serialized_data.

2494 Description

2495 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object
2496 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The
2497 domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized
2498 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2499 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix
2500 serialized in one library implementation can be deserialized by another.

2501 It is not an error to call this method more than once on the same output matrix; however, the
2502 handle to the previously created object will be overwritten.

2503 4.2.6 Descriptor methods

2504 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-
2505 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

2506 4.2.6.1 Descriptor_new: Create new descriptor

2507 Creates a new (empty or default) descriptor.

2508 C Syntax

2509 GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);

2510 Parameters

2511 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS
2512 descriptor.

2513 Return Value

2514 GrB_SUCCESS The method completed successfully.

2515 GrB_PANIC unknown internal error.

2516 GrB_OUT_OF_MEMORY not enough memory available for operation.

2517 GrB_NULL_POINTER desc pointer is NULL.

2518 **Description**

2519 Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can
2520 be populated by calls to Descriptor_set.

2521 It is not an error to call this method more than once on the same variable; however, the handle to
2522 the previously created object will be overwritten.

2523 **4.2.6.2 Descriptor_set: Set content of descriptor**

2524 Sets the content for a field for an existing descriptor.

2525 **C Syntax**

```
2526        GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2527                                    GrB_Desc_Field        field,  
2528                                    GrB_Desc_Value        val);
```

2529 **Parameters**

2530 desc (IN) An existing GraphBLAS descriptor to be modified.

2531 field (IN) The field being set.

2532 val (IN) New value for the field being set.

2533 **Return Values**

2534 GrB_SUCCESS operation completed successfully.

2535 GrB_PANIC unknown internal error.

2536 GrB_OUT_OF_MEMORY not enough memory available for operation.

2537 GrB_UNINITIALIZED_OBJECT the desc parameter has not been initialized by a call to new.

2538 GrB_INVALID_VALUE invalid value set on the field, or invalid field.

2539 Description

2540 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor
2541 to set the value associated with that field. Valid values for the `field` parameter include the following:

2542 `GrB_OUTP` refers to the output parameter (result) of the operation.

2543 `GrB_MASK` refers to the mask parameter of the operation.

2544 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2545 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2546 Valid values for the `val` parameter are:

2547 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask
2548 (`GrB_MASK`) parameter.

2549 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-
2550 eter. When combined with `GrB_STRUCTURE`, the complement of the
2551 structure of the mask is used without evaluating the values stored.

2552 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input
2553 matrix parameters only).

2554 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear
2555 the matrix first (or clear the non-masked entries). The default behavior
2556 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`
2557 parameter only.

2558 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,
2559 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).
2560 A value for a given field may be set multiple times but will have no additional effect. Fields that
2561 have no values set result in their default behavior, as defined in Section ??.

2562 4.2.7 free: Destroy an object and release its resources

2563 Destroys a previously created GraphBLAS object and releases any resources associated with the
2564 object.

2565 C Syntax

2566 `GrB_Info GrB_free(<GrB_Object> *obj);`

2567 Parameters

2568 **obj** (INOUT) An existing GraphBLAS object to be destroyed. The object must have
2569 been created by an explicit call to a GraphBLAS constructor. It can be any of the
2570 opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,
2571 binary op, unary op, or type. On successful completion of **GrB_free**, **obj** behaves
2572 as an uninitialized object.

2573 Return Values

2574 **GrB_SUCCESS** operation completed successfully

2575 **GrB_PANIC** unknown internal error. If this return value is encountered when
2576 in nonblocking mode, the error responsible for the panic condition
2577 could be from any method involved in the computation of the input
2578 object. The **GrB_error()** method should be called for additional
2579 information.

2580 Description

2581 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime
2582 system. A call to **GrB_free** frees those resources so they are available for use by other GraphBLAS
2583 objects.

2584 The parameter passed into **GrB_free** is a handle referencing a GraphBLAS opaque object of a data
2585 type from table **??**. The object must have been created by an explicit call to a GraphBLAS con-
2586 structor. The behavior of a program that calls **GrB_free** on a pre-defined object is implementation
2587 defined.

2588 After the **GrB_free** method returns, the object referenced by the input handle is destroyed and the
2589 handle has the value **GrB_INVALID_HANDLE**. The handle can be used in subsequent GraphBLAS
2590 methods but only after the handle has been reinitialized with a call the the appropriate **_new** or
2591 **_dup** method.

2592 Note that unlike other GraphBLAS methods, calling **GrB_free** with an object with an invalid handle
2593 is legal. The system may attempt to free resources that might be associated with that object, if
2594 possible, and return normally.

2595 When using **GrB_free** it is possible to create a dangling reference to an object. This would occur
2596 when a handle is assigned to a second variable of the same opaque type. This creates two handles
2597 that reference the same object. If **GrB_free** is called with one of the variables, the object is destroyed
2598 and the handle associated with the other variable no longer references a valid object. This is not an
2599 error condition that the implementation of the GraphBLAS API can be expected to catch, hence
2600 programmers must take care to prevent this situation from occurring.

2601 **4.2.8 wait: Return once an object is either *complete* or *materialized***

2602 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2603 **C Syntax**

2604 `GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);`

2605 **Parameters**

2606 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an
2607 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
2608 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
2609 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another
2610 thread (completion) or all computing to produce `obj` by all GraphBLAS operations
2611 in its sequence have finished (materialization).

2612 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the
2613 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or
2614 `GrB_MATERIALIZE`.

2615 **Return values**

2616 `GrB_SUCCESS` operation completed successfully.

2617 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-
2618 pletion of pending operations.

2619 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion
2620 of pending operations.

2621 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,
2622 or other constructor, method.

2623 `GrB_PANIC` unknown internal error.

2624 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`
2625 `GrB_MATERIALIZE`.

2626 **Description**

2627 On successful return from `GrB_wait()`, the input object, `obj` is in one of two states depending on
2628 the mode of `GrB_wait`:

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

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

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

4.2.9 error: Retrieve an error string

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

C Syntax

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

Parameters

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

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

Return value

`GrB_SUCCESS` operation completed successfully.

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

`GrB_PANIC` unknown internal error.

Description

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

4.3 GraphBLAS operations

The GraphBLAS operations are defined in the GraphBLAS math specification and summarized in Table ???. 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 ???.

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

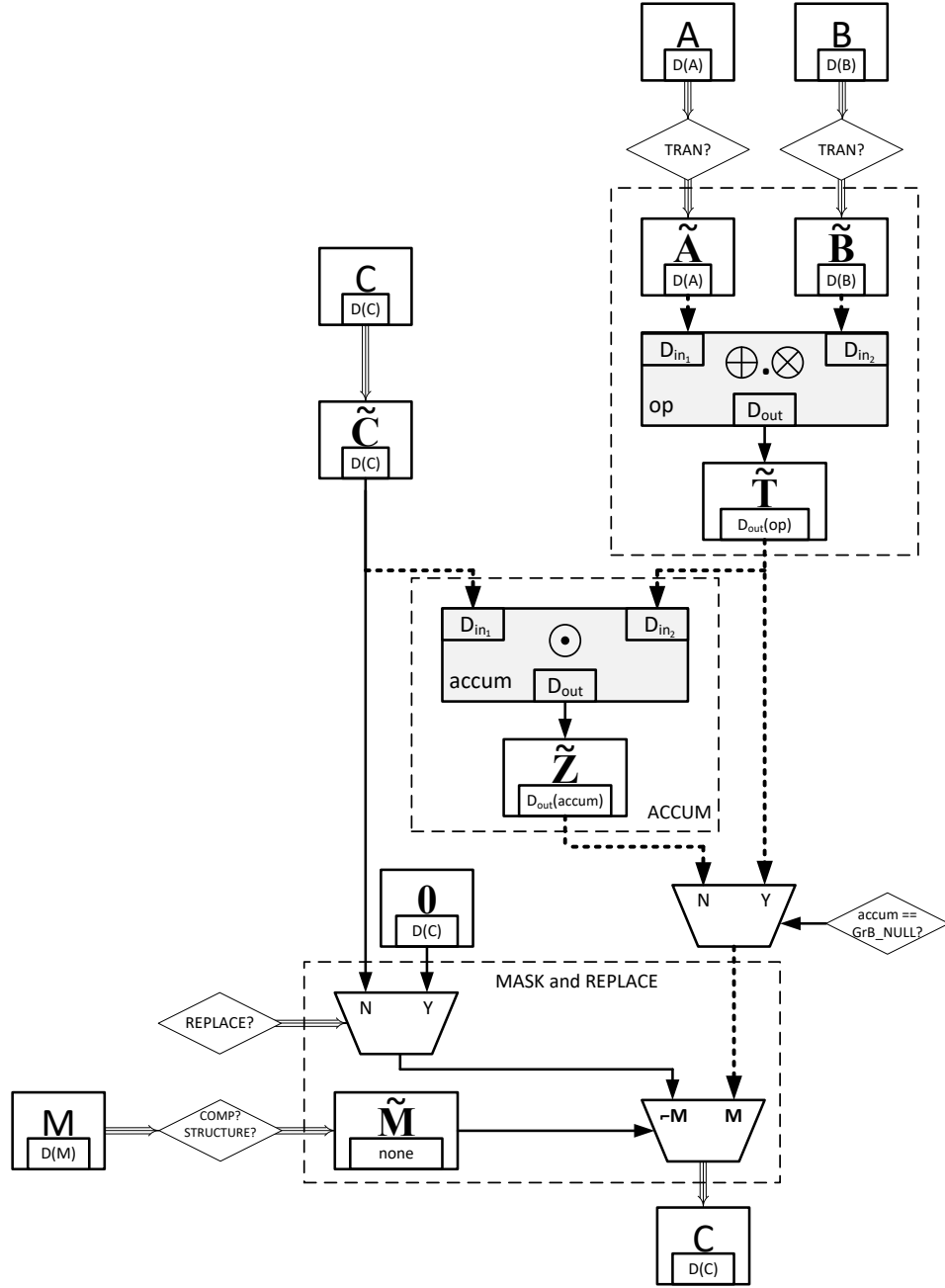


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows (\Rightarrow) denote where “as if copy” takes place (including both collections and descriptor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

2729 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-
 2730 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at
 2731 the point of their declaration, as shown in the following examples:

```
2732         GrB_Type          type = GrB_INVALID_HANDLE;
2733         GrB_Semiring      semiring = GrB_INVALID_HANDLE;
2734         GrB_Matrix       matrix = GrB_INVALID_HANDLE;
```

2735 Compliance

2736 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.
 2737 That is, for each operation we give a recipe for producing its outcome. Any implementation that
 2738 produces the same outcome, and follows the GraphBLAS execution model (Section ??) and error
 2739 model (Section ??) is a conforming implementation.

2740 4.3.1 mxm: Matrix-matrix multiply

2741 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

2742 C Syntax

```
2743         GrB_Info GrB_mxm(GrB_Matrix          C,
2744                         const GrB_Matrix     Mask,
2745                         const GrB_BinaryOp    accum,
2746                         const GrB_Semiring    op,
2747                         const GrB_Matrix     A,
2748                         const GrB_Matrix     B,
2749                         const GrB_Descriptor  desc);
```

2750 Parameters

2751 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 2752 that may be accumulated with the result of the matrix product. On output, the
 2753 matrix holds the results of the operation.

2754 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 2755 stored into the output matrix C. The mask dimensions must match those of the
 2756 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
 2757 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
 2758 in Table ?. If the default mask is desired (i.e., a mask that is all true with the
 2759 dimensions of C), `GrB_NULL` should be specified.

2760 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
 2761 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 2762 specified.

2763 **op** (IN) The semiring used in the matrix-matrix multiply.

2764 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 2765 multiplication.

2766 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 2767 multiplication.

2768 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 2769 should be specified. Non-default field/value pairs are listed as follows:
 2770

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.

2772 Return Values

2773 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 2774 blocking mode, this indicates that the compatibility tests on di-
 2775 mensions and domains for the input arguments passed successfully.
 2776 Either way, output matrix **C** is ready to be used in the next method
 2777 of the sequence.

2778 **GrB_PANIC** Unknown internal error.

2779 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 2780 GraphBLAS objects (input or output) is in an invalid state caused
 2781 by a previous execution error. Call **GrB_error()** to access any error
 2782 messages generated by the implementation.

2783 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

2784 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 2785 a call to **new** (or **Matrix_dup** for matrix parameters).

2786 **GrB_DIMENSION_MISMATCH** Mask and/or matrix dimensions are incompatible.

2787 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 2788 corresponding domains of the semiring or accumulation operator,
 2789 or the mask's domain is not compatible with `bool` (in the case where
 2790 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

2791 Description

2792 GrB_mxm computes the matrix product $C = A \oplus . \otimes B$ or, if an optional binary accumulation operator
 2793 (\odot) is provided, $C = C \odot (A \oplus . \otimes B)$ (where matrices A and B can be optionally transposed).
 2794 Logically, this operation occurs in three steps:

2795 **Setup** The internal matrices and mask used in the computation are formed and their domains
 2796 and dimensions are tested for compatibility.

2797 **Compute** The indicated computations are carried out.

2798 **Output** The result is written into the output matrix, possibly under control of a mask.

2799 Up to four argument matrices are used in the GrB_mxm operation:

- 2800 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 2801 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)
- 2802 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 2803 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

2804 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for
 2805 domain compatibility as follows:

- 2806 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 2807 must be from one of the pre-defined types of Table ??.
- 2808 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 2809 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 2810 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 2811 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 2812 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 2813 of the accumulation operator.

2814 Two domains are compatible with each other if values from one domain can be cast to values in
 2815 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 2816 compatible with each other. A domain from a user-defined type is only compatible with itself. If

2817 any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch
 2818 error listed above is returned.

2819 From the argument matrices, the internal matrices and mask used in the computation are formed
 2820 (\leftarrow denotes copy):

- 2821 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 2822 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 2823 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 2824 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 2825 (b) If `Mask \neq GrB_NULL`,
 - 2826 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 2827 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 2828 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 2829 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 2830 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 2831 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 2832 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

2833 The internal matrices and masks are checked for dimension compatibility. The following conditions
 2834 must hold:

- 2835 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 2836 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 2837 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 2838 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.
- 2839 5. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.

2840 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch
 2841 error listed above is returned.

2842 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 2843 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2844 We are now ready to carry out the matrix multiplication and any additional associated operations.
 2845 We describe this in terms of two intermediate matrices:

- 2846 • $\tilde{\mathbf{T}}$: The matrix holding the product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 2847 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

2848 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

$$2850 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

2851 where \oplus and \otimes are the additive and multiplicative operators of semiring \mathbf{op} , respectively.

2852 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 2853 • If $\mathbf{accum} = \mathbf{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 2854 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$2855 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

2856 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
2857 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 2858 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 2859 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 2860 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 2861 \end{aligned}$$

2862 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

2864 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
2865 using what is called a *standard matrix mask and replace*. This is carried out under control of the
2866 mask which acts as a “write mask”.

- 2867 • If $\mathbf{desc}[\mathbf{GrB_OUTP}].\mathbf{GrB_REPLACE}$ is set, then any values in \mathbf{C} on input to this operation are
2868 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$2869 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 2870 • If $\mathbf{desc}[\mathbf{GrB_OUTP}].\mathbf{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
2871 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
2872 mask are unchanged:

$$2873 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

2874 In $\mathbf{GrB_BLOCKING}$ mode, the method exits with return value $\mathbf{GrB_SUCCESS}$ and the new content
2875 of matrix \mathbf{C} is as defined above and fully computed. In $\mathbf{GrB_NONBLOCKING}$ mode, the method
2876 exits with return value $\mathbf{GrB_SUCCESS}$ and the new content of matrix \mathbf{C} is as defined above but
2877 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
2878 sequence.

2879 4.3.2 vxm: Vector-matrix multiply

2880 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

2881 C Syntax

```
2882      GrB_Info GrB_vxm(GrB_Vector      w,  
2883                      const GrB_Vector mask,  
2884                      const GrB_BinaryOp accum,  
2885                      const GrB_Semiring op,  
2886                      const GrB_Vector u,  
2887                      const GrB_Matrix A,  
2888                      const GrB_Descriptor desc);
```

2889 Parameters

2890 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
2891 that may be accumulated with the result of the vector-matrix product. On output,
2892 this vector holds the results of the operation.

2893 **mask** (IN) An optional “write” mask that controls which results from this operation are
2894 stored into the output vector **w**. The mask dimensions must match those of the
2895 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
2896 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
2897 in Table ???. If the default mask is desired (i.e., a mask that is all **true** with the
2898 dimensions of **w**), **GrB_NULL** should be specified.

2899 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
2900 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
2901 specified.

2902 **op** (IN) Semiring used in the vector-matrix multiply.

2903 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
2904 multiplication.

2905 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
2906 multiplication.

2907 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
2908 should be specified. Non-default field/value pairs are listed as follows:

2909

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.

2937 **Output** The result is written into the output vector, possibly under control of a mask.

2938 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 2939 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 2940 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 2941 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 2942 4. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

2943 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
 2944 tested for domain compatibility as follows:

- 2945 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 2946 must be from one of the pre-defined types of Table ??.
- 2947 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 2948 3. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 2949 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.
- 2950 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 2951 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$
 2952 of the accumulation operator.

2953 Two domains are compatible with each other if values from one domain can be cast to values in
 2954 the other domain as per the rules of the C language. In particular, domains from Table ?? are
 2955 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 2956 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch
 2957 error listed above is returned.

2958 From the argument vectors and matrices, the internal matrices and mask used in the computation
 2959 are formed (\leftarrow denotes copy):

- 2960 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 2961 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 2962 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 2963 (b) If `mask` \neq `GrB_NULL`,
 - 2964 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 2965 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$.
 - 2966 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 2967 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

2968 4. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

2969 The internal matrices and masks are checked for shape compatibility. The following conditions
2970 must hold:

2971 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.

2972 2. $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$.

2973 3. $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$.

2974 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch
2975 error listed above is returned.

2976 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
2977 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2978 We are now ready to carry out the vector-matrix multiplication and any additional associated
2979 operations. We describe this in terms of two intermediate vectors:

- 2980 • $\tilde{\mathbf{t}}$: The vector holding the product of vector $\tilde{\mathbf{u}}^T$ and matrix $\tilde{\mathbf{A}}$.
- 2981 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

2982 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.
2983 The value of each of its elements is computed by

$$2984 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

2985 where \oplus and \otimes are the additive and multiplicative operators of semiring `op`, respectively.

2986 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 2987 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 2988 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$2989 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

2990 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
2991 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 2992 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 2993 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 2994 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 2995 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 2996 \end{aligned}$$

2997 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

2998 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 2999 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3000 mask which acts as a “write mask”.

- 3001 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 3002 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3003 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3004 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3005 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3006 mask are unchanged:

$$3007 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3008 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 3009 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 3010 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 3011 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3012 sequence.

3013 4.3.3 mxv: Matrix-vector multiply

3014 Multiplies a matrix by a vector on a semiring. The result is a vector.

3015 C Syntax

```
3016      GrB_Info GrB_mxv(GrB_Vector      w,
3017                      const GrB_Vector mask,
3018                      const GrB_BinaryOp accum,
3019                      const GrB_Semiring op,
3020                      const GrB_Matrix A,
3021                      const GrB_Vector u,
3022                      const GrB_Descriptor desc);
```

3023 Parameters

3024 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 3025 that may be accumulated with the result of the matrix-vector product. On output,
 3026 this vector holds the results of the operation.

3027 **mask** (IN) An optional “write” mask that controls which results from this operation are
 3028 stored into the output vector \mathbf{w} . The mask dimensions must match those of the
 3029 vector \mathbf{w} . If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain

3030 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types
 3031 in Table ???. If the default mask is desired (i.e., a mask that is all `true` with the
 3032 dimensions of `w`), `GrB_NULL` should be specified.

3033 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`
 3034 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 3035 specified.

3036 `op` (IN) Semiring used in the vector-matrix multiply.

3037 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3038 multiplication.

3039 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3040 multiplication.

3041 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 3042 should be specified. Non-default field/value pairs are listed as follows:
 3043

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.

3045 Return Values

3046 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 3047 blocking mode, this indicates that the compatibility tests on di-
 3048 mensions and domains for the input arguments passed successfully.
 3049 Either way, output vector `w` is ready to be used in the next method
 3050 of the sequence.

3051 `GrB_PANIC` Unknown internal error.

3052 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 3053 GraphBLAS objects (input or output) is in an invalid state caused
 3054 by a previous execution error. Call `GrB_error()` to access any error
 3055 messages generated by the implementation.

3056 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3057 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by
 3058 a call to `new` (or `dup` for matrix or vector parameters).

3059 GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3060 GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with
3061 the corresponding domains of the semiring or accumulation opera-
3062 tor, or the mask's domain is not compatible with **bool** (in the case
3063 where desc[GrB_MASK].GrB_STRUCTURE is not set).

3064 Description

3065 GrB_mvx computes the matrix-vector product $w = A \oplus . \otimes u$, or, if an optional binary accumulation
3066 operator (\odot) is provided, $w = w \odot (A \oplus . \otimes u)$ (where matrix A can be optionally transposed).
3067 Logically, this operation occurs in three steps:

3068 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
3069 domains/dimensions are tested for compatibility.

3070 **Compute** The indicated computations are carried out.

3071 **Output** The result is written into the output vector, possibly under control of a mask.

3072 Up to four argument vectors or matrices are used in the GrB_mvx operation:

- 3073 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3074 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3075 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3076 4. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3077 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
3078 tested for domain compatibility as follows:

- 3079 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
3080 must be from one of the pre-defined types of Table ??.
- 3081 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 3082 3. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 3083 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 3084 5. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3085 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
3086 of the accumulation operator.

3087 Two domains are compatible with each other if values from one domain can be cast to values in
 3088 the other domain as per the rules of the C language. In particular, domains from Table ?? are
 3089 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 3090 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the domain mismatch
 3091 error listed above is returned.

3092 From the argument vectors and matrices, the internal matrices and mask used in the computation
 3093 are formed (\leftarrow denotes copy):

- 3094 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3095 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3096 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3097 (b) If `mask \neq GrB_NULL`,
 - 3098 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3099 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3100 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3101 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3102 4. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3103 The internal matrices and masks are checked for shape compatibility. The following conditions
 3104 must hold:

- 3105 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.
- 3106 2. $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}(\tilde{\mathbf{A}})$.
- 3107 3. $\text{size}(\tilde{\mathbf{u}}) = \text{ncols}(\tilde{\mathbf{A}})$.

3108 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the dimension mismatch
 3109 error listed above is returned.

3110 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3111 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3112 We are now ready to carry out the matrix-vector multiplication and any additional associated
 3113 operations. We describe this in terms of two intermediate vectors:

- 3114 • $\tilde{\mathbf{t}}$: The vector holding the product of matrix $\tilde{\mathbf{A}}$ and vector $\tilde{\mathbf{u}}$.
- 3115 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3116 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.
 3117 The value of each of its elements is computed by

$$3118 \quad t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3119 where \oplus and \otimes are the additive and multiplicative operators of semiring **op**, respectively.

3120 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3121 • If **accum** = GrB_NULL, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3122 • If **accum** is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3123 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3124 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3125 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 3126 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3127 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3128 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3129 \quad & \\ 3130 \end{aligned}$$

3131 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3132 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector **w**,
 3133 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3134 mask which acts as a “write mask”.

- 3135 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in **w** on input to this operation are
 3136 deleted and the content of the new output vector, **w**, is defined as,

$$3137 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3138 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3139 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the
 3140 mask are unchanged:

$$3141 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3142 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 3143 of vector **w** is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 3144 exits with return value GrB_SUCCESS and the new content of vector **w** is as defined above but
 3145 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3146 sequence.

3147 4.3.4 eWiseMult: Element-wise multiplication

3148 **Note:** The difference between eWiseAdd and eWiseMult is not about the element-wise operation
 3149 but how the index sets are treated. eWiseAdd returns an object whose indices are the “union” of
 3150 the indices of the inputs whereas eWiseMult returns an object whose indices are the “intersection”
 3151 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
 3152 the set of values from the resulting index set.

3153 4.3.4.1 eWiseMult: Vector variant

3154 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-
3155 ducing a third vector as result.

3156 C Syntax

```
3157     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3158                           const GrB_Vector mask,  
3159                           const GrB_BinaryOp accum,  
3160                           const GrB_Semiring op,  
3161                           const GrB_Vector u,  
3162                           const GrB_Vector v,  
3163                           const GrB_Descriptor desc);  
3164  
3165     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3166                           const GrB_Vector mask,  
3167                           const GrB_BinaryOp accum,  
3168                           const GrB_Monoid op,  
3169                           const GrB_Vector u,  
3170                           const GrB_Vector v,  
3171                           const GrB_Descriptor desc);  
3172  
3173     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3174                           const GrB_Vector mask,  
3175                           const GrB_BinaryOp accum,  
3176                           const GrB_BinaryOp op,  
3177                           const GrB_Vector u,  
3178                           const GrB_Vector v,  
3179                           const GrB_Descriptor desc);
```

3180 Parameters

3181 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3182 that may be accumulated with the result of the element-wise operation. On output,
3183 this vector holds the results of the operation.

3184 **mask** (IN) An optional “write” mask that controls which results from this operation are
3185 stored into the output vector **w**. The mask dimensions must match those of the
3186 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3187 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3188 in Table ???. If the default mask is desired (i.e., a mask that is all **true** with the
3189 dimensions of **w**), **GrB_NULL** should be specified.

3190 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3191 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 3192 specified.

3193 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
 3194 operation. Depending on which type is passed, the following defines the binary
 3195 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3196 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3197 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-
 3198 nored.

3199 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
 3200 is ignored.

3201 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 3202 operation.

3203 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3204 operation.

3205 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3206 should be specified. Non-default field/value pairs are listed as follows:
 3207

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.

3209 Return Values

3210 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3211 blocking mode, this indicates that the compatibility tests on di-
 3212 mensions and domains for the input arguments passed successfully.
 3213 Either way, output vector w is ready to be used in the next method
 3214 of the sequence.

3215 GrB_PANIC Unknown internal error.

3216 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3217 GraphBLAS objects (input or output) is in an invalid state caused
 3218 by a previous execution error. Call GrB_error() to access any error
 3219 messages generated by the implementation.

3220 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3221 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3222 a call to `new` (or `dup` for vector parameters).

3223 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3224 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 3225 responding domains of the binary operator (`op`) or accumulation
 3226 operator, or the mask's domain is not compatible with `bool` (in the
 3227 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3228 Description

3229 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:
 3230 $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$.
 3231 Logically, this operation occurs in three steps:

3232 **Setup** The internal vectors and mask used in the computation are formed and their domains
 3233 and dimensions are tested for compatibility.

3234 **Compute** The indicated computations are carried out.

3235 **Output** The result is written into the output vector, possibly under control of a mask.

3236 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3237 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3238 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3239 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3240 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3241 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)
 3242 are tested for domain compatibility as follows:

- 3243 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 3244 must be from one of the pre-defined types of Table ??.
- 3245 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 3246 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 3247 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 3248 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 3249 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
 3250 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 ?? are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the domain mismatch error listed above is returned.

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

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

The internal vectors and mask are checked for dimension compatibility. The following conditions must hold:

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

If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the dimension mismatch error listed above is returned.

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

We are now ready to carry out the element-wise “product” and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$: The vector holding the element-wise “product” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$ is created. The value of each of its elements is computed by:

$$t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))$$

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

3282 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3283 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

3284
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3285 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3286 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

3287
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3288

3289
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3290

3291
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3292 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3293 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3294 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3295 mask which acts as a “write mask”.

3296 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in \mathbf{w} on input to this operation are
 3297 deleted and the content of the new output vector, \mathbf{w} , is defined as,

3298
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3299 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3300 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3301 mask are unchanged:

3302
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3303 In **GrB_BLOCKING** mode, the method exits with return value **GrB_SUCCESS** and the new content
 3304 of vector \mathbf{w} is as defined above and fully computed. In **GrB_NONBLOCKING** mode, the method
 3305 exits with return value **GrB_SUCCESS** and the new content of vector \mathbf{w} is as defined above but
 3306 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3307 sequence.

3308 4.3.4.2 eWiseMult: Matrix variant

3309 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-
 3310 ducing a third matrix as result.

3311 C Syntax

```

3312     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3313                           const GrB_Matrix  Mask,
3314                           const GrB_BinaryOp accum,
3315                           const GrB_Semiring op,
3316                           const GrB_Matrix  A,
3317                           const GrB_Matrix  B,
3318                           const GrB_Descriptor desc);
3319
3320     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3321                           const GrB_Matrix  Mask,
3322                           const GrB_BinaryOp accum,
3323                           const GrB_Monoid  op,
3324                           const GrB_Matrix  A,
3325                           const GrB_Matrix  B,
3326                           const GrB_Descriptor desc);
3327
3328     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3329                           const GrB_Matrix  Mask,
3330                           const GrB_BinaryOp accum,
3331                           const GrB_BinaryOp op,
3332                           const GrB_Matrix  A,
3333                           const GrB_Matrix  B,
3334                           const GrB_Descriptor desc);

```

3335 Parameters

3336 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3337 that may be accumulated with the result of the element-wise operation. On output,
3338 the matrix holds the results of the operation.

3339 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3340 stored into the output matrix C. The mask dimensions must match those of the
3341 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3342 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types
3343 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
3344 dimensions of C), `GrB_NULL` should be specified.

3345 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3346 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3347 specified.

3348 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
3349 operation. Depending on which type is passed, the following defines the binary
3350 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

3351 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 3352 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-
 3353 nored.
 3354 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
 3355 is ignored.

3356 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 3357 operation.

3358 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 3359 operation.

3360 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3361 should be specified. Non-default field/value pairs are listed as follows:
 3362

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.

3364 Return Values

3365 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3366 blocking mode, this indicates that the compatibility tests on di-
 3367 mensions and domains for the input arguments passed successfully.
 3368 Either way, output matrix C is ready to be used in the next method
 3369 of the sequence.

3370 GrB_PANIC Unknown internal error.

3371 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3372 GraphBLAS objects (input or output) is in an invalid state caused
 3373 by a previous execution error. Call GrB_error() to access any error
 3374 messages generated by the implementation.

3375 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3376 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3377 a call to new (or Matrix_dup for matrix parameters).

3378 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3379 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3380 corresponding domains of the binary operator (\otimes) or accumulation
 3381 operator, or the mask's domain is not compatible with `bool` (in the
 3382 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3383 Description

3384 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:
 3385 $C = A \otimes B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \otimes B)$.
 3386 Logically, this operation occurs in three steps:

3387 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3388 and dimensions are tested for compatibility.

3389 **Compute** The indicated computations are carried out.

3390 **Output** The result is written into the output matrix, possibly under control of a mask.

3391 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3392 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3393 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)
- 3394 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3395 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3396 The argument matrices, the “product” operator (\otimes), and the accumulation operator (if provided)
 3397 are tested for domain compatibility as follows:

- 3398 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3399 must be from one of the pre-defined types of Table ??.
- 3400 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\otimes)$.
- 3401 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\otimes)$.
- 3402 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\otimes)$.
- 3403 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3404 of the accumulation operator and $\mathbf{D}_{out}(\otimes)$ of \otimes must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3405 the accumulation operator.

3406 Two domains are compatible with each other if values from one domain can be cast to values in
 3407 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 3408 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3409 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch
 3410 error listed above is returned.

3411 From the argument matrices, the internal matrices and mask used in the computation are formed
 3412 (\leftarrow denotes copy):

- 3413 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3414 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3415 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3416 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3417 (b) If `Mask \neq GrB_NULL`,
 - 3418 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3419 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3420 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3421 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$.
 - 3422 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3423 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3424 4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3425 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3426 must hold:

- 3427 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3428 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3429 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3430 mismatch error listed above is returned.

3431 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3432 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3433 We are now ready to carry out the element-wise “product” and any additional associated operations.
 3434 We describe this in terms of two intermediate matrices:

- 3435 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise product of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3436 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3437 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$
 3438 is created. The value of each of its elements is computed by

$$3439 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3440 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

3441 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

3442 • If accum is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3443 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3444 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
3445 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3446 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3447 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3449 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3451 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3452 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
3453 using what is called a *standard matrix mask and replace*. This is carried out under control of the
3454 mask which acts as a “write mask”.

3455 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in \mathbf{C} on input to this operation are
3456 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3457 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3458 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
3459 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
3460 mask are unchanged:

$$3461 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3462 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
3463 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
3464 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
3465 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3466 sequence.

3467 4.3.5 eWiseAdd: Element-wise addition

3468 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation
3469 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of
3470 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”
3471 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
3472 the set of values from the resulting index set.

3473 4.3.5.1 eWiseAdd: Vector variant

3474 Perform element-wise (general) addition on the elements of two vectors, producing a third vector
3475 as result.

3476 C Syntax

```
3477     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3478                          const GrB_Vector mask,  
3479                          const GrB_BinaryOp accum,  
3480                          const GrB_Semiring op,  
3481                          const GrB_Vector u,  
3482                          const GrB_Vector v,  
3483                          const GrB_Descriptor desc);  
3484  
3485     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3486                          const GrB_Vector mask,  
3487                          const GrB_BinaryOp accum,  
3488                          const GrB_Monoid op,  
3489                          const GrB_Vector u,  
3490                          const GrB_Vector v,  
3491                          const GrB_Descriptor desc);  
3492  
3493     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3494                          const GrB_Vector mask,  
3495                          const GrB_BinaryOp accum,  
3496                          const GrB_BinaryOp op,  
3497                          const GrB_Vector u,  
3498                          const GrB_Vector v,  
3499                          const GrB_Descriptor desc);
```

3500 Parameters

3501 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3502 that may be accumulated with the result of the element-wise operation. On output,
3503 this vector holds the results of the operation.

3504 **mask** (IN) An optional “write” mask that controls which results from this operation are
3505 stored into the output vector **w**. The mask dimensions must match those of the
3506 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3507 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3508 in Table ???. If the default mask is desired (i.e., a mask that is all **true** with the
3509 dimensions of **w**), **GrB_NULL** should be specified.

3510 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

3511 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 3512 specified.

3513 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
 3514 operation. Depending on which type is passed, the following defines the binary
 3515 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3516 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

3517 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-
 3518 nored.

3519 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-
 3520 nary op and additive identity are ignored.

3521 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 3522 operation.

3523 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 3524 operation.

3525 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 3526 should be specified. Non-default field/value pairs are listed as follows:
 3527

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

3529 Return Values

3530 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 3531 blocking mode, this indicates that the compatibility tests on di-
 3532 mensions and domains for the input arguments passed successfully.
 3533 Either way, output vector **w** is ready to be used in the next method
 3534 of the sequence.

3535 **GrB_PANIC** Unknown internal error.

3536 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 3537 GraphBLAS objects (input or output) is in an invalid state caused
 3538 by a previous execution error. Call `GrB_error()` to access any error
 3539 messages generated by the implementation.

3540 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

3541 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3542 a call to `new` (or `dup` for vector parameters).

3543 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

3544 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
3545 responding domains of the binary operator (`op`) or accumulation
3546 operator, or the mask's domain is not compatible with `bool` (in the
3547 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3548 Description

3549 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors: $w =$
3550 $u \oplus v$, or, if an optional binary accumulation operator (\odot) is provided, $w = w \odot (u \oplus v)$. Logically,
3551 this operation occurs in three steps:

3552 **Setup** The internal vectors and mask used in the computation are formed and their domains
3553 and dimensions are tested for compatibility.

3554 **Compute** The indicated computations are carried out.

3555 **Output** The result is written into the output vector, possibly under control of a mask.

3556 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3557 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3558 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3559 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3560 4. $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3561 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are
3562 tested for domain compatibility as follows:

- 3563 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3564 must be from one of the pre-defined types of Table ??.
- 3565 2. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3566 3. $\mathbf{D}(v)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3567 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3568 5. $\mathbf{D}(u)$ and $\mathbf{D}(v)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3569 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3570 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
3571 the accumulation operator.

3572 Two domains are compatible with each other if values from one domain can be cast to values in
 3573 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 3574 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3575 any compatibility rule above is violated, execution of GrB_eWiseAdd ends and the domain mismatch
 3576 error listed above is returned.

3577 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 3578 denotes copy):

- 3579 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3580 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3581 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 3582 (b) If `mask \neq GrB_NULL`,
 - 3583 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 3584 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 3585 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3586 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3587 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

3588 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3589 must hold:

- 3590 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$.

3591 If any compatibility rule above is violated, execution of GrB_eWiseAdd ends and the dimension
 3592 mismatch error listed above is returned.

3593 From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with
 3594 GrB_SUCCESS return code and defer any computation and/or execution error codes.

3595 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3596 We describe this in terms of two intermediate vectors:

- 3597 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “sum” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 3598 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3599 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
 3600 value of each of its elements is computed by:

$$\begin{aligned}
 3601 \quad t_i &= (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})) \\
 3602 \\
 3603 \quad t_i &= \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})))
 \end{aligned}$$

3604
3605

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3606

where the difference operator in the previous expressions refers to set difference.

3607

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

3608

- If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3609

- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

3610

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

3611

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 $\tilde{\mathbf{t}}$.

3612

3613

$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

3614

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3615

3616

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3617

3618

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

3619

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”.

3620

3621

3622

- 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,

3623

3624

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

3625

- 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:

3626

3627

3628

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

3629

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.

3630

3631

3632

3633

3634

4.3.5.2 eWiseAdd: Matrix variant

3635

Perform element-wise (general) addition on the elements of two matrices, producing a third matrix as result.

3636

3637 C Syntax

```

3638     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3639                          const GrB_Matrix Mask,
3640                          const GrB_BinaryOp accum,
3641                          const GrB_Semiring op,
3642                          const GrB_Matrix A,
3643                          const GrB_Matrix B,
3644                          const GrB_Descriptor desc);
3645
3646     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3647                          const GrB_Matrix Mask,
3648                          const GrB_BinaryOp accum,
3649                          const GrB_Monoid op,
3650                          const GrB_Matrix A,
3651                          const GrB_Matrix B,
3652                          const GrB_Descriptor desc);
3653
3654     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3655                          const GrB_Matrix Mask,
3656                          const GrB_BinaryOp accum,
3657                          const GrB_BinaryOp op,
3658                          const GrB_Matrix A,
3659                          const GrB_Matrix B,
3660                          const GrB_Descriptor desc);

```

3661 Parameters

3662 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3663 that may be accumulated with the result of the element-wise operation. On output,
3664 the matrix holds the results of the operation.

3665 **Mask** (IN) An optional “write” mask that controls which results from this operation are
3666 stored into the output matrix C. The mask dimensions must match those of the
3667 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
3668 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
3669 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
3670 dimensions of C), GrB_NULL should be specified.

3671 **accum** (IN) An optional binary operator used for accumulating entries into existing C
3672 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3673 specified.

3674 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
3675 operation. Depending on which type is passed, the following defines the binary
3676 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

3705 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3706 corresponding domains of the binary operator (op) or accumulation
 3707 operator, or the mask's domain is not compatible with `bool` (in the
 3708 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3709 Description

3710 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:
 3711 $C = A \oplus B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \oplus B)$.
 3712 Logically, this operation occurs in three steps:

3713 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3714 and dimensions are tested for compatibility.

3715 **Compute** The indicated computations are carried out.

3716 **Output** The result is written into the output matrix, possibly under control of a mask.

3717 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3718 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3719 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)
- 3720 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3721 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3722 The argument matrices, the “sum” operator (op), and the accumulation operator (if provided) are
 3723 tested for domain compatibility as follows:

- 3724 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3725 must be from one of the pre-defined types of Table ??.
- 3726 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 3727 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 3728 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3729 5. $\mathbf{D}(A)$ and $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 3730 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3731 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3732 the accumulation operator.

3733 Two domains are compatible with each other if values from one domain can be cast to values in
 3734 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 3735 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3736 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch
 3737 error listed above is returned.

3738 From the argument matrices, the internal matrices and mask used in the computation are formed
 3739 (\leftarrow denotes copy):

- 3740 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3741 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3742 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3743 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3744 (b) If `Mask \neq GrB_NULL`,
 - 3745 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3746 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3747 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3748 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 3749 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3750 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3751 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3752 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3753 must hold:

- 3754 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3755 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3756 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension
 3757 mismatch error listed above is returned.

3758 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3759 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3760 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3761 We describe this in terms of two intermediate matrices:

- 3762 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise sum of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3763 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3764 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$
 3765 is created. The value of each of its elements is computed by

$$\begin{aligned} 3766 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\ 3767 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\ 3768 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \end{aligned}$$

3771 where the difference operator in the previous expressions refers to set difference.

3772 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3773 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3774 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3775 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3776 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3777 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 3778 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3779 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3780 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \end{aligned}$$

3783 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3784 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 3785 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 3786 mask which acts as a “write mask”.

- 3787 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 3788 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3789 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3790 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 3791 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 3792 mask are unchanged:

$$3793 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3794 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3795 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3796 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 3797 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3798 sequence.

3799 4.3.6 extract: Selecting sub-graphs

3800 Extract a subset of a matrix or vector.

3801 4.3.6.1 extract: Standard vector variant

3802 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector
3803 whose size is equal to the number of indices.

3804 C Syntax

```
3805         GrB_Info GrB_extract(GrB_Vector          w,  
3806                             const GrB_Vector    mask,  
3807                             const GrB_BinaryOp    accum,  
3808                             const GrB_Vector    u,  
3809                             const GrB_Index      *indices,  
3810                             GrB_Index            nindices,  
3811                             const GrB_Descriptor desc);
```

3812 Parameters

3813 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3814 that may be accumulated with the result of the extract operation. On output, this
3815 vector holds the results of the operation.

3816 **mask** (IN) An optional “write” mask that controls which results from this operation are
3817 stored into the output vector **w**. The mask dimensions must match those of the
3818 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
3819 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
3820 in Table ???. If the default mask is desired (i.e., a mask that is all **true** with the
3821 dimensions of **w**), **GrB_NULL** should be specified.

3822 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3823 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
3824 specified.

3825 **u** (IN) The GraphBLAS vector from which the subset is extracted.

3826 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of
3827 elements from **u** that are extracted. If all elements of **u** are to be extracted in order
3828 from 0 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution
3829 mode and return value, this array may be manipulated by the caller after this
3830 operation returns without affecting any deferred computations for this operation.

3831 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

3832 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3833 should be specified. Non-default field/value pairs are listed as follows:

3834

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 .

3835

3836 Return Values

3837 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3838 blocking mode, this indicates that the compatibility tests on
 3839 dimensions and domains for the input arguments passed suc-
 3840 cessfully. Either way, output vector **w** is ready to be used in the
 3841 next method of the sequence.

3842 GrB_PANIC Unknown internal error.

3843 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
 3844 opaque GraphBLAS objects (input or output) is in an invalid
 3845 state caused by a previous execution error. Call GrB_error() to
 3846 access any error messages generated by the implementation.

3847 GrB_OUT_OF_MEMORY Not enough memory available for operation.

3848 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
 3849 by a call to **new** (or **dup** for vector parameters).

3850 GrB_INDEX_OUT_OF_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-
 3851 blocking mode, this error can be deferred.

3852 GrB_DIMENSION_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices** \neq **size(w)**.

3853 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each
 3854 other or the corresponding domains of the accumulation oper-
 3855 ator, or the mask's domain is not compatible with **bool** (in the
 3856 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

3857 GrB_NULL_POINTER Argument **row_indices** is a NULL pointer.

3858 Description

3859 This variant of GrB_extract computes the result of extracting a subset of locations from a Graph-
 3860 BLAS vector in a specific order: $w = u(\text{indices})$; or, if an optional binary accumulation operator

3861 (\odot) is provided, $w = w \odot u(\text{indices})$. More explicitly:

$$3862 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

3863 Logically, this operation occurs in three steps:

3864 **Setup** The internal vectors and mask used in the computation are formed and their domains
3865 and dimensions are tested for compatibility.

3866 **Compute** The indicated computations are carried out.

3867 **Output** The result is written into the output vector, possibly under control of a mask.

3868 Up to three argument vectors are used in this GrB_extract operation:

- 3869 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3870 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3871 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3872 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
3873 bility as follows:

- 3874 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3875 must be from one of the pre-defined types of Table ??.
- 3876 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(u)$.
- 3877 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3878 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
3879 mulation operator.

3880 Two domains are compatible with each other if values from one domain can be cast to values in
3881 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
3882 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3883 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
3884 error listed above is returned.

3885 From the arguments, the internal vectors, mask, and index array used in the computation are
3886 formed (\leftarrow denotes copy):

- 3887 1. Vector $\tilde{w} \leftarrow w$.
- 3888 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
3889 (a) If `mask` = `GrB_NULL`, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.

- 3890 (b) If $\text{mask} \neq \text{GrB_NULL}$,
 3891 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 3892 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 3893 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
 3894 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
 3895 4. The internal index array, $\widetilde{\mathbf{I}}$, is computed from argument indices as follows:
 3896 (a) If $\text{indices} = \text{GrB_ALL}$, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.
 3897 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

3898 The internal vectors and mask are checked for dimension compatibility. The following conditions
 3899 must hold:

- 3900 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
 3901 2. $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$.

3902 If any compatibility rule above is violated, execution of GrB_extract ends and the dimension mis-
 3903 match error listed above is returned.

3904 From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with
 3905 GrB_SUCCESS return code and defer any computation and/or execution error codes.

3906 We are now ready to carry out the extract and any additional associated operations. We describe
 3907 this in terms of two intermediate vectors:

- 3908 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{w}}$.
- 3909 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3910 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$3911 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

3912 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
 3913 GrB_extract ends and the index-out-of-bounds error listed above is generated. In GrB_NONBLOCKING
 3914 mode, the error can be deferred until a sequence-terminating $\text{GrB_wait}()$ is called. Regardless, the
 3915 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

3916 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3917 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 3918 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$3919 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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 $\tilde{\mathbf{t}}$.

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

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.

4.3.6.2 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 ???. If the default mask is desired (i.e., a mask that is all **true** with the dimensions of **C**), **GrB_NULL** should be specified.

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

A (IN) The GraphBLAS matrix from which the subset is extracted.

row_indices (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **A** from which elements are extracted. If elements in all rows of **A** are to be extracted in order, **GrB_ALL** should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

nrows (IN) The number of values in the **row_indices** array. Must be equal to **nrows(C)**.

col_indices (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **A** from which elements are extracted. If elements in all columns of **A** are to be extracted in order, then **GrB_ALL** should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

ncols (IN) The number of values in the **col_indices** array. Must be equal to **ncols(C)**.

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

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

3986 Return Values

3987	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
3988		blocking mode, this indicates that the compatibility tests on
3989		dimensions and domains for the input arguments passed suc-
3990		cessfully. Either way, output matrix C is ready to be used in the
3991		next method of the sequence.
3992	GrB_PANIC	Unknown internal error.
3993	GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the
3994		opaque GraphBLAS objects (input or output) is in an invalid
3995		state caused by a previous execution error. Call <code>GrB_error()</code> to
3996		access any error messages generated by the implementation.
3997	GrB_OUT_OF_MEMORY	Not enough memory available for the operation.
3998	GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized
3999		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4000	GrB_INDEX_OUT_OF_BOUNDS	A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or
4001		a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In
4002		non-blocking mode, this error can be deferred.
4003	GrB_DIMENSION_MISMATCH	Mask and C dimensions are incompatible, <code>nrows</code> \neq <code>nrows(C)</code> , or
4004		<code>ncols</code> \neq <code>ncols(C)</code> .
4005	GrB_DOMAIN_MISMATCH	The domains of the various matrices are incompatible with each
4006		other or the corresponding domains of the accumulation oper-
4007		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4008		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4009	GrB_NULL_POINTER	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4010		is a NULL pointer, or both.

4011 Description

4012 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified
 4013 rows and columns of a GraphBLAS matrix in a specific order: $C = A(\text{row_indices}, \text{col_indices})$; or,
 4014 if an optional binary accumulation operator (\odot) is provided, $C = C \odot A(\text{row_indices}, \text{col_indices})$.
 4015 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4017 Logically, this operation occurs in three steps:

4018 **Setup** The internal matrices and mask used in the computation are formed and their domains
 4019 and dimensions are tested for compatibility.

4020 **Compute** The indicated computations are carried out.

4021 **Output** The result is written into the output matrix, possibly under control of a mask.

4022 Up to three argument matrices are used in the `GrB_extract` operation:

- 4023 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
4024 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)
4025 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4026 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
4027 ibility as follows:

- 4028 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
4029 must be from one of the pre-defined types of Table ??.
4030 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
4031 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4032 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4033 mulation operator.

4034 Two domains are compatible with each other if values from one domain can be cast to values in
4035 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
4036 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4037 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
4038 error listed above is returned.

4039 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are
4040 formed (\leftarrow denotes copy):

- 4041 1. Matrix $\tilde{C} \leftarrow C$.
4042 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:
4043 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
4044 $j < \mathbf{ncols}(C)\} \rangle$.
4045 (b) If `Mask` \neq `GrB_NULL`,
4046 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
4047 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
4048 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
4049 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
4050 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
4051 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

- 4052 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4053 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 4054 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \text{nrows}$.
- 4055 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4056 (a) If `col_indices` = `GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$.
- 4057 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \text{ncols}$.

4058 The internal matrices and mask are checked for dimension compatibility. The following conditions
4059 must hold:

- 4060 1. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$.
- 4061 2. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$.
- 4062 3. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$.
- 4063 4. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$.

4064 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
4065 match error listed above is returned.

4066 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4067 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4068 We are now ready to carry out the extract and any additional associated operations. We describe
4069 this in terms of two intermediate matrices:

- 4070 • $\tilde{\mathbf{T}}$: The matrix holding the extraction from $\tilde{\mathbf{A}}$.
- 4071 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4072 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

4073
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{ (i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}}) \} \rangle.$$

4074 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 $\tilde{\mathbf{J}}$
4075 array is not in the range $[0, \text{ncols}(\tilde{\mathbf{A}}))$, the execution of `GrB_extract` ends and the index out-of-
4076 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
4077 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
4078 this point forward in the sequence.

4079 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4080 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

4081 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4082 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4083 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4084 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4085 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4086 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4088 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4090 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4091 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
4092 using what is called a *standard matrix mask and replace*. This is carried out under control of the
4093 mask which acts as a “write mask”.

4094 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
4095 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4096 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4097 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
4098 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
4099 mask are unchanged:

$$4100 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4101 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4102 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4103 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
4104 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4105 sequence.

4106 4.3.6.3 extract: Column (and row) variant

4107 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the
4108 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as
4109 well.

4110 C Syntax

```

4111         GrB_Info GrB_extract(GrB_Vector          w,
4112                               const GrB_Vector    mask,
4113                               const GrB_BinaryOp    accum,
4114                               const GrB_Matrix      A,
4115                               const GrB_Index       *row_indices,
4116                               GrB_Index            nrows,
4117                               GrB_Index            col_index,
4118                               const GrB_Descriptor   desc);

```

4119 Parameters

4120 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4121 that may be accumulated with the result of the extract operation. On output, this
4122 vector holds the results of the operation.

4123 **mask** (IN) An optional “write” mask that controls which results from this operation are
4124 stored into the output vector **w**. The mask dimensions must match those of the
4125 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4126 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4127 in Table ???. If the default mask is desired (i.e., a mask that is all **true** with the
4128 dimensions of **w**), **GrB_NULL** should be specified.

4129 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4130 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4131 specified.

4132 **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4133 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations
4134 within the specified column of **A** from which elements are extracted. If elements in
4135 all rows of **A** are to be extracted in order, **GrB_ALL** should be specified. Regardless
4136 of execution mode and return value, this array may be manipulated by the caller
4137 after this operation returns without affecting any deferred computations for this
4138 operation.

4139 **nrows** (IN) The number of indices in the **row_indices** array. Must be equal to **size(w)**.

4140 **col_index** (IN) The index of the column of **A** from which to extract values. It must be in the
4141 range $[0, \mathbf{ncols}(A))$.

4142 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4143 should be specified. Non-default field/value pairs are listed as follows:

4144

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

Return Values

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

GrB_PANIC Unknown internal error.

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

GrB_OUT_OF_MEMORY Not enough memory available for operation.

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

GrB_INVALID_INDEX **col_index** is outside the allowable range (i.e., greater than **ncols(A)**).

GrB_INDEX_OUT_OF_BOUNDS A value in **row_indices** is greater than or equal to **nrows(A)**. In non-blocking mode, this error can be deferred.

GrB_DIMENSION_MISMATCH **mask** and **w** dimensions are incompatible, or **nrows** \neq **size(w)**.

GrB_DOMAIN_MISMATCH The domains of the vector or matrix are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB_MASK].GrB_STRUCTURE** is not set).

GrB_NULL_POINTER Argument **row_indices** is a NULL pointer.

Description

This variant of **GrB_extract** computes the result of extracting a subset of locations (in a specific order) from a specified column of a GraphBLAS matrix: **w** = **A(:, col_index)(row_indices)**; or, if

4172 an optional binary accumulation operator (\odot) is provided, $w = w \odot A(:, \text{col_index})(\text{row_indices})$.
 4173 More explicitly:

$$4174 \quad \begin{aligned} w(i) &= A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}, \text{ or} \\ w(i) &= w(i) \odot A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4175 Logically, this operation occurs in three steps:

4176 **Setup** The internal matrices, vectors, and mask used in the computation are formed and their
 4177 domains and dimensions are tested for compatibility.

4178 **Compute** The indicated computations are carried out.

4179 **Output** The result is written into the output vector, possibly under control of a mask.

4180 Up to three argument vectors and matrices are used in this GrB_extract operation:

- 4181 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4182 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4183 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4184 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain
 4185 compatibility as follows:

- 4186 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
 4187 must be from one of the pre-defined types of Table ??.
- 4188 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(A)$.
- 4189 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4190 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4191 mulation operator.

4192 Two domains are compatible with each other if values from one domain can be cast to values in
 4193 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 4194 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4195 any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch
 4196 error listed above is returned.

4197 From the arguments, the internal vector, matrix, mask, and index array used in the computation
 4198 are formed (\leftarrow denotes copy):

- 4199 1. Vector $\tilde{w} \leftarrow w$.
- 4200 2. One-dimensional mask, \tilde{m} , is computed from argument **mask** as follows:
 4201 (a) If **mask** = GrB_NULL, then $\tilde{m} = \langle \mathbf{size}(w), \{i, \forall i : 0 \leq i < \mathbf{size}(w)\} \rangle$.

- 4202 (b) If $\text{mask} \neq \text{GrB_NULL}$,
- 4203 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
- 4204 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 4205 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
- 4206 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 4207 4. The internal row index array, $\widetilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4208 (a) If `indices = GrB_ALL`, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 4209 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$.

4210 The internal vector, `mask`, and index array are checked for dimension compatibility. The following
 4211 conditions must hold:

- 4212 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4213 2. $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$.

4214 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
 4215 match error listed above is returned.

4216 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4217 1. $0 \leq \text{col_index} < \text{ncols}(\mathbf{A})$

4218 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above
 4219 is returned.

4220 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4221 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4222 We are now ready to carry out the extract and any additional associated operations. We describe
 4223 this in terms of two intermediate vectors:

- 4224 • $\widetilde{\mathbf{t}}$: The vector holding the extraction from a column of $\widetilde{\mathbf{A}}$.
- 4225 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4226 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

4227
$$\widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4228 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`
 4229 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,
 4230 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result
 4231 vector, \mathbf{w} , is invalid from this point forward in the sequence.

4232 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

4233 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

4234 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4235 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4236 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4237 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4238 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

$$4239 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4240 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4241 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4242 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 4243 using what is called a *standard vector mask and replace*. This is carried out under control of the
 4244 mask which acts as a “write mask”.

4245 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 4246 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4247 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4250 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 4251 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 4252 mask are unchanged:

$$4253 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4254 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4255 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4256 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
 4257 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4258 sequence.

4259 4.3.7 assign: Modifying sub-graphs

4260 Assign the contents of a subset of a matrix or vector.

4261 4.3.7.1 assign: Standard vector variant

4262 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.
 4263 The size of the input vector is the same size as the index array provided.

4264 C Syntax

```
4265         GrB_Info GrB_assign(GrB_Vector      w,  
4266                             const GrB_Vector mask,  
4267                             const GrB_BinaryOp accum,  
4268                             const GrB_Vector u,  
4269                             const GrB_Index *indices,  
4270                             GrB_Index      nindices,  
4271                             const GrB_Descriptor desc);
```

4272 Parameters

4273 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4274 that may be accumulated with the result of the assign operation. On output, this
4275 vector holds the results of the operation.

4276 **mask** (IN) An optional “write” mask that controls which results from this operation are
4277 stored into the output vector **w**. The mask dimensions must match those of the
4278 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4279 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4280 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
4281 dimensions of **w**), **GrB_NULL** should be specified.

4282 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4283 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4284 specified.

4285 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4286 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4287 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0
4288 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4289 and return value, this array may be manipulated by the caller after this operation
4290 returns without affecting any deferred computations for this operation. If this
4291 array contains duplicate values, it implies in assignment of more than one value to
4292 the same location which leads to undefined results.

4293 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4294 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4295 should be specified. Non-default field/value pairs are listed as follows:
4296

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:

$$\begin{aligned}
 w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
 \end{aligned}$$

4325 Logically, this operation occurs in three steps:

4326 **Setup** The internal vectors and mask used in the computation are formed and their domains
4327 and dimensions are tested for compatibility.

4328 **Compute** The indicated computations are carried out.

4329 **Output** The result is written into the output vector, possibly under control of a mask.

4330 Up to three argument vectors are used in the `GrB_assign` operation:

- 4331 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4332 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4333 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4334 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4335 bility as follows:

- 4336 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
4337 must be from one of the pre-defined types of Table ??.
- 4338 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 4339 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
4340 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
4341 mulation operator.

4342 Two domains are compatible with each other if values from one domain can be cast to values in
4343 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
4344 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4345 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4346 error listed above is returned.

4347 From the arguments, the internal vectors, mask and index array used in the computation are formed
4348 (\leftarrow denotes copy):

- 4349 1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 4350 2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 4351 (a) If `mask` = `GrB_NULL`, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 4352 (b) If `mask` \neq `GrB_NULL`,
 - 4353 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 4354 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 4355 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.

4356 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4357 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:

4358 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i$, $\forall i : 0 \leq i < \text{nindices}$.

4359 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i]$, $\forall i : 0 \leq i < \text{nindices}$.

4360 The internal vector and mask are checked for dimension compatibility. The following conditions
4361 must hold:

4362 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4363 2. $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$.

4364 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4365 match error listed above is returned.

4366 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4367 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4368 We are now ready to carry out the assign and any additional associated operations. We describe
4369 this in terms of two intermediate vectors:

- 4370 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.
- 4371 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4372 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4373 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4374 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{w}}$, computation
4375 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`
4376 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4377 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

4378 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4379 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4380 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4381 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4382 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
4383 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4384 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4385 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4386 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

$$4387 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4389 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,
```

```

4423         const GrB_Index      *row_indices,
4424         GrB_Index             nrows,
4425         const GrB_Index      *col_indices,
4426         GrB_Index             ncols,
4427         const GrB_Descriptor desc);

```

4428 Parameters

4429 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
4430 that may be accumulated with the result of the assign operation. On output, the
4431 matrix holds the results of the operation.

4432 **Mask** (IN) An optional “write” mask that controls which results from this operation are
4433 stored into the output matrix **C**. The mask dimensions must match those of the
4434 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
4435 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
4436 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
4437 dimensions of **C**), **GrB_NULL** should be specified.

4438 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4439 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4440 specified.

4441 **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4442 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
4443 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,
4444 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
4445 this array may be manipulated by the caller after this operation returns without
4446 affecting any deferred computations for this operation. If this array contains du-
4447 plicate values, it implies assignment of more than one value to the same location
4448 which leads to undefined results.

4449 **nrows** (IN) The number of values in the **row_indices** array. Must be equal to **nrows(A)**
4450 if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4451 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns
4452 of **C** that are assigned. If all columns of **C** are to be assigned in order from 0
4453 to **ncols** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
4454 and return value, this array may be manipulated by the caller after this operation
4455 returns without affecting any deferred computations for this operation. If this
4456 array contains duplicate values, it implies assignment of more than one value to
4457 the same location which leads to undefined results.

4458 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **ncols(A)** if **A** is
4459 not transposed, or equal to **nrows(A)** if **A** is transposed.

4460
4461
4462

4463

4464

4465
4466
4467
4468
4469

4470

4471
4472
4473
4474

4475

4476
4477

4478
4479
4480

4481
4482

4483
4484
4485
4486

4487
4488

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

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

Return Values

- GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.
- GrB_PANIC Unknown internal error.
- GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
- GrB_OUT_OF_MEMORY Not enough memory available for the operation.
- GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).
- GrB_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.
- GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrows \neq nrows(A), or ncols \neq ncols(A).
- 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).
- GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices is a NULL pointer, or both.

4489 Description

4490 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows
 4491 and columns in `C` in a specified order: $C(\text{row_indices}, \text{col_indices}) = A$; or, if an optional binary
 4492 accumulation operator (\odot) is provided, $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot$
 4493 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4494 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot A(i, j), \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4495 Logically, this operation occurs in three steps:

4496 Setup The internal matrices and mask used in the computation are formed and their domains
 4497 and dimensions are tested for compatibility.

4498 Compute The indicated computations are carried out.

4499 Output The result is written into the output matrix, possibly under control of a mask.

4500 Up to three argument matrices are used in the `GrB_assign` operation:

- 4501 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4502 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)
- 4503 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4504 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
 4505 ibility as follows:

- 4506 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 4507 must be from one of the pre-defined types of Table ??.
- 4508 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 4509 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4510 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4511 mulation operator.

4512 Two domains are compatible with each other if values from one domain can be cast to values in
 4513 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 4514 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4515 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4516 error listed above is returned.

4517 From the arguments, the internal matrices, mask, and index arrays used in the computation are
 4518 formed (\leftarrow denotes copy):

- 4519 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4520 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
- 4521 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
4522 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 4523 (b) If `Mask \neq GrB_NULL`,
- 4524 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
4525 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
- 4526 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
4527 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
- 4528 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4529 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 4530 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4531 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
- 4532 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 4533 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4534 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
- 4535 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4536 The internal matrices and mask are checked for dimension compatibility. The following conditions
4537 must hold:

- 4538 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 4539 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 4540 3. $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$.
- 4541 4. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$.

4542 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4543 match error listed above is returned.

4544 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4545 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4546 We are now ready to carry out the assign and any additional associated operations. We describe
4547 this in terms of two intermediate vectors:

- 4548 • $\tilde{\mathbf{T}}$: The matrix holding the contents from $\tilde{\mathbf{A}}$ in their destination locations relative to $\tilde{\mathbf{C}}$.
- 4549 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4550 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4551 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4552 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
 4553 $\tilde{\mathbf{J}}$ array is not in the range $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-
 4554 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
 4555 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
 4556 this point forward in the sequence.

4557 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 4558 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$4559 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4560 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4561 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 4562 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
 4563 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

4564 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4565 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4566 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4567 \\ 4568 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4569 where the difference operator refers to set difference.

- 4570 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4571 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4572 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4573 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4574 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4575 \\ 4576 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4577 \\ 4578 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4579 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4580 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4581 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4582 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

4616 bool or any of the predefined “built-in” types in Table ?? . If the default mask
 4617 is desired (i.e., a mask that is all true with the dimensions of a column of C),
 4618 GrB_NULL should be specified.

4619 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 4620 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 4621 specified.

4622 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column
 4623 of C.

4624 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
 4625 the specified column of C that are to be assigned. If all elements of the column
 4626 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB_ALL should
 4627 be specified. Regardless of execution mode and return value, this array may be
 4628 manipulated by the caller after this operation returns without affecting any de-
 4629 ferred computations for this operation. If this array contains duplicate values, it
 4630 implies in assignment of more than one value to the same location which leads to
 4631 undefined results.

4632 **nrows** (IN) The number of values in **row_indices** array. Must be equal to **size(u)**.

4633 **col_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4634 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 4635 should be specified. Non-default field/value pairs are listed as follows:

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

4638 Return Values

4639 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 4640 blocking mode, this indicates that the compatibility tests on
 4641 dimensions and domains for the input arguments passed suc-
 4642 cessfully. Either way, output matrix C is ready to be used in the
 4643 next method of the sequence.

4644 **GrB_PANIC** Unknown internal error.

4645 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
4646 opaque GraphBLAS objects (input or output) is in an invalid
4647 state caused by a previous execution error. Call `GrB_error()` to
4648 access any error messages generated by the implementation.

4649 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4650 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
4651 by a call to `new` (or `dup` for vector or matrix parameters).

4652 GrB_INVALID_INDEX `col_index` is outside the allowable range (i.e., greater than `ncols(C)`).

4653 GrB_INDEX_OUT_OF_BOUNDS A value in `row_indices` is greater than or equal to `nrows(C)`. In
4654 non-blocking mode, this can be reported as an execution error.

4655 GrB_DIMENSION_MISMATCH `mask` size and number of rows in `C` are not the same, or `nrows` \neq
4656 `size(u)`.

4657 GrB_DOMAIN_MISMATCH The domains of the matrix and vector are incompatible with
4658 each other or the corresponding domains of the accumulation
4659 operator, or the mask's domain is not compatible with `bool` (in
4660 the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4661 GrB_NULL_POINTER Argument `row_indices` is a NULL pointer.

4662 Description

4663 This variant of `GrB_assign` computes the result of assigning a subset of locations in a column of a
4664 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:
4665 $C(:, col_index) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(:, col_index) =$
4666 $C(:, col_index) \odot u$. Taking order of `row_indices` into account, it is more explicitly written as:

$$4667 \quad C(row_indices[i], col_index) = u(i), \forall i : 0 \leq i < nrows, \text{ or}$$

$$C(row_indices[i], col_index) = C(row_indices[i], col_index) \odot u(i), \forall i : 0 \leq i < nrows.$$

4668 Logically, this operation occurs in three steps:

4669 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
4670 domains and dimensions are tested for compatibility.

4671 **Compute** The indicated computations are carried out.

4672 **Output** The result is written into the output matrix, possibly under control of a mask.

4673 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 4674 1. $C = \langle D(C), nrows(C), ncols(C), L(C) = \{(i, j, C_{ij})\} \rangle$
- 4675 2. $mask = \langle D(mask), size(mask), L(mask) = \{(i, m_i)\} \rangle$ (optional)

4676 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4677 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
4678 compatibility as follows:

- 4679 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4680 must be from one of the pre-defined types of Table ??.
- 4681 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 4682 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4683 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4684 mulation operator.

4685 Two domains are compatible with each other if values from one domain can be cast to values in
4686 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
4687 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4688 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4689 error listed above is returned.

4690 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4691 1. $0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{C})$

4692 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
4693 is returned.

4694 From the arguments, the internal vectors, `mask`, and index array used in the computation are
4695 formed (\leftarrow denotes copy):

- 4696 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a column of \mathbf{C} as follows:

$$4697 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

- 4698 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

- 4699 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$.
- 4700 (b) If `mask` \neq `GrB_NULL`,
 - 4701 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,
 - 4702 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 4703 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

- 4704 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

- 4705 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:

- 4706 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.

4707 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i]$, $\forall i : 0 \leq i < \text{nrows}$.

4708 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
4709 conditions must hold:

- 4710 1. $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4711 2. $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$.

4712 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4713 match error listed above is returned.

4714 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4715 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4716 We are now ready to carry out the assign and any additional associated operations. We describe
4717 this in terms of two intermediate vectors:

- 4718 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 4719 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4720 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4721 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4722 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
4723 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
4724 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4725 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4726 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4727 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4728 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4729 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
4730 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
4731 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4732 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4733 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$4734 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4735 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4737 where the difference operator refers to set difference.

4738 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4739 \quad (\mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\}).$$

4740 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4741 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4742 \quad z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$4743 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$4744 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

4747 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4748 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
4749 result matrix, $\mathbf{C}(:, \text{col_index})$. This is carried out under control of the mask which acts as a “write
4750 mask”.

4751 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(:, \text{col_index})$ on input to this
4752 operation are deleted and the new contents of the column is given by:

$$4753 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4754 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4755 copied into the column of the final result matrix, $\mathbf{C}(:, \text{col_index})$, and elements of this column
4756 that fall outside the set indicated by the mask are unchanged:

$$4757 \quad \begin{aligned} \mathbf{L}(\mathbf{C}) = & \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \\ & \{(i, \text{col_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ & \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

4760 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4761 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4762 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may
4763 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

4764 4.3.7.4 assign: Row variant

4765 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the
4766 output cannot be transposed, a different variant of `assign` is provided to assign to a column of a
4767 matrix.

4768 C Syntax

```
4769         GrB_Info GrB_assign(GrB_Matrix      C,  
4770                             const GrB_Vector mask,  
4771                             const GrB_BinaryOp accum,  
4772                             const GrB_Vector u,  
4773                             GrB_Index      row_index,  
4774                             const GrB_Index *col_indices,  
4775                             GrB_Index      ncols,  
4776                             const GrB_Descriptor desc);
```

4777 Parameters

4778 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values
4779 that may be accumulated with the result of the assign operation. On output, this
4780 matrix holds the results of the operation.

4781 **mask** (IN) An optional “write” mask that controls which results from this operation are
4782 stored into the specified row of the output matrix **C**. The mask dimensions must
4783 match those of a single row of the matrix **C**. If the **GrB_STRUCTURE** descriptor
4784 is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or
4785 any of the predefined “built-in” types in Table ???. If the default mask is desired
4786 (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB_NULL** should
4787 be specified.

4788 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4789 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4790 specified.

4791 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of
4792 **C**.

4793 **row_index** (IN) The index of the row in **C** to assign. Must be in the range $[0, \mathbf{nrows}(\mathbf{C})]$.

4794 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4795 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to
4796 be assigned in order from index 0 to $\mathbf{ncols} - 1$, then **GrB_ALL** should be specified.
4797 Regardless of execution mode and return value, this array may be manipulated by
4798 the caller after this operation returns without affecting any deferred computations
4799 for this operation. If this array contains duplicate values, it implies in assignment
4800 of more than one value to the same location which leads to undefined results.

4801 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **size(u)**.

4802 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4803 should be specified. Non-default field/value pairs are listed as follows:
4804

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

Return Values

GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
GrB_OUT_OF_MEMORY	Not enough memory available for operation.
GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector or matrix parameters).
GrB_INVALID_INDEX	row_index is outside the allowable range (i.e., greater than nrows(C)).
GrB_INDEX_OUT_OF_BOUNDS	A value in col_indices is greater than or equal to ncols(C) . In non-blocking mode, this can be reported as an execution error.
GrB_DIMENSION_MISMATCH	mask size and number of columns in C are not the same, or ncols \neq size(u) .
GrB_DOMAIN_MISMATCH	The domains of the matrix and vector are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB_MASK].GrB_STRUCTURE is not set).
GrB_NULL_POINTER	Argument col_indices is a NULL pointer.

Description

This variant of **GrB_assign** computes the result of assigning a subset of locations in a row of a GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

4833 $C(\text{row_index}, :) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(\text{row_index}, :$
 4834 $) = C(\text{row_index}, :) \odot u$. Taking order of `col_indices` into account it is more explicitly written as:

$$4835 \quad C(\text{row_index}, \text{col_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or} \\ C(\text{row_index}, \text{col_indices}[j]) = C(\text{row_index}, \text{col_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$$

4836 Logically, this operation occurs in three steps:

4837 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
 4838 domains and dimensions are tested for compatibility.

4839 **Compute** The indicated computations are carried out.

4840 **Output** The result is written into the output matrix, possibly under control of a mask.

4841 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 4842 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4843 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4844 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4845 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
 4846 compatibility as follows:

- 4847 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 4848 must be from one of the pre-defined types of Table ??.
- 4849 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(u)$.
- 4850 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4851 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4852 mulation operator.

4853 Two domains are compatible with each other if values from one domain can be cast to values in
 4854 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 4855 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4856 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4857 error listed above is returned.

4858 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 4859 1. $0 \leq \text{row_index} < \mathbf{nrows}(C)$

4860 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
 4861 is returned.

4862 From the arguments, the internal vectors, mask, and index array used in the computation are
 4863 formed (\leftarrow denotes copy):

4864 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a row of \mathbf{C} as follows:

$$4865 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4866 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

4867 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$.

4868 (b) If `mask \neq GrB_NULL`,

4869 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

4870 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

4871 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

4872 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4873 4. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:

4874 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.

4875 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4876 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
4877 conditions must hold:

4878 1. $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

4879 2. $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$.

4880 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-
4881 match error listed above is returned.

4882 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4883 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4884 We are now ready to carry out the assign and any additional associated operations. We describe
4885 this in terms of two intermediate vectors:

- 4886 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 4887 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4888 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4889 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4890 At this point, if any value of $\tilde{\mathbf{J}}[j]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
4891 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
4892 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4893 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4894 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure 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 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

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 $\tilde{\mathbf{t}}$.

$$z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))), \\ z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

where the difference operator refers to set difference.

- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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

$$z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})), \\ z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))), \\ z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

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, $\mathbf{C}(\text{row_index}, :)$. 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 $\mathbf{C}(\text{row_index}, :)$ on input to this operation are deleted and the new contents of the column is given by:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 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, $\mathbf{C}(\text{row_index}, :)$, and elements of this column that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup \\ \{(\text{row_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of 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.

4932 4.3.7.5 assign: Constant vector variant[Scott: NEW CONTENT]

4933 Assign the same value to a specified subset of vector elements. With the use of GrB_ALL, the entire
4934 destination vector can be filled with the constant.

4935 C Syntax

```
4936      GrB_Info GrB_assign(GrB_Vector      w,  
4937                          const GrB_Vector mask,  
4938                          const GrB_BinaryOp accum,  
4939                          <type>          val,  
4940                          const GrB_Index *indices,  
4941                          GrB_Index      nindices,  
4942                          const GrB_Descriptor desc);
```

```
4943      GrB_Info GrB_assign(GrB_Vector      w,  
4944                          const GrB_Vector mask,  
4945                          const GrB_BinaryOp accum,  
4946                          const GrB_Scalar s,  
4947                          const GrB_Index *indices,  
4948                          GrB_Index      nindices,  
4949                          const GrB_Descriptor desc);
```

4950 Parameters

4951 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4952 that may be accumulated with the result of the assign operation. On output, this
4953 vector holds the results of the operation.

4954 **mask** (IN) An optional “write” mask that controls which results from this operation are
4955 stored into the output vector **w**. The mask dimensions must match those of the
4956 vector **w**. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
4957 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
4958 in Table ???. If the default mask is desired (i.e., a mask that is all **true** with the
4959 dimensions of **w**), GrB_NULL should be specified.

4960 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4961 entries. If assignment rather than accumulation is desired, GrB_NULL should be
4962 specified.

4963 **val** (IN) Scalar value to assign to (a subset of) **w**.

4964 **s** (IN) Scalar value to assign to (a subset of) **w**.

4965 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4966 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0

4967 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode
 4968 and return value, this array may be manipulated by the caller after this operation
 4969 returns without affecting any deferred computations for this operation. In this
 4970 variant, the specific order of the values in the array has no effect on the result.
 4971 Unlike other variants, if there are duplicated values in this array the result is still
 4972 defined.

4973 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If
 4974 `nindices` is zero, the operation becomes a NO-OP.

4975 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 4976 should be specified. Non-default field/value pairs are listed as follows:

4977

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> .

4978

4979 Return Values

4980 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 4981 blocking mode, this indicates that the compatibility tests on
 4982 dimensions and domains for the input arguments passed suc-
 4983 cessfully. Either way, output vector `w` is ready to be used in the
 4984 next method of the sequence.

4985 **GrB_PANIC** Unknown internal error.

4986 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
 4987 opaque GraphBLAS objects (input or output) is in an invalid
 4988 state caused by a previous execution error. Call `GrB_error()` to
 4989 access any error messages generated by the implementation.

4990 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

4991 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized
 4992 by a call to `new` (or `dup` for vector parameters).

4993 **GrB_INDEX_OUT_OF_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-
 4994 blocking mode, this can be reported as an execution error.

4995 **GrB_DIMENSION_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less
 4996 than `size(w)`.

5027 4. If **accum** is not **GrB_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the
 5028 method, must be compatible with **D_{in2}(accum)** of the accumulation operator.

5029 Two domains are compatible with each other if values from one domain can be cast to values in
 5030 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 5031 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5032 any compatibility rule above is violated, execution of **GrB_assign** ends and the domain mismatch
 5033 error listed above is returned.

5034 From the arguments, the internal vectors, mask and index array used in the computation are formed
 5035 (\leftarrow denotes copy):

- 5036 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5037 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument **mask** as follows:
 - 5038 (a) If **mask** = **GrB_NULL**, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 5039 (b) If **mask** \neq **GrB_NULL**,
 - 5040 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 5041 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$.
 - 5042 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5043 3. Scalar $\tilde{s} \leftarrow s$ (**GrB_Scalar** version only).
- 5044 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument **indices** as follows:
 - 5045 (a) If **indices** = **GrB_ALL**, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$.
 - 5046 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$.

5047 The internal vector and mask are checked for dimension compatibility. The following conditions
 5048 must hold:

- 5049 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5050 2. $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$.

5051 If any compatibility rule above is violated, execution of **GrB_assign** ends and the dimension mis-
 5052 match error listed above is returned.

5053 From this point forward, in **GrB_NONBLOCKING** mode, the method can optionally exit with
 5054 **GrB_SUCCESS** return code and defer any computation and/or execution error codes.

5055 We are now ready to carry out the assign and any additional associated operations. We describe
 5056 this in terms of two intermediate vectors:

- 5057 • $\tilde{\mathbf{t}}$: The vector holding the copies of the scalar, either **val** or \tilde{s} , in their destination locations
 5058 relative to $\tilde{\mathbf{w}}$.

- $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows. If a non-opaque scalar \mathbf{val} is provided:

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 1$):

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., $\mathbf{size}(\tilde{s}) = 0$):

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

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 is not in the range $[0, \mathbf{size}(\tilde{\mathbf{w}}))$, the execution of `GrB_assign` ends and the index out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result vector, \mathbf{w} , is invalid from this point forward in the sequence.

The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure 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 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

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 $\tilde{\mathbf{t}}$.

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

where the difference operator refers to set difference. We note that in this case of assigning a constant, $\{\tilde{\mathbf{I}}[k], \forall k\}$ and $\mathbf{ind}(\tilde{\mathbf{t}})$ are identical.

- If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

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

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 $\tilde{\mathbf{t}}$.

$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

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

5094 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 5095 using what is called a *standard vector mask and replace*. This is carried out under control of the
 5096 mask which acts as a “write mask”.

- 5097 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 5098 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$5099 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5100 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 5101 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 5102 mask are unchanged:

$$5103 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5104 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 5105 of vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 5106 exits with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but
 5107 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 5108 sequence.

5109 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5110 Assign the same value to a specified subset of matrix elements. With the use of GrB_ALL, the
 5111 entire destination matrix can be filled with the constant.

5112 C Syntax

```
5113     GrB_Info GrB_assign(GrB_Matrix      C,
5114                        const GrB_Matrix Mask,
5115                        const GrB_BinaryOp accum,
5116                        <type>          val,
5117                        const GrB_Index  *row_indices,
5118                        GrB_Index        nrows,
5119                        const GrB_Index  *col_indices,
5120                        GrB_Index        ncols,
5121                        const GrB_Descriptor desc);
```

```
5122     GrB_Info GrB_assign(GrB_Matrix      C,
5123                        const GrB_Matrix Mask,
5124                        const GrB_BinaryOp accum,
5125                        const GrB_Scalar  s,
5126                        const GrB_Index  *row_indices,
5127                        GrB_Index        nrows,
```

```

5128         const GrB_Index      *col_indices,
5129         GrB_Index             ncols,
5130         const GrB_Descriptor  desc);

```

5131 Parameters

5132 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5133 that may be accumulated with the result of the assign operation. On output, the
5134 matrix holds the results of the operation.

5135 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5136 stored into the output matrix **C**. The mask dimensions must match those of the
5137 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5138 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
5139 in Table ???. If the default mask is desired (i.e., a mask that is all **true** with the
5140 dimensions of **C**), **GrB_NULL** should be specified.

5141 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
5142 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5143 specified.

5144 **val** (IN) Scalar value to assign to (a subset of) **C**.

5145 **s** (IN) Scalar value to assign to (a subset of) **C**.

5146 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
5147 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** − 1,
5148 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
5149 this array may be manipulated by the caller after this operation returns without
5150 affecting any deferred computations for this operation. Unlike other variants, if
5151 there are duplicated values in this array the result is still defined.

5152 **nrows** (IN) The number of values in **row_indices** array. Must be in the range: [0, **nrows**(**C**)].
5153 If **nrows** is zero, the operation becomes a NO-OP.

5154 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**
5155 that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** − 1,
5156 then **GrB_ALL** should be specified. Regardless of execution mode and return value,
5157 this array may be manipulated by the caller after this operation returns without
5158 affecting any deferred computations for this operation. Unlike other variants, if
5159 there are duplicated values in this array the result is still defined.

5160 **ncols** (IN) The number of values in **col_indices** array. Must be in the range: [0, **ncols**(**C**)].
5161 If **ncols** is zero, the operation becomes a NO-OP.

5162 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5163 should be specified. Non-default field/value pairs are listed as follows:
5164

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`

5194 or $C(\text{row_indices}, \text{col_indices}) = s$ is performed. If an optional binary accumulation operator
 5195 (\odot) is provided, then either $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot \text{val}$ or
 5196 $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot s$ is performed. More explicitly, if a
 5197 non-opaque value val is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = \text{val}, \text{ or} \\ 5198 \quad & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5199 Correspondingly, if a `GrB_Scalar` s is provided:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = s, \text{ or} \\ 5200 \quad & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5201 Logically, this operation occurs in three steps:

5202 Setup The internal vectors and mask used in the computation are formed and their domains
 5203 and dimensions are tested for compatibility.

5204 Compute The indicated computations are carried out.

5205 Output The result is written into the output matrix, possibly under control of a mask.

5206 Up to two argument matrices are used in the `GrB_assign` operation:

- 5207 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5208 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)

5209 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain
 5210 compatibility as follows:

- 5211 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 5212 must be from one of the pre-defined types of Table ??.
- 5213 2. $\mathbf{D}(C)$ must be compatible with either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5214 method.
- 5215 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5216 of the accumulation operator.
- 5217 4. If `accum` is not `GrB_NULL`, then either $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the
 5218 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 ?? 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 $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - (c) If desc[GrB_MASK].GrB_COMP is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
3. Scalar $\tilde{s} \leftarrow s$ (GrB_Scalar version only).
4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument row_indices as follows:
 - (a) If row_indices = GrB_ALL, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
 - (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument col_indices as follows:
 - (a) If col_indices = GrB_ALL, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
 - (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

The internal matrix and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
3. $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$.
4. $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$.

If any compatibility rule above is violated, execution of GrB_assign ends and the dimension mismatch error listed above is returned.

5251 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5252 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5253 We are now ready to carry out the assign and any additional associated operations. We describe
 5254 this in terms of two intermediate matrices:

- 5255 • $\tilde{\mathbf{T}}$: The matrix holding the copies of the scalar, either `val` or \tilde{s} , in their destination locations
 5256 relative to $\tilde{\mathbf{C}}$.
- 5257 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5258 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows. If a non-opaque scalar `val` is provided:

$$5259 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5260 Correspondingly, if a non-empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 1):

$$5261 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5262 Finally, if an empty `GrB_Scalar` \tilde{s} is provided (i.e., `size`(\tilde{s}) = 0):

$$5263 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5264 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
 5265 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
 5266 $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-bounds error listed above is
 5267 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating
 5268 `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from this point forward in the
 5269 sequence.

5270 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 5271 • If `accum` = `GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$5272 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5273 \quad \{(i, j, Z_{ij}) \mid (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5274 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 5275 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
 5276 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

5277 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 5278 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5279 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5280 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \\ 5281$$

5282 where the difference operator refers to set difference. We note that, in this particular case of
 5283 assigning a constant to a matrix, the sets $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\}$ and $\mathbf{ind}(\tilde{\mathbf{T}})$ are identical.

- If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

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

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.

4.3.8 apply: Apply a function to the elements of an object

Computes the transformation of the values of the elements of a vector or a matrix using a unary function, or a binary function where one argument is bound to a scalar.

4.3.8.1 apply: Vector variant

Computes the transformation of the values of the elements of a vector using a unary function.

5314 C Syntax

```

5315      GrB_Info GrB_apply(GrB_Vector      w,
5316                        const GrB_Vector  mask,
5317                        const GrB_BinaryOp accum,
5318                        const GrB_UnaryOp  op,
5319                        const GrB_Vector  u,
5320                        const GrB_Descriptor desc);

```

5321 Parameters

5322 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5323 that may be accumulated with the result of the apply operation. On output, this
5324 vector holds the results of the operation.

5325 **mask** (IN) An optional “write” mask that controls which results from this operation are
5326 stored into the output vector **w**. The mask dimensions must match those of the
5327 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5328 of the mask vector must be of type **bool** or any of the predefined “built-in” types
5329 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
5330 dimensions of **w**), **GrB_NULL** should be specified.

5331 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5332 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5333 specified.

5334 **op** (IN) A unary operator applied to each element of input vector **u**.

5335 **u** (IN) The GraphBLAS vector to which the unary function is applied.

5336 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5337 should be specified. Non-default field/value pairs are listed as follows:

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

5340 Return Values

5341 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
5342 blocking mode, this indicates that the compatibility tests on di-
5343 mensions and domains for the input arguments passed successfully.

5344 Either way, output vector w is ready to be used in the next method
 5345 of the sequence.

5346 **GrB_PANIC** Unknown internal error.

5347 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 5348 GraphBLAS objects (input or output) is in an invalid state caused
 5349 by a previous execution error. Call **GrB_error()** to access any error
 5350 messages generated by the implementation.

5351 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

5352 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 5353 a call to **new** (or **dup** for vector parameters).

5354 **GrB_DIMENSION_MISMATCH** $mask$, w and/or u dimensions are incompatible.

5355 **GrB_DOMAIN_MISMATCH** The domains of the various vectors are incompatible with the corre-
 5356 sponding domains of the accumulation operator or unary function,
 5357 or the mask's domain is not compatible with **bool** (in the case where
 5358 $desc[GrB_MASK].GrB_STRUCTURE$ is not set).

5359 Description

5360 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
 5361 GraphBLAS vector: $w = f(u)$; or, if an optional binary accumulation operator (\odot) is provided,
 5362 $w = w \odot f(u)$.

5363 Logically, this operation occurs in three steps:

5364 **Setup** The internal vectors and mask used in the computation are formed and their domains
 5365 and dimensions are tested for compatibility.

5366 **Compute** The indicated computations are carried out.

5367 **Output** The result is written into the output vector, possibly under control of a mask.

5368 Up to three argument vectors are used in this **GrB_apply** operation:

- 5369 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5370 2. $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$ (optional)
- 5371 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5372 The argument vectors, unary operator and the accumulation operator (if provided) are tested for
 5373 domain compatibility as follows:

- 5374 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
5375 must be from one of the pre-defined types of Table ??.
- 5376 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5377 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5378 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5379 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5380 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in}(\text{op})$.

5381 Two domains are compatible with each other if values from one domain can be cast to values in
5382 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
5383 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5384 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
5385 error listed above is returned.

5386 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
5387 denotes copy):

- 5388 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5389 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 5390 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
 - 5391 (b) If `mask` \neq `GrB_NULL`,
 - 5392 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
 - 5393 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 5394 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5395 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

5396 The internal vectors and masks are checked for dimension compatibility. The following conditions
5397 must hold:

- 5398 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5399 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

5400 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5401 error listed above is returned.

5402 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5403 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5404 We are now ready to carry out the apply and any additional associated operations. We describe
5405 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 ???. 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 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.

5466 Return Values

5467 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 5468 blocking mode, this indicates that the compatibility tests on
 5469 dimensions and domains for the input arguments passed suc-
 5470 cessfully. Either way, output matrix **C** is ready to be used in the
 5471 next method of the sequence.

5472 **GrB_PANIC** Unknown internal error.

5473 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the
 5474 opaque GraphBLAS objects (input or output) is in an invalid
 5475 state caused by a previous execution error. Call **GrB_error()** to
 5476 access any error messages generated by the implementation.

5477 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

5478 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized
 5479 by a call to **new** (or **Matrix_dup** for matrix parameters).

5480 **GrB_DIMENSION_MISMATCH** Mask and **C** dimensions are incompatible, **nrows** \neq **nrows**(**C**), or
 5481 **ncols** \neq **ncols**(**C**).

5482 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
 5483 corresponding domains of the accumulation operator or unary
 5484 function, or the mask's domain is not compatible with **bool** (in
 5485 the case where **desc**[**GrB_MASK**].**GrB_STRUCTURE** is not set).

5486 Description

5487 This variant of **GrB_apply** computes the result of applying a unary function to the elements of a
 5488 GraphBLAS matrix: $C = f(A)$; or, if an optional binary accumulation operator (\odot) is provided,
 5489 $C = C \odot f(A)$.

5490 Logically, this operation occurs in three steps:

5491 **Setup** The internal matrices and mask used in the computation are formed and their domains
 5492 and dimensions are tested for compatibility.

5493 **Compute** The indicated computations are carried out.

5494 **Output** The result is written into the output matrix, possibly under control of a mask.

5495 Up to three argument matrices are used in the **GrB_apply** operation:

- 5496 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5497 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)

5498 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5499 The argument matrices, unary operator and the accumulation operator (if provided) are tested for
5500 domain compatibility as follows:

- 5501 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\text{Mask})$
5502 must be from one of the pre-defined types of Table ??.
- 5503 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 5504 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5505 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
5506 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5507 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in}(\text{op})$ of the unary operator.

5508 Two domains are compatible with each other if values from one domain can be cast to values in
5509 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
5510 compatible with each other. A domain from a user-defined type is only compatible with itself. If
5511 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
5512 error listed above is returned.

5513 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
5514 are formed (\leftarrow denotes copy):

- 5515 1. Matrix $\tilde{C} \leftarrow C$.
- 5516 2. Two-dimensional mask, \tilde{M} , is computed from argument **Mask** as follows:
 - 5517 (a) If **Mask** = **GrB_NULL**, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
5518 $j < \mathbf{ncols}(C)\} \rangle$.
 - 5519 (b) If **Mask** \neq **GrB_NULL**,
 - 5520 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
5521 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - 5522 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
5523 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - 5524 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
- 5525 3. Matrix $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$.

5526 The internal matrices and mask are checked for dimension compatibility. The following conditions
5527 must hold:

- 5528 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.
- 5529 2. $\mathbf{ncols}(\tilde{C}) = \mathbf{ncols}(\tilde{M})$.
- 5530 3. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{A})$.

5531 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

5532 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5533 error listed above is returned.

5534 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5535 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5536 We are now ready to carry out the apply and any additional associated operations. We describe
5537 this in terms of two intermediate matrices:

- 5538 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the unary operator to the input matrix $\tilde{\mathbf{A}}$.
- 5539 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5540 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$5541 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5542 where $f = \mathbf{f}(\mathbf{op})$.

5543 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 5544 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 5545 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5546 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5547 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5548 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 5549 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 5550 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5551 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5552 \quad & \\ 5553 \end{aligned}$$

5554 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5555 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5556 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5557 mask which acts as a “write mask”.

- 5558 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5559 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5560 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

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

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

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

4.3.8.3 apply: Vector-BinaryOp variants[Scott: NEW CONTENT]

Computes the transformation of the values of the stored elements of a vector using a binary operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the first argument to the binary operator and stored elements of the vector are passed as the second argument. In the *bind-second* variant, the elements of the vector are passed as the first argument and the specified scalar value is passed as the second argument. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

C Syntax

```
// bind-first + scalar value
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
                   const GrB_BinaryOp    accum,
                   const GrB_BinaryOp    op,
                   <type>                val,
                   const GrB_Vector      u,
                   const GrB_Descriptor  desc);
```

```
// bind-first + GraphBLAS scalar
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
                   const GrB_BinaryOp    accum,
                   const GrB_BinaryOp    op,
                   const GrB_Scalar      s,
                   const GrB_Vector      u,
                   const GrB_Descriptor  desc);
```

```
// bind-second + scalar value
GrB_Info GrB_apply(GrB_Vector          w,
                   const GrB_Vector     mask,
```

```

5597         const GrB_BinaryOp      accum,
5598         const GrB_BinaryOp      op,
5599         const GrB_Vector        u,
5600         <type>                  val,
5601         const GrB_Descriptor    desc);

5602 // bind-second + GraphBLAS scalar
5603 GrB_Info GrB_apply(GrB_Vector      w,
5604                   const GrB_Vector mask,
5605                   const GrB_BinaryOp accum,
5606                   const GrB_BinaryOp op,
5607                   const GrB_Vector u,
5608                   const GrB_Scalar s,
5609                   const GrB_Descriptor desc);

```

5610 Parameters

5611 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5612 that may be accumulated with the result of the apply operation. On output, this
5613 vector holds the results of the operation.

5614 **mask** (IN) An optional “write” mask that controls which results from this operation are
5615 stored into the output vector **w**. The mask dimensions must match those of the
5616 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
5617 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
5618 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
5619 dimensions of **w**), **GrB_NULL** should be specified.

5620 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
5621 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
5622 specified.

5623 **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar
5624 value, **val**.

5625 **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as
5626 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
5627 argument in the *bind-second* variant.

5628 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
5629 argument in the *bind-first* variant, or the right-hand (second) argument in the
5630 *bind-second* variant.

5631 **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand
5632 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
5633 the *bind-second* variant. It must not be empty.

5634 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 5635 should be specified. Non-default field/value pairs are listed as follows:

5636

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 .

5637

5638 Return Values

5639 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 5640 blocking mode, this indicates that the compatibility tests on di-
 5641 mensions and domains for the input arguments passed successfully.
 5642 Either way, output vector **w** is ready to be used in the next method
 5643 of the sequence.

5644 GrB_PANIC Unknown internal error.

5645 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 5646 GraphBLAS objects (input or output) is in an invalid state caused
 5647 by a previous execution error. Call GrB_error() to access any error
 5648 messages generated by the implementation.

5649 GrB_OUT_OF_MEMORY Not enough memory available for operation.

5650 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 5651 a call to new (or dup for vector parameters).

5652 GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

5653 GrB_DOMAIN_MISMATCH The domains of the various vectors and scalar are incompatible with
 5654 the corresponding domains of the binary operator or accumulation
 5655 operator, or the mask's domain is not compatible with bool (in the
 5656 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

5657 GrB_EMPTY_OBJECT The GrB_Scalar **s** used in the call is empty (**nvals(s) = 0**) and
 5658 therefore a value cannot be passed to the binary operator.

5659 Description

5660 This variant of GrB_apply computes the result of applying a binary operator to the elements of a
 5661 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5662 bind-first: $w = f(\text{val}, u)$ or $w = f(s, u)$

5663 bind-second: $w = f(u, \text{val})$ or $w = f(u, s)$,

5664 or if an optional binary accumulation operator (\odot) is provided:

5665 bind-first: $w = w \odot f(\text{val}, u)$ or $w = w \odot f(s, u)$

5666 bind-second: $w = w \odot f(u, \text{val})$ or $w = w \odot f(u, s)$.

5667 Logically, this operation occurs in three steps:

5668 **Setup** The internal vectors and mask used in the computation are formed and their domains
5669 and dimensions are tested for compatibility.

5670 **Compute** The indicated computations are carried out.

5671 **Output** The result is written into the output vector, possibly under control of a mask.

5672 Up to three argument vectors are used in this GrB_apply operation:

5673 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

5674 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

5675 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5676 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are
5677 tested for domain compatibility as follows:

5678 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
5679 must be from one of the pre-defined types of Table ??.

5680 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.

5681 3. If **accum** is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5682 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
5683 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5684 4. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

5685 5. If bind-first:

5686 (a) $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

5687 (b) If the non-opaque scalar **val** is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
5688 of the binary operator.

5689 (c) If the GrB_Scalar **s** is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
5690 binary operator.

- 5691 6. If bind-second:
- 5692 (a) $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the binary operator.
- 5693 (b) If the non-opaque scalar \mathbf{val} is provided, then $\mathbf{D}(\mathbf{val})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
- 5694 of the binary operator.
- 5695 (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
- 5696 binary operator.

5697 Two domains are compatible with each other if values from one domain can be cast to values in

5698 the other domain as per the rules of the C language. In particular, domains from Table ?? are all

5699 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5700 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5701 error listed above is returned.

5702 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow

5703 denotes copy):

- 5704 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5705 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 5706 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 5707 (b) If `mask \neq GrB_NULL`,
- 5708 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 5709 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 5710 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 5711 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 5712 4. Scalar $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$ (GraphBLAS scalar case).

5713 The internal vectors and masks are checked for dimension compatibility. The following conditions

5714 must hold:

- 5715 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5716 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

5717 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5718 error listed above is returned.

5719 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5720 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5721 If an empty `GrB_Scalar` $\tilde{\mathbf{s}}$ is provided ($\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

5722 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

5723 `val` with the same domain as $\tilde{\mathbf{s}}$ and set `val = val($\tilde{\mathbf{s}}$)`.

5724 We are now ready to carry out the apply and any additional associated operations. We describe

5725 this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$: The vector holding the result from applying the binary operator to the input vector $\tilde{\mathbf{u}}$.
- $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\tilde{\mathbf{t}}$, is created as one of the following:

$$\begin{aligned} \text{bind-first: } \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\text{val}, \tilde{\mathbf{u}}(i))) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \\ \text{bind-second: } \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i), \text{val})) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \end{aligned}$$

where $f = \mathbf{f}(\text{op})$.

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

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

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

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

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

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

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

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

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

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

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

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

5759 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

5760 Computes the transformation of the values of the stored elements of a matrix using a binary
5761 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the
5762 first argument to the binary operator and stored elements of the matrix are passed as the second
5763 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument
5764 and the specified scalar value is passed as the second argument. The scalar can be passed either as
5765 a non-opaque variable or as a GrB_Scalar object.

5766 C Syntax

```
5767 // bind-first + scalar value
5768 GrB_Info GrB_apply(GrB_Matrix      C,
5769                   const GrB_Matrix Mask,
5770                   const GrB_BinaryOp accum,
5771                   const GrB_BinaryOp op,
5772                   <type>           val,
5773                   const GrB_Matrix A,
5774                   const GrB_Descriptor desc);
```

```
5775 // bind-first + GraphBLAS scalar
5776 GrB_Info GrB_apply(GrB_Matrix      C,
5777                   const GrB_Matrix Mask,
5778                   const GrB_BinaryOp accum,
5779                   const GrB_BinaryOp op,
5780                   const GrB_Scalar s,
5781                   const GrB_Matrix A,
5782                   const GrB_Descriptor desc);
```

```
5783 // bind-second + scalar value
5784 GrB_Info GrB_apply(GrB_Matrix      C,
5785                   const GrB_Matrix Mask,
5786                   const GrB_BinaryOp accum,
5787                   const GrB_BinaryOp op,
5788                   const GrB_Matrix A,
5789                   <type>           val,
5790                   const GrB_Descriptor desc);
```

```
5791 // bind-second + GraphBLAS scalar
5792 GrB_Info GrB_apply(GrB_Matrix      C,
5793                   const GrB_Matrix Mask,
5794                   const GrB_BinaryOp accum,
5795                   const GrB_BinaryOp op,
5796                   const GrB_Matrix A,
```

```

5797         const GrB_Scalar      s,
5798         const GrB_Descriptor desc);

```

5799 Parameters

5800 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5801 that may be accumulated with the result of the apply operation. On output, the
5802 matrix holds the results of the operation.

5803 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5804 stored into the output matrix C. The mask dimensions must match those of the
5805 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
5806 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
5807 in Table ???. If the default mask is desired (i.e., a mask that is all `true` with the
5808 dimensions of C), `GrB_NULL` should be specified.

5809 **accum** (IN) An optional binary operator used for accumulating entries into existing C
5810 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
5811 specified.

5812 **op** (IN) A binary operator applied to each element of input matrix, A, with the element
5813 of the input matrix used as the left-hand argument, and the scalar value, `val`, used
5814 as the right-hand argument.

5815 **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as
5816 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
5817 argument in the *bind-second* variant.

5818 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
5819 argument in the *bind-first* variant, or the right-hand (second) argument in the
5820 *bind-second* variant.

5821 **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand
5822 (first) argument in the *bind-first* variant, or the right-hand (second) argument in
5823 the *bind-second* variant. It must not be empty.

5824 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
5825 should be specified. Non-default field/value pairs are listed as follows:
5826

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.

5854 Description

5855 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a
 5856 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

5857 bind-first: $C = f(\text{val}, A)$ or $C = f(s, A)$

5858 bind-second: $C = f(A, \text{val})$ or $C = f(A, s)$,

5859 or if an optional binary accumulation operator (\odot) is provided:

5860 bind-first: $C = C \odot f(\text{val}, A)$ or $C = C \odot f(s, A)$

5861 bind-second: $C = C \odot f(A, \text{val})$ or $C = C \odot f(A, s)$.

5862 Logically, this operation occurs in three steps:

5863 **Setup** The internal matrices and mask used in the computation are formed and their domains
 5864 and dimensions are tested for compatibility.

5865 **Compute** The indicated computations are carried out.

5866 **Output** The result is written into the output matrix, possibly under control of a mask.

5867 Up to three argument matrices are used in the `GrB_apply` operation:

- 5868 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5869 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)
- 5870 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5871 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are
 5872 tested for domain compatibility as follows:

- 5873 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 5874 must be from one of the pre-defined types of Table ??.
- 5875 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.
- 5876 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5877 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
 5878 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5879 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.
- 5880 5. If bind-first:
 5881 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

5882 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$
 5883 of the binary operator.

5884 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the
 5885 binary operator.

5886 6. If `bind-second`:

5887 (a) $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

5888 (b) If the non-opaque scalar val is provided, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$
 5889 of the binary operator.

5890 (c) If the `GrB_Scalar` s is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the
 5891 binary operator.

5892 Two domains are compatible with each other if values from one domain can be cast to values in
 5893 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 5894 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5895 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
 5896 error listed above is returned.

5897 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
 5898 are formed (\leftarrow denotes copy):

5899 1. Matrix $\tilde{C} \leftarrow C$.

5900 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:

5901 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
 5902 $j < \mathbf{ncols}(C)\} \rangle$.

5903 (b) If `Mask` \neq `GrB_NULL`,

5904 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
 5905 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,

5906 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
 5907 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.

5908 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.

5909 3. Matrix \tilde{A} is computed from argument `A` as follows:

5910 `bind-first`: $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? A^T : A$

5911 `bind-second`: $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$

5912 4. Scalar $\tilde{s} \leftarrow s$ (`GraphBLAS` scalar case).

5913 The internal matrices and mask are checked for dimension compatibility. The following conditions
 5914 must hold:

5915 1. $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$.

5916 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.

5917 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

5918 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

5919 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
5920 error listed above is returned.

5921 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
5922 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5923 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.

5924 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
5925 `val` with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

5926 We are now ready to carry out the apply and any additional associated operations. We describe
5927 this in terms of two intermediate matrices:

- 5928 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the binary operator to the input matrix $\tilde{\mathbf{A}}$.
- 5929 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5930 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as one of the following:

5931 bind-first: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

5932 bind-second: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

5933 where $f = \mathbf{f}(\mathbf{op})$.

5934 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 5935 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 5936 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5937 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5938 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5939 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5940 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5941 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5942 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5943 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

5946 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 5947 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 5948 mask which acts as a “write mask”.

- 5949 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 5950 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5951 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 5952 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 5953 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 5954 mask are unchanged:

$$5955 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5956 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 5957 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 5958 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 5959 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 5960 sequence.

5961 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

5962 Computes the transformation of the values of the stored elements of a vector using an index unary
 5963 operator that is a function of the stored value, its location indices, and an user provided scalar
 5964 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

5965 C Syntax

```
5966      GrB_Info GrB_apply(GrB_Vector          w,
5967                        const GrB_Vector      mask,
5968                        const GrB_BinaryOp    accum,
5969                        const GrB_IndexUnaryOp op,
5970                        const GrB_Vector      u,
5971                        <type>                val,
5972                        const GrB_Descriptor  desc);
```

```
5973      GrB_Info GrB_apply(GrB_Vector          w,
5974                        const GrB_Vector      mask,
5975                        const GrB_BinaryOp    accum,
5976                        const GrB_IndexUnaryOp op,
5977                        const GrB_Vector      u,
5978                        const GrB_Scalar      s,
5979                        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 ???. If the default mask is desired (i.e., a mask that is all **true** with the dimensions of **w**), **GrB_NULL** should be specified.

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

op (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\mathbf{GrB_Index}), D_{in_2}, f_i \rangle$, applied to each element stored in the input vector, **u**. It is a function of the stored element’s value, its location index, and a user supplied scalar value (either **s** or **val**).

u (IN) The GraphBLAS vector whose elements are passed to the index unary operator.

val (IN) An additional scalar value that is passed to the index unary operator.

s (IN) An additional GraphBLAS scalar that is passed to the index unary operator. It must not be empty.

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

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

Return Values

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

6011 GrB_PANIC Unknown internal error.

6012 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6013 opaque GraphBLAS objects (input or output) is in an invalid
6014 state caused by a previous execution error. Call `GrB_error()` to
6015 access any error messages generated by the implementation.

6016 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6017 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6018 by a call to `new` (or another constructor).

6019 GrB_DIMENSION_MISMATCH `mask`, `w` and/or `u` dimensions are incompatible.

6020 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
6021 responding domains of the accumulation operator or index unary
6022 operator, or the mask's domain is not compatible with `bool` (in
6023 the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6024 GrB_EMPTY_OBJECT The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and
6025 therefore a value cannot be passed to the index unary operator.

6026 Description

6027 This variant of `GrB_apply` computes the result of applying an index unary operator to the elements
6028 of a GraphBLAS vector each composed with the element's index and a scalar constant, `val` or `s`:

$$6029 \qquad w = f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = f_i(u, \mathbf{ind}(u), 0, s),$$

6030 or if an optional binary accumulation operator (\odot) is provided:

$$6031 \qquad w = w \odot f_i(u, \mathbf{ind}(u), 0, \text{val}) \text{ or } w = w \odot f_i(u, \mathbf{ind}(u), 0, s).$$

6032 Logically, this operation occurs in three steps:

6033 **Setup** The internal vectors and mask used in the computation are formed and their domains
6034 and dimensions are tested for compatibility.

6035 **Compute** The indicated computations are carried out.

6036 **Output** The result is written into the output vector, possibly under control of a mask.

6037 Up to three argument vectors are used in this `GrB_apply` operation:

- 6038 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6039 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

6040 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6041 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6042 are tested for domain compatibility as follows:

- 6043 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
6044 must be from one of the pre-defined types of Table ??.
- 6045 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the index unary operator.
- 6046 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6047 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be compatible
6048 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6049 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6050 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
6051 the index unary operator.
- 6052 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
6053 unary operator.

6054 Two domains are compatible with each other if values from one domain can be cast to values in
6055 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
6056 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6057 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
6058 error listed above is returned.

6059 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6060 denotes copy):

- 6061 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6062 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 6063 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 6064 (b) If `mask` \neq `GrB_NULL`,
 - 6065 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,
 - 6066 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 6067 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6068 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 6069 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6070 The internal vectors and masks are checked for dimension compatibility. The following conditions
6071 must hold:

6072 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6073 2. $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$.

6074 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6075 error listed above is returned.

6076 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6077 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6078 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6079 If a non-empty `GrB_Scalar`, \tilde{s} , is provided ($\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable `val`
6080 with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6081 We are now ready to carry out the apply and any additional associated operations. We describe
6082 this in terms of two intermediate vectors:

- 6083 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6084 $\tilde{\mathbf{u}}$.
- 6085 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6086 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6087 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6088 where $f_i = \mathbf{f}(\text{op})$.

6089 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6090 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 6091 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6092 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6093 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6094 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6095 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6096 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6097 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6098 \quad & \\ 6099 \end{aligned}$$

6100 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6101 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
6102 using what is called a *standard vector mask and replace*. This is carried out under control of the
6103 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.

6116 4.3.8.6 apply: Matrix index unary operator variant[Scott: NEW CONTENT]

6117 Computes the transformation of the values of the stored elements of a matrix using an index unary
6118 operator that is a function of the stored value, its location indices, and an user provided scalar
6119 value. The scalar can be passed either as a non-opaque variable or as a GrB_Scalar object.

6120 C Syntax

```
6121     GrB_Info GrB_apply(GrB_Matrix      C,
6122                       const GrB_Matrix Mask,
6123                       const GrB_BinaryOp accum,
6124                       const GrB_IndexUnaryOp op,
6125                       const GrB_Matrix A,
6126                       <type>          val,
6127                       const GrB_Descriptor desc);
```

```
6128     GrB_Info GrB_apply(GrB_Matrix      C,
6129                       const GrB_Matrix Mask,
6130                       const GrB_BinaryOp accum,
6131                       const GrB_IndexUnaryOp op,
6132                       const GrB_Matrix A,
6133                       const GrB_Scalar s,
6134                       const GrB_Descriptor desc);
```

6135 Parameters

6136 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6137 that may be accumulated with the result of the apply operation. On output, the
6138 matrix holds the results of the operation.

6139 Mask (IN) An optional “write” mask that controls which results from this operation are
6140 stored into the output matrix C. The mask dimensions must match those of the
6141 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
6142 of the Mask matrix must be of type **bool** or any of the predefined “built-in” types
6143 in Table ???. If the default mask is desired (i.e., a mask that is all **true** with the
6144 dimensions of C), GrB_NULL should be specified.

6145 accum (IN) An optional binary operator used for accumulating entries into existing C
6146 entries. If assignment rather than accumulation is desired, GrB_NULL should be
6147 specified.

6148 op (IN) An index unary operator, $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB_Index}), D_{in_2}, f_i \rangle$, applied
6149 to each element stored in the input matrix, A. It is a function of the stored element’s
6150 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6151 A (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6152 ator.

6153 val (IN) An additional scalar value that is passed to the index unary operator.

6154 s (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6155 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
6156 should be specified. Non-default field/value pairs are listed as follows:

6157

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.

6158

6159 Return Values

6160 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
6161 blocking mode, this indicates that the compatibility tests on di-
6162 mensions and domains for the input arguments passed successfully.
6163 Either way, output matrix C is ready to be used in the next method
6164 of the sequence.

6165 GrB_PANIC Unknown internal error.

6166 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
6167 GraphBLAS objects (input or output) is in an invalid state caused

6168 by a previous execution error. Call `GrB_error()` to access any error
 6169 messages generated by the implementation.

6170 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

6171 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 6172 a call to `new` (or another constructor).

6173 **GrB_DIMENSION_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6174 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the
 6175 corresponding domains of the accumulation operator or index unary
 6176 operator, or the mask's domain is not compatible with `bool` (in the
 6177 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6178 **GrB_EMPTY_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and
 6179 therefore a value cannot be passed to the index unary operator.

6180 Description

6181 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements
 6182 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar
 6183 constant, `val` or `s`:

$$6184 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6185 or if an optional binary accumulation operator (\odot) is provided:

$$6186 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6187 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
 6188 indices, respectively.

6189 Logically, this operation occurs in three steps:

6190 **Setup** The internal matrices and mask used in the computation are formed and their domains
 6191 and dimensions are tested for compatibility.

6192 **Compute** The indicated computations are carried out.

6193 **Output** The result is written into the output matrix, possibly under control of a mask.

6194 Up to three argument matrices are used in the `GrB_apply` operation:

- 6195 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6196 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)

6197 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6198 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6199 are tested for domain compatibility as follows:

- 6200 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\text{Mask})$
6201 must be from one of the pre-defined types of Table ??.
- 6202 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the index unary operator.
- 6203 3. If **accum** is not **GrB_NULL**, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6204 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be compatible
6205 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6206 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6207 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
6208 the index unary operator.
- 6209 6. If the **GrB_Scalar** **s** is provided, then $\mathbf{D}(s)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the index
6210 unary operator.

6211 Two domains are compatible with each other if values from one domain can be cast to values in
6212 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
6213 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6214 any compatibility rule above is violated, execution of **GrB_apply** ends and the domain mismatch
6215 error listed above is returned.

6216 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
6217 are formed (\leftarrow denotes copy):

- 6218 1. Matrix $\tilde{C} \leftarrow C$.
- 6219 2. Two-dimensional mask, \tilde{M} , is computed from argument **Mask** as follows:
 - 6220 (a) If **Mask** = **GrB_NULL**, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
6221 $j < \mathbf{ncols}(C)\} \rangle$.
 - 6222 (b) If **Mask** \neq **GrB_NULL**,
 - 6223 i. If **desc[GrB_MASK].GrB_STRUCTURE** is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
6224 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - 6225 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
6226 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - 6227 (c) If **desc[GrB_MASK].GrB_COMP** is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
- 6228 3. Matrix \tilde{A} is computed from argument **A** as follows:
 - 6229 $\tilde{A} \leftarrow \text{desc[GrB_INP0].GrB_TRAN} ? A^T : A$
- 6230 4. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6231 The internal matrices and mask are checked for dimension compatibility. The following conditions
6232 must hold:

- 6233 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 6234 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 6235 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 6236 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

6237 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
6238 error listed above is returned.

6239 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6240 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6241 If an empty `GrB_Scalar` \tilde{s} is provided ($\mathbf{nvals}(\tilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6242 If a non-empty `GrB_Scalar`, \tilde{s} , is provided (i.e., $\mathbf{nvals}(\tilde{s}) = 1$), we then create an internal variable
6243 `val` with the same domain as \tilde{s} and set `val = val(\tilde{s})`.

6244 We are now ready to carry out the apply and any additional associated operations. We describe
6245 this in terms of two intermediate matrices:

- 6246 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6247 $\tilde{\mathbf{A}}$.
- 6248 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6249 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6250 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6251 where $f_i = \mathbf{f}(\mathbf{op})$.

6252 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6253 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6254 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6255 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6256 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6257 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$\begin{aligned} 6258 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6259 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6260 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6261 \end{aligned}$$

6263 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

6264 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6265 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6266 mask which acts as a “write mask”.

- 6267 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
6268 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6269 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6270 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6271 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6272 mask are unchanged:

$$6273 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6274 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
6275 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
6276 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
6277 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6278 sequence.

6279 4.3.9 select:

6280 Apply a select operator to the stored elements of an object to determine whether or not to keep
6281 them.

6282 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6283 Apply a select operator (an index unary operator) to the elements of a vector.

6284 C Syntax

```
6285 // scalar value variant
6286 GrB_Info GrB_select(GrB_Vector          w,
6287                    const GrB_Vector     mask,
6288                    const GrB_BinaryOp   accum,
6289                    const GrB_IndexUnaryOp op,
6290                    const GrB_Vector     u,
6291                    <type>               val,
6292                    const GrB_Descriptor desc);
6293
6294 // GraphBLAS scalar variant
6295 GrB_Info GrB_select(GrB_Vector          w,
6296                    const GrB_Vector     mask,
```

```

6297         const GrB_BinaryOp      accum,
6298         const GrB_IndexUnaryOp  op,
6299         const GrB_Vector        u,
6300         const GrB_Scalar        s,
6301         const GrB_Descriptor    desc);
6302

```

6303 Parameters

6304 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6305 that may be accumulated with the result of the select operation. On output, this
6306 vector holds the results of the operation.

6307 **mask** (IN) An optional “write” mask that controls which results from this operation are
6308 stored into the output vector **w**. The mask dimensions must match those of the
6309 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6310 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6311 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
6312 dimensions of **w**), **GrB_NULL** should be specified.

6313 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6314 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6315 specified.

6316 **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
6317 to each element stored in the input vector, **u**. It is a function of the stored element’s
6318 value, its location index, and a user supplied scalar value (either **s** or **val**).

6319 **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-
6320 ator.

6321 **val** (IN) An additional scalar value that is passed to the index unary operator.

6322 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6323 not be empty.

6324 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6325 should be specified. Non-default field/value pairs are listed as follows:

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

6328 Return Values

6329 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
6330 blocking mode, this indicates that the compatibility tests on di-
6331 mensions and domains for the input arguments passed success-
6332 fully. Either way, output vector **w** is ready to be used in the next
6333 method of the sequence.

6334 GrB_PANIC Unknown internal error.

6335 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the
6336 opaque GraphBLAS objects (input or output) is in an invalid
6337 state caused by a previous execution error. Call **GrB_error()** to
6338 access any error messages generated by the implementation.

6339 GrB_OUT_OF_MEMORY Not enough memory available for operation.

6340 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized
6341 by a call to one of its constructors.

6342 GrB_DIMENSION_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

6343 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
6344 responding domains of the accumulation operator or index unary
6345 operator, or the **mask**'s domain is not compatible with **bool** (in
6346 the case where **desc[GrB_MASK].GrB_STRUCTURE** is not set).

6347 GrB_EMPTY_OBJECT The **GrB_Scalar s** used in the call is empty (**nvals(s) = 0**) and
6348 therefore a value cannot be passed to the index unary operator.

6349 Description

6350 This variant of **GrB_select** computes the result of applying a index unary operator to select the
6351 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored
6352 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding
6353 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.
6354 This acts like a functional mask on the input vector as follows:

$$6355 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6356 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6357 Correspondingly, if a **GrB_Scalar s**, is provided:

$$6358 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6359 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6360 Logically, this operation occurs in three steps:

6361 **Setup** The internal vectors and mask used in the computation are formed and their domains
6362 and dimensions are tested for compatibility.

6363 **Compute** The indicated computations are carried out.

6364 **Output** The result is written into the output vector, possibly under control of a mask.

6365 Up to three argument vectors are used in this GrB_select operation:

- 6366 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
6367 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
6368 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6369 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)
6370 are tested for domain compatibility as follows:

- 6371 1. If **mask** is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\mathbf{mask})$
6372 must be from one of the pre-defined types of Table ??.
- 6373 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 6374 3. If **accum** is not GrB_NULL, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
6375 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
6376 mulation operator.
- 6377 4. $\mathbf{D}_{out}(\mathbf{op})$ of the index unary operator must be from one of the pre-defined types of Table ??;
6378 i.e., castable to **bool**.
- 6379 5. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the index unary operator.
- 6380 6. $\mathbf{D}(\mathbf{val})$ or $\mathbf{D}(\mathbf{s})$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$
6381 of the index unary operator.

6382 Two domains are compatible with each other if values from one domain can be cast to values in
6383 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
6384 compatible with each other. A domain from a user-defined type is only compatible with itself. If
6385 any compatibility rule above is violated, execution of GrB_select ends and the domain mismatch
6386 error listed above is returned.

6387 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
6388 denotes copy):

- 6389 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
6390 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument **mask** as follows:

- 6391 (a) If $\text{mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$.
- 6392 (b) If $\text{mask} \neq \text{GrB_NULL}$,
- 6393 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
- 6394 ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 6395 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
- 6396 3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 6397 4. Scalar $\widetilde{s} \leftarrow s$ (GrB_Scalar version only).

6398 The internal vectors and masks are checked for dimension compatibility. The following conditions
6399 must hold:

- 6400 1. $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6401 2. $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$.

6402 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch
6403 error listed above is returned.

6404 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6405 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6406 If an empty `GrB_Scalar` \widetilde{s} is provided (i.e., $\text{nvals}(\widetilde{s}) = 0$), the method returns with code `GrB_EMPTY_OBJECT`.
6407 If a non-empty `GrB_Scalar`, \widetilde{s} , is provided (i.e., $\text{nvals}(\widetilde{s}) = 1$), we then create an internal variable
6408 `val` with the same domain as \widetilde{s} and set $\text{val} = \text{val}(\widetilde{s})$.

6409 We are now ready to carry out the `select` and any additional associated operations. We describe
6410 this in terms of two intermediate vectors:

- 6411 • $\widetilde{\mathbf{t}}$: The vector holding the result from applying the index unary operator to the input vector
6412 $\widetilde{\mathbf{u}}$.
- 6413 • $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6414 The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$6415 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6416 where $f_i = \mathbf{f}(\text{op})$.

6417 The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6418 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- 6419 • If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$6420 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{\text{out}}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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);
```

```

6456 // GraphBLAS scalar variant
6457 GrB_Info GrB_select(GrB_Matrix          C,
6458                   const GrB_Matrix      Mask,
6459                   const GrB_BinaryOp    accum,
6460                   const GrB_IndexUnaryOp op,
6461                   const GrB_Matrix      A,
6462                   const GrB_Scalar      s,
6463                   const GrB_Descriptor   desc);

```

6464 Parameters

6465 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6466 that may be accumulated with the result of the select operation. On output, the
6467 matrix holds the results of the operation.

6468 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6469 stored into the output matrix **C**. The mask dimensions must match those of the
6470 matrix **C**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6471 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types
6472 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
6473 dimensions of **C**), **GrB_NULL** should be specified.

6474 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
6475 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6476 specified.

6477 **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
6478 to each element stored in the input matrix, **A**. It is a function of the stored element’s
6479 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6480 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-
6481 ator.

6482 **val** (IN) An additional scalar value that is passed to the index unary operator.

6483 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must
6484 not be empty.

6485 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6486 should be specified. Non-default field/value pairs are listed as follows:

6487

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:

6517 $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$, or
6518 $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$.

6519 Correspondingly, if a GrB_Scalar, s, is provided:

6520 $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$, or
6521 $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$.

6522 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional
6523 indices, respectively.

6524 Logically, this operation occurs in three steps:

6525 **Setup** The internal matrices and mask used in the computation are formed and their domains
6526 and dimensions are tested for compatibility.

6527 **Compute** The indicated computations are carried out.

6528 **Output** The result is written into the output matrix, possibly under control of a mask.

6529 Up to three argument matrices are used in the GrB_select operation:

- 6530 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6531 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)
- 6532 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6533 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)
6534 are tested for domain compatibility as follows:

- 6535 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
6536 must be from one of the pre-defined types of Table ??.
- 6537 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 6538 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
6539 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
6540 mulation operator.
- 6541 4. $\mathbf{D}_{out}(\text{op})$ of the index unary operator must be from one of the pre-defined types of Table ??;
6542 i.e., castable to bool.
- 6543 5. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the index unary operator.
- 6544 6. $\mathbf{D}(\text{val})$ or $\mathbf{D}(s)$, depending on the signature of the method, must be compatible with $\mathbf{D}_{in_2}(\text{op})$
6545 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 ?? 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:

- 6580 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the index unary operator to the input matrix
6581 $\tilde{\mathbf{A}}$.
- 6582 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

6583 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$6584 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6585 where $f_i = \mathbf{f}(\text{op})$.

6586 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 6587 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 6588 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$6589 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6590 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
6591 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$6592 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6593 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6594 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6595 \quad 6596$$

6597 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6598 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
6599 using what is called a *standard matrix mask and replace*. This is carried out under control of the
6600 mask which acts as a “write mask”.

- 6601 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
6602 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$6603 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6604 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
6605 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
6606 mask are unchanged:

$$6607 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6608 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6609 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6610 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
6611 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6612 sequence.

6613 4.3.10 reduce: Perform a reduction across the elements of an object

6614 Computes the reduction of the values of the elements of a vector or matrix.

6615 4.3.10.1 reduce: Standard matrix to vector variant

6616 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns
6617 is desired, the input matrix should be transposed using the descriptor.

6618 C Syntax

```
6619         GrB_Info GrB_reduce(GrB_Vector          w,  
6620                             const GrB_Vector    mask,  
6621                             const GrB_BinaryOp   accum,  
6622                             const GrB_Monoid     op,  
6623                             const GrB_Matrix     A,  
6624                             const GrB_Descriptor desc);  
6625  
6626         GrB_Info GrB_reduce(GrB_Vector          w,  
6627                             const GrB_Vector    mask,  
6628                             const GrB_BinaryOp   accum,  
6629                             const GrB_BinaryOp   op,  
6630                             const GrB_Matrix     A,  
6631                             const GrB_Descriptor desc);
```

6632 Parameters

6633 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
6634 that may be accumulated with the result of the reduction operation. On output,
6635 this vector holds the results of the operation.

6636 **mask** (IN) An optional “write” mask that controls which results from this operation are
6637 stored into the output vector **w**. The mask dimensions must match those of the
6638 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
6639 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
6640 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
6641 dimensions of **w**), **GrB_NULL** should be specified.

6642 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
6643 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6644 specified.

6645 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.
6646 Depending on which type is passed, the following defines the binary operator with
6647 one domain, $F_b = \langle D, D, D, \oplus \rangle$, that is used:

6648 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 6649 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
 6650 monoid is ignored.

6651 If `op` is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in
 6652 both cases $\odot(\text{op})$ must be commutative and associative. Otherwise, the outcome
 6653 of the operation is undefined.

6654 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

6655 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 6656 should be specified. Non-default field/value pairs are listed as follows:
 6657

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.

6659 Return Values

6660 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 6661 blocking mode, this indicates that the compatibility tests on di-
 6662 mensions and domains for the input arguments passed successfully.
 6663 Either way, output vector `w` is ready to be used in the next method
 6664 of the sequence.

6665 **GrB_PANIC** Unknown internal error.

6666 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 6667 GraphBLAS objects (input or output) is in an invalid state caused
 6668 by a previous execution error. Call `GrB_error()` to access any error
 6669 messages generated by the implementation.

6670 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

6671 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 6672 a call to `new` (or `dup` for vector parameters).

6673 **GrB_DIMENSION_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6674 **GrB_DOMAIN_MISMATCH** Either the domains of the various vectors and matrices are incom-
 6675 patible with the corresponding domains of the accumulation oper-
 6676 ator or reduce function, or the domains of the GraphBLAS binary

operator `op` are not all the same, or the mask's domain is not compatible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6680 Description

6681 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows
 6682 of an input matrix: $w(i) = \bigoplus A(i, :) \forall i$; or, if an optional binary accumulation operator is provided,
 6683 $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$, where $\bigoplus = \odot(F_b)$ and $\odot = \odot(\text{accum})$.

6684 Logically, this operation occurs in three steps:

6685 **Setup** The internal vector, matrix and mask used in the computation are formed and their
 6686 domains and dimensions are tested for compatibility.

6687 **Compute** The indicated computations are carried out.

6688 **Output** The result is written into the output vector, possibly under control of a mask.

6689 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6690 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6691 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 6692 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6693 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested
 6694 for domain compatibility as follows:

- 6695 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 6696 must be from one of the pre-defined types of Table ??.
- 6697 2. $\mathbf{D}(w)$ must be compatible with the domain of the reduction binary operator, $\mathbf{D}(F_b)$.
- 6698 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 6699 of the accumulation operator and $\mathbf{D}(F_b)$, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 6700 mulation operator.
- 6701 4. $\mathbf{D}(A)$ must be compatible with the domain of the binary reduction operator, $\mathbf{D}(F_b)$.

6702 Two domains are compatible with each other if values from one domain can be cast to values in
 6703 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 6704 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6705 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 6706 error listed above is returned.

6707 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 6708 denotes copy):

- 6709 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 6710 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 6711 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 6712 (b) If `mask \neq GrB_NULL`,
- 6713 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 6714 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 6715 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 6716 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

6717 The internal vectors and masks are checked for dimension compatibility. The following conditions
6718 must hold:

- 6719 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6720 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

6721 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-
6722 match error listed above is returned.

6723 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
6724 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6725 We carry out the reduce and any additional associated operations. We describe this in terms of
6726 two intermediate vectors:

- 6727 • $\tilde{\mathbf{t}}$: The vector holding the result from reducing along the rows of input matrix $\tilde{\mathbf{A}}$.
- 6728 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

6729 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$6730 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6731 The value of each of its elements is computed by

$$6732 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6733 where $\bigoplus = \odot(F_b)$.

6734 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 6735 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

6736 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$6737 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6738 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
6739 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 6740 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 6741 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6742 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6743 \quad & \\ 6744 \end{aligned}$$

6745 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

6746 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
6747 using what is called a *standard vector mask and replace*. This is carried out under control of the
6748 mask which acts as a “write mask”.

6749 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
6750 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$6751 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6752 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
6753 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
6754 mask are unchanged:

$$6755 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

6756 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
6757 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
6758 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but
6759 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
6760 sequence.

6761 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

6762 Reduce all stored values into a single scalar.

6763 C Syntax

```
6764 // scalar value + monoid (only)
6765 GrB_Info GrB_reduce(<type>          *val,
6766                      const GrB_BinaryOp accum,
6767                      const GrB_Monoid  op,
6768                      const GrB_Vector  u,
```

```

6769             const GrB_Descriptor desc);
6770
6771 // GraphBLAS Scalar + monoid
6772 GrB_Info GrB_reduce(GrB_Scalar      s,
6773                    const GrB_BinaryOp accum,
6774                    const GrB_Monoid op,
6775                    const GrB_Vector u,
6776                    const GrB_Descriptor desc);
6777
6778 // GraphBLAS Scalar + binary operator
6779 GrB_Info GrB_reduce(GrB_Scalar      s,
6780                    const GrB_BinaryOp accum,
6781                    const GrB_BinaryOp op,
6782                    const GrB_Vector u,
6783                    const GrB_Descriptor desc);

```

6784 Parameters

6785 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
6786 a value that may be accumulated (optionally) with the result of the reduction
6787 operation. On output, this scalar holds the results of the operation.

6788 **accum** (IN) An optional binary operator used for accumulating entries into an exist-
6789 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,
6790 GrB_NULL should be specified.

6791 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
6792 the reduction operation. The \oplus operator must be commutative and associative;
6793 otherwise, the outcome of the operation is undefined.

6794 **u** (IN) The GraphBLAS vector on which reduction will be performed.

6795 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
6796 should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
-------	-------	-------	-------------

6799 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
6800 tions. There are currently no non-default field/value pairs that can be set for this
6801 operation.

6802 Return Values

6803 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
6804 cessfully, and the output scalar (**s** or **val**) is ready to be used in the
6805 next method of the sequence.

6806 GrB_PANIC Unknown internal error.

6807 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
6808 GraphBLAS objects (input or output) is in an invalid state caused
6809 by a previous execution error. Call GrB_error() to access any error
6810 messages generated by the implementation.

6811 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

6812 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
6813 a call to a respective constructor.

6814 GrB_NULL_POINTER val pointer is NULL.

6815 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
6816 the corresponding domains of the accumulation operator, or reduce
6817 operator.

6818 Description

This variant of GrB_reduce computes the result of performing a reduction across all of the stored elements of an input vector storing the result into either **s** or **val**. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[\bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

6819 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

6820 Logically, this operation occurs in three steps:

6821 **Setup** The internal vector used in the computation is formed and its domain is tested for
6822 compatibility.

6823 **Compute** The indicated computations are carried out.

6824 **Output** The result is written into the output scalar.

6825 One vector argument is used in this GrB_reduce operation:

- 6826 1. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6827 The output scalar, argument vector, reduction operator and accumulation operator (if provided)
6828 are tested for domain compatibility as follows:

- 6829 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
6830 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

- 6831 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
 6832 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 6833 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 6834 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).

6835 Two domains are compatible with each other if values from one domain can be cast to values in
 6836 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 6837 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6838 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 6839 error listed above is returned.

6840 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one
 6841 of the following occurs depending on the output variant:

$$6842 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

6843 or

$$6844 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

6845 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

6846 For all other cases, the internal vector and scalar used in the computation is formed (\leftarrow denotes
 6847 copy):

- 6848 1. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 6849 2. Scalar $\tilde{s} \leftarrow \text{s}$ (GraphBLAS scalar case).

6850 We are now ready to carry out the reduction and any additional associated operations. An inter-
 6851 mediate scalar result t is computed as follows:

$$6852 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

6853 where $\oplus = \odot(\text{op})$.

6854 The final reduction value is computed as follows:

$$6855 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

6856 or

$$6857 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

6858 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
6859 GrB_SUCCESS and the new contents of the output scalar is as defined above.

6860 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

6861 Reduce all stored values into a single scalar.

6862 C Syntax

```
6863 // scalar value + monoid (only)
6864 GrB_Info GrB_reduce(<type>          *val,
6865                    const GrB_BinaryOp accum,
6866                    const GrB_Monoid   op,
6867                    const GrB_Matrix   A,
6868                    const GrB_Descriptor desc);
6869
6870 // GraphBLAS Scalar + monoid
6871 GrB_Info GrB_reduce(GrB_Scalar      s,
6872                    const GrB_BinaryOp accum,
6873                    const GrB_Monoid   op,
6874                    const GrB_Matrix   A,
6875                    const GrB_Descriptor desc);
6876
6877 // GraphBLAS Scalar + binary operator
6878 GrB_Info GrB_reduce(GrB_Scalar      s,
6879                    const GrB_BinaryOp accum,
6880                    const GrB_BinaryOp op,
6881                    const GrB_Matrix   A,
6882                    const GrB_Descriptor desc);
```

6883 Parameters

6884 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
6885 a value that may be accumulated (optionally) with the result of the reduction
6886 operation. On output, this scalar holds the results of the operation.

6887 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or
6888 **val**) value. If assignment rather than accumulation is desired, GrB_NULL should
6889 be specified.

6890 **op** (IN) The monoid ($M = \langle D, \oplus, 0 \rangle$) or binary operator ($F_b = \langle D, D, D, \oplus \rangle$) used in
6891 the reduction operation. The \oplus operator must be commutative and associative;
6892 otherwise, the outcome of the operation is undefined.

6893 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

6894 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
6895 should be specified. Non-default field/value pairs are listed as follows:
6896

6897	Param	Field	Value	Description
------	-------	-------	-------	-------------

6898 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
6899 tions. There are currently no non-default field/value pairs that can be set for this
6900 operation.

6901 Return Values

6902 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
6903 cessfully, and the output scalar (s or val) is ready to be used in the
6904 next method of the sequence.

6905 GrB_PANIC Unknown internal error.

6906 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
6907 GraphBLAS objects (input or output) is in an invalid state caused
6908 by a previous execution error. Call GrB_error() to access any error
6909 messages generated by the implementation.

6910 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

6911 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
6912 a call to a respective constructor.

6913 GrB_NULL_POINTER val pointer is NULL.

6914 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
6915 the corresponding domains of the accumulation operator, or reduce
6916 operator.

6917 Description

This variant of GrB_reduce computes the result of performing a reduction across all of the stored elements of an input matrix storing the result into either s or val. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[\bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

6918 where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

6919 Logically, this operation occurs in three steps:

6920 **Setup** The internal matrix used in the computation is formed and its domain is tested for
 6921 compatibility.

6922 **Compute** The indicated computations are carried out.

6923 **Output** The result is written into the output scalar.

6924 One matrix argument is used in this GrB_reduce operation:

6925 1. $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

6926 The output scalar, argument matrix, reduction operator and accumulation operator (if provided)
 6927 are tested for domain compatibility as follows:

6928 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
 6929 $\mathbf{D}_{in_1}(\text{op})$ and $\mathbf{D}_{in_2}(\text{op})$ from F_b).

6930 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
 6931 $\mathbf{D}_{out}(\text{accum})$ of the accumulation operator, and $\mathbf{D}(\text{op})$ from M (or $\mathbf{D}_{out}(\text{op})$ from F_b) must
 6932 be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

6933 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).

6934 Two domains are compatible with each other if values from one domain can be cast to values in
 6935 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 6936 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 6937 any compatibility rule above is violated, execution of GrB_reduce ends and the domain mismatch
 6938 error listed above is returned.

6939 The number of values stored in the input, A , is checked. If there are no stored values in A , then
 6940 one of the following occurs depending on the output variant:

$$6941 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

6942 or

$$6943 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

6944 where $\mathbf{0}(\text{op})$ is the identity of the monoid. The operation returns immediately with GrB_SUCCESS.

6945 For all other cases, the internal matrix and scalar used in the computation is formed (\leftarrow denotes
 6946 copy):

6947 1. Matrix $\tilde{A} \leftarrow A$.

6948 2. Scalar $\tilde{s} \leftarrow s$ (GraphBLAS scalar case).

6949 We are now ready to carry out the reduce and any additional associated operations. An intermediate
6950 scalar result t is computed as follows:

$$6951 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

6952 where $\oplus = \odot(\text{op})$.

6953 The final reduction value is computed as follows:

$$6954 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

6955 or

$$6956 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

6957 In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value
6958 GrB_SUCCESS and the new contents of the output scalar is as defined above.

6959 4.3.11 transpose: Transpose rows and columns of a matrix

6960 This version computes a new matrix that is the transpose of the source matrix.

6961 C Syntax

```
6962      GrB_Info GrB_transpose(GrB_Matrix      C,
6963                           const GrB_Matrix Mask,
6964                           const GrB_BinaryOp accum,
6965                           const GrB_Matrix  A,
6966                           const GrB_Descriptor desc);
```

6967 Parameters

6968 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
6969 that may be accumulated with the result of the transpose operation. On output,
6970 the matrix holds the results of the operation.

6971 **Mask** (IN) An optional “write” mask that controls which results from this operation are
6972 stored into the output matrix C. The mask dimensions must match those of the
6973 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
6974 of the Mask matrix must be of type bool or any of the predefined “built-in” types
6975 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
6976 dimensions of C), GrB_NULL should be specified.

6977 **accum** (IN) An optional binary operator used for accumulating entries into existing C
6978 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
6979 specified.

6980 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

6981 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
6982 should be specified. Non-default field/value pairs are listed as follows:
6983

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.

6985 Return Values

6986 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
6987 blocking mode, this indicates that the compatibility tests on di-
6988 mensions and domains for the input arguments passed successfully.
6989 Either way, output matrix C is ready to be used in the next method
6990 of the sequence.

6991 **GrB_PANIC** Unknown internal error.

6992 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
6993 GraphBLAS objects (input or output) is in an invalid state caused
6994 by a previous execution error. Call **GrB_error()** to access any error
6995 messages generated by the implementation.

6996 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

6997 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
6998 a call to **new** (or **Matrix_dup** for matrix parameters).

6999 **GrB_DIMENSION_MISMATCH** mask, C and/or A dimensions are incompatible.

7000 **GrB_DOMAIN_MISMATCH** The domains of the various matrices are incompatible with the cor-
7001 responding domains of the accumulation operator, or the mask's do-
7002 main is not compatible with **bool** (in the case where **desc[GrB_MASK].GrB_STRUCT**
7003 is not set).

7004 Description

7005 GrB_transpose computes the result of performing a transpose of the input matrix: $C = A^T$; or, if an
 7006 optional binary accumulation operator (\odot) is provided, $C = C \odot A^T$. We note that the input matrix
 7007 A can itself be optionally transposed before the operation, which would cause either an assignment
 7008 from A to C or an accumulation of A into C.

7009 Logically, this operation occurs in three steps:

7010 **Setup** The internal matrix and mask used in the computation are formed and their domains
 7011 and dimensions are tested for compatibility.

7012 **Compute** The indicated computations are carried out.

7013 **Output** The result is written into the output matrix, possibly under control of a mask.

7014 Up to three matrix arguments are used in this GrB_transpose operation:

- 7015 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7016 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)
- 7017 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7018 The argument matrices and accumulation operator (if provided) are tested for domain compatibility
 7019 as follows:

- 7020 1. If Mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{Mask})$
 7021 must be from one of the pre-defined types of Table ??.
- 7022 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$ of the input matrix.
- 7023 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 7024 of the accumulation operator and $\mathbf{D}(A)$ of the input matrix must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 7025 of the accumulation operator.

7026 Two domains are compatible with each other if values from one domain can be cast to values in
 7027 the other domain as per the rules of the C language. In particular, domains from Table ?? are all
 7028 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 7029 any compatibility rule above is violated, execution of GrB_transpose ends and the domain mismatch
 7030 error listed above is returned.

7031 From the argument matrices, the internal matrices and mask used in the computation are formed
 7032 (\leftarrow denotes copy):

- 7033 1. Matrix $\tilde{C} \leftarrow C$.
- 7034 2. Two-dimensional mask, \tilde{M} , is computed from argument Mask as follows:

- 7035 (a) If $\text{Mask} = \text{GrB_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
7036 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 7037 (b) If $\text{Mask} \neq \text{GrB_NULL}$,
- 7038 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
7039 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
- 7040 ii. Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
7041 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
- 7042 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$.
- 7043 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

7044 The internal matrices and masks are checked for dimension compatibility. The following conditions
7045 must hold:

- 7046 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
- 7047 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
- 7048 3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.
- 7049 4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

7050 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension
7051 mismatch error listed above is returned.

7052 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
7053 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7054 We are now ready to carry out the matrix transposition and any additional associated operations.
7055 We describe this in terms of two intermediate matrices:

- 7056 • $\widetilde{\mathbf{T}}$: The matrix holding the transpose of $\widetilde{\mathbf{A}}$.
- 7057 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

7058 The intermediate matrix

$$7059 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7060 is created.

7061 The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7062 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- 7063 • If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$7064 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

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

$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

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

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

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

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

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

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

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

4.3.12 kronecker: Kronecker product of two matrices

Computes the Kronecker product of two matrices. The result is a matrix.

C Syntax

```
GrB_Info GrB_kronecker(GrB_Matrix      C,
                        const GrB_Matrix Mask,
                        const GrB_BinaryOp accum,
                        const GrB_Semiring op,
                        const GrB_Matrix A,
                        const GrB_Matrix B,
                        const GrB_Descriptor desc);
```

```

7099     GrB_Info GrB_kronecker(GrB_Matrix      C,
7100                           const GrB_Matrix  Mask,
7101                           const GrB_BinaryOp accum,
7102                           const GrB_Monoid   op,
7103                           const GrB_Matrix  A,
7104                           const GrB_Matrix  B,
7105                           const GrB_Descriptor desc);
7106
7107     GrB_Info GrB_kronecker(GrB_Matrix      C,
7108                           const GrB_Matrix  Mask,
7109                           const GrB_BinaryOp accum,
7110                           const GrB_BinaryOp op,
7111                           const GrB_Matrix  A,
7112                           const GrB_Matrix  B,
7113                           const GrB_Descriptor desc);

```

7114 Parameters

7115 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
7116 that may be accumulated with the result of the Kronecker product. On output,
7117 the matrix holds the results of the operation.

7118 **Mask** (IN) An optional “write” mask that controls which results from this operation are
7119 stored into the output matrix C. The mask dimensions must match those of the
7120 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
7121 of the Mask matrix must be of type bool or any of the predefined “built-in” types
7122 in Table ???. If the default mask is desired (i.e., a mask that is all true with the
7123 dimensions of C), GrB_NULL should be specified.

7124 **accum** (IN) An optional binary operator used for accumulating entries into existing C
7125 entries. If assignment rather than accumulation is desired, GrB_NULL should be
7126 specified.

7127 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
7128 operation. Depending on which type is passed, the following defines the binary
7129 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

7130 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

7131 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-
7132 nored.

7133 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
7134 is ignored.

7135 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
7136 product.

7137 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
7138 product.

7139 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
7140 should be specified. Non-default field/value pairs are listed as follows:
7141

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.

7143 Return Values

7144 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
7145 blocking mode, this indicates that the compatibility tests on di-
7146 mensions and domains for the input arguments passed successfully.
7147 Either way, output matrix C is ready to be used in the next method
7148 of the sequence.

7149 GrB_PANIC Unknown internal error.

7150 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
7151 GraphBLAS objects (input or output) is in an invalid state caused
7152 by a previous execution error. Call GrB_error() to access any error
7153 messages generated by the implementation.

7154 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

7155 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
7156 a call to new (or Matrix_dup for matrix parameters).

7157 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

7158 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
7159 corresponding domains of the binary operator (op) or accumulation
7160 operator, or the mask's domain is not compatible with bool (in the
7161 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

7162 Description

7163 GrB_kronecker computes the Kronecker product $C = A \otimes B$ or, if an optional binary accumulation
7164 operator (\odot) is provided, $C = C \odot (A \otimes B)$ (where matrices A and B can be optionally transposed).

7165 The Kronecker product is defined as follows:

7166

$$7167 \quad \mathbf{C} = \mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{0,0} \otimes \mathbf{B} & A_{0,1} \otimes \mathbf{B} & \dots & A_{0,n_A-1} \otimes \mathbf{B} \\ A_{1,0} \otimes \mathbf{B} & A_{1,1} \otimes \mathbf{B} & \dots & A_{1,n_A-1} \otimes \mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes \mathbf{B} & A_{m_A-1,1} \otimes \mathbf{B} & \dots & A_{m_A-1,n_A-1} \otimes \mathbf{B} \end{bmatrix}$$

7168 where $\mathbf{A} : \mathbb{S}^{m_A \times n_A}$, $\mathbf{B} : \mathbb{S}^{m_B \times n_B}$, and $\mathbf{C} : \mathbb{S}^{m_A m_B \times n_A n_B}$. More explicitly, the elements of the
7169 Kronecker product are defined as

$$7170 \quad \mathbf{C}(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7171 where \otimes is the multiplicative operator specified by the **op** parameter.

7172 Logically, this operation occurs in three steps:

7173 **Setup** The internal matrices and mask used in the computation are formed and their domains
7174 and dimensions are tested for compatibility.

7175 **Compute** The indicated computations are carried out.

7176 **Output** The result is written into the output matrix, possibly under control of a mask.

7177 Up to four argument matrices are used in the **GrB_kronecker** operation:

- 7178 1. $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 7179 2. $\mathbf{Mask} = \langle \mathbf{D}(\mathbf{Mask}), \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \mathbf{L}(\mathbf{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 7180 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 7181 4. $\mathbf{B} = \langle \mathbf{D}(\mathbf{B}), \mathbf{nrows}(\mathbf{B}), \mathbf{ncols}(\mathbf{B}), \mathbf{L}(\mathbf{B}) = \{(i, j, B_{ij})\} \rangle$

7182 The argument matrices, the "product" operator (**op**), and the accumulation operator (if provided)
7183 are tested for domain compatibility as follows:

- 7184 1. If **Mask** is not **GrB_NULL**, and **desc[GrB_MASK].GrB_STRUCTURE** is not set, then $\mathbf{D}(\mathbf{Mask})$
7185 must be from one of the pre-defined types of Table ??.
- 7186 2. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 7187 3. $\mathbf{D}(\mathbf{B})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 7188 4. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 7189 5. If **accum** is not **GrB_NULL**, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
7190 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of **op** must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
7191 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 ?? 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.

7225 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$
7226 $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$ is created. The value of
7227 each of its elements is computed by

$$7228 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7229 where \otimes is the multiplicative operator specified by the `op` parameter.

7230 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 7231 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 7232 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$7233 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7234 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
7235 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$7236 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7237 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7239 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7241 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

7242 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
7243 using what is called a *standard matrix mask and replace*. This is carried out under control of the
7244 mask which acts as a “write mask”.

- 7245 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
7246 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$7247 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7248 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
7249 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
7250 mask are unchanged:

$$7251 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7252 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
7253 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
7254 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
7255 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
7256 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 ?? through ??.

Table 5.1: Long-name, nonpolymorphic form of GraphBLAS methods.

Polymorphic signature	Nonpolymorphic signature
GrB_Monoid_new(GrB_Monoid*,...,bool)	GrB_Monoid_new_BOOL(GrB_Monoid*,GrB_BinaryOp,bool)
GrB_Monoid_new(GrB_Monoid*,...,int8_t)	GrB_Monoid_new_INT8(GrB_Monoid*,GrB_BinaryOp,int8_t)
GrB_Monoid_new(GrB_Monoid*,...,uint8_t)	GrB_Monoid_new_UINT8(GrB_Monoid*,GrB_BinaryOp,uint8_t)
GrB_Monoid_new(GrB_Monoid*,...,int16_t)	GrB_Monoid_new_INT16(GrB_Monoid*,GrB_BinaryOp,int16_t)
GrB_Monoid_new(GrB_Monoid*,...,uint16_t)	GrB_Monoid_new_UINT16(GrB_Monoid*,GrB_BinaryOp,uint16_t)
GrB_Monoid_new(GrB_Monoid*,...,int32_t)	GrB_Monoid_new_INT32(GrB_Monoid*,GrB_BinaryOp,int32_t)
GrB_Monoid_new(GrB_Monoid*,...,uint32_t)	GrB_Monoid_new_UINT32(GrB_Monoid*,GrB_BinaryOp,uint32_t)
GrB_Monoid_new(GrB_Monoid*,...,int64_t)	GrB_Monoid_new_INT64(GrB_Monoid*,GrB_BinaryOp,int64_t)
GrB_Monoid_new(GrB_Monoid*,...,uint64_t)	GrB_Monoid_new_UINT64(GrB_Monoid*,GrB_BinaryOp,uint64_t)
GrB_Monoid_new(GrB_Monoid*,...,float)	GrB_Monoid_new_FP32(GrB_Monoid*,GrB_BinaryOp,float)
GrB_Monoid_new(GrB_Monoid*,...,double)	GrB_Monoid_new_FP64(GrB_Monoid*,GrB_BinaryOp,double)
GrB_Monoid_new(GrB_Monoid*,...,other)	GrB_Monoid_new_UDT(GrB_Monoid*,GrB_BinaryOp,void*)

Table 5.2: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Scalar_setElement(..., bool,...)	GrB_Scalar_setElement_BOOL(..., bool,...)
GrB_Scalar_setElement(..., int8_t,...)	GrB_Scalar_setElement_INT8(..., int8_t,...)
GrB_Scalar_setElement(..., uint8_t,...)	GrB_Scalar_setElement_UINT8(..., uint8_t,...)
GrB_Scalar_setElement(..., int16_t,...)	GrB_Scalar_setElement_INT16(..., int16_t,...)
GrB_Scalar_setElement(..., uint16_t,...)	GrB_Scalar_setElement_UINT16(..., uint16_t,...)
GrB_Scalar_setElement(..., int32_t,...)	GrB_Scalar_setElement_INT32(..., int32_t,...)
GrB_Scalar_setElement(..., uint32_t,...)	GrB_Scalar_setElement_UINT32(..., uint32_t,...)
GrB_Scalar_setElement(..., int64_t,...)	GrB_Scalar_setElement_INT64(..., int64_t,...)
GrB_Scalar_setElement(..., uint64_t,...)	GrB_Scalar_setElement_UINT64(..., uint64_t,...)
GrB_Scalar_setElement(..., float,...)	GrB_Scalar_setElement_FP32(..., float,...)
GrB_Scalar_setElement(..., double,...)	GrB_Scalar_setElement_FP64(..., double,...)
GrB_Scalar_setElement(..., <i>other</i> ,...)	GrB_Scalar_setElement_UDT(..., const void*,...)
GrB_Scalar_extractElement(bool*,...)	GrB_Scalar_extractElement_BOOL(bool*,...)
GrB_Scalar_extractElement(int8_t*,...)	GrB_Scalar_extractElement_INT8(int8_t*,...)
GrB_Scalar_extractElement(uint8_t*,...)	GrB_Scalar_extractElement_UINT8(uint8_t*,...)
GrB_Scalar_extractElement(int16_t*,...)	GrB_Scalar_extractElement_INT16(int16_t*,...)
GrB_Scalar_extractElement(uint16_t*,...)	GrB_Scalar_extractElement_UINT16(uint16_t*,...)
GrB_Scalar_extractElement(int32_t*,...)	GrB_Scalar_extractElement_INT32(int32_t*,...)
GrB_Scalar_extractElement(uint32_t*,...)	GrB_Scalar_extractElement_UINT32(uint32_t*,...)
GrB_Scalar_extractElement(int64_t*,...)	GrB_Scalar_extractElement_INT64(int64_t*,...)
GrB_Scalar_extractElement(uint64_t*,...)	GrB_Scalar_extractElement_UINT64(uint64_t*,...)
GrB_Scalar_extractElement(float*,...)	GrB_Scalar_extractElement_FP32(float*,...)
GrB_Scalar_extractElement(double*,...)	GrB_Scalar_extractElement_FP64(double*,...)
GrB_Scalar_extractElement(<i>other</i> *,...)	GrB_Scalar_extractElement_UDT(void*,...)

Table 5.3: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Vector_build(...,const bool*,...)	GrB_Vector_build_BOOL(...,const bool*,...)
GrB_Vector_build(...,const int8_t*,...)	GrB_Vector_build_INT8(...,const int8_t*,...)
GrB_Vector_build(...,const uint8_t*,...)	GrB_Vector_build_UINT8(...,const uint8_t*,...)
GrB_Vector_build(...,const int16_t*,...)	GrB_Vector_build_INT16(...,const int16_t*,...)
GrB_Vector_build(...,const uint16_t*,...)	GrB_Vector_build_UINT16(...,const uint16_t*,...)
GrB_Vector_build(...,const int32_t*,...)	GrB_Vector_build_INT32(...,const int32_t*,...)
GrB_Vector_build(...,const uint32_t*,...)	GrB_Vector_build_UINT32(...,const uint32_t*,...)
GrB_Vector_build(...,const int64_t*,...)	GrB_Vector_build_INT64(...,const int64_t*,...)
GrB_Vector_build(...,const uint64_t*,...)	GrB_Vector_build_UINT64(...,const uint64_t*,...)
GrB_Vector_build(...,const float*,...)	GrB_Vector_build_FP32(...,const float*,...)
GrB_Vector_build(...,const double*,...)	GrB_Vector_build_FP64(...,const double*,...)
GrB_Vector_build(...,const <i>other</i> *,...)	GrB_Vector_build_UDT(...,const void*,...)
GrB_Vector_setElement(...,GrB_Scalar,...)	GrB_Vector_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Vector_setElement(...,bool,...)	GrB_Vector_setElement_BOOL(..., bool,...)
GrB_Vector_setElement(...,int8_t,...)	GrB_Vector_setElement_INT8(..., int8_t,...)
GrB_Vector_setElement(...,uint8_t,...)	GrB_Vector_setElement_UINT8(..., uint8_t,...)
GrB_Vector_setElement(...,int16_t,...)	GrB_Vector_setElement_INT16(..., int16_t,...)
GrB_Vector_setElement(...,uint16_t,...)	GrB_Vector_setElement_UINT16(..., uint16_t,...)
GrB_Vector_setElement(...,int32_t,...)	GrB_Vector_setElement_INT32(..., int32_t,...)
GrB_Vector_setElement(...,uint32_t,...)	GrB_Vector_setElement_UINT32(..., uint32_t,...)
GrB_Vector_setElement(...,int64_t,...)	GrB_Vector_setElement_INT64(..., int64_t,...)
GrB_Vector_setElement(...,uint64_t,...)	GrB_Vector_setElement_UINT64(..., uint64_t,...)
GrB_Vector_setElement(...,float,...)	GrB_Vector_setElement_FP32(..., float,...)
GrB_Vector_setElement(...,double,...)	GrB_Vector_setElement_FP64(..., double,...)
GrB_Vector_setElement(..., <i>other</i> ,...)	GrB_Vector_setElement_UDT(...,const void*,...)
GrB_Vector_extractElement(GrB_Scalar,...)	GrB_Vector_extractElement_Scalar(GrB_Scalar,...)
GrB_Vector_extractElement(bool*,...)	GrB_Vector_extractElement_BOOL(bool*,...)
GrB_Vector_extractElement(int8_t*,...)	GrB_Vector_extractElement_INT8(int8_t*,...)
GrB_Vector_extractElement(uint8_t*,...)	GrB_Vector_extractElement_UINT8(uint8_t*,...)
GrB_Vector_extractElement(int16_t*,...)	GrB_Vector_extractElement_INT16(int16_t*,...)
GrB_Vector_extractElement(uint16_t*,...)	GrB_Vector_extractElement_UINT16(uint16_t*,...)
GrB_Vector_extractElement(int32_t*,...)	GrB_Vector_extractElement_INT32(int32_t*,...)
GrB_Vector_extractElement(uint32_t*,...)	GrB_Vector_extractElement_UINT32(uint32_t*,...)
GrB_Vector_extractElement(int64_t*,...)	GrB_Vector_extractElement_INT64(int64_t*,...)
GrB_Vector_extractElement(uint64_t*,...)	GrB_Vector_extractElement_UINT64(uint64_t*,...)
GrB_Vector_extractElement(float*,...)	GrB_Vector_extractElement_FP32(float*,...)
GrB_Vector_extractElement(double*,...)	GrB_Vector_extractElement_FP64(double*,...)
GrB_Vector_extractElement(<i>other</i> *,...)	GrB_Vector_extractElement_UDT(void*,...)
GrB_Vector_extractTuples(...,bool*,...)	GrB_Vector_extractTuples_BOOL(..., bool*,...)
GrB_Vector_extractTuples(...,int8_t*,...)	GrB_Vector_extractTuples_INT8(..., int8_t*,...)
GrB_Vector_extractTuples(...,uint8_t*,...)	GrB_Vector_extractTuples_UINT8(..., uint8_t*,...)
GrB_Vector_extractTuples(...,int16_t*,...)	GrB_Vector_extractTuples_INT16(..., int16_t*,...)
GrB_Vector_extractTuples(...,uint16_t*,...)	GrB_Vector_extractTuples_UINT16(..., uint16_t*,...)
GrB_Vector_extractTuples(...,int32_t*,...)	GrB_Vector_extractTuples_INT32(..., int32_t*,...)
GrB_Vector_extractTuples(...,uint32_t*,...)	GrB_Vector_extractTuples_UINT32(..., uint32_t*,...)
GrB_Vector_extractTuples(...,int64_t*,...)	GrB_Vector_extractTuples_INT64(..., int64_t*,...)
GrB_Vector_extractTuples(...,uint64_t*,...)	GrB_Vector_extractTuples_UINT64(..., uint64_t*,...)
GrB_Vector_extractTuples(...,float*,...)	GrB_Vector_extractTuples_FP32(..., float*,...)
GrB_Vector_extractTuples(...,double*,...)	GrB_Vector_extractTuples_FP64(..., double*,...)
GrB_Vector_extractTuples(..., <i>other</i> *,...)	GrB_Vector_extractTuples_UDT(..., void*,...)

Table 5.4: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_build(...,const bool*,...)	GrB_Matrix_build_BOOL(...,const bool*,...)
GrB_Matrix_build(...,const int8_t*,...)	GrB_Matrix_build_INT8(...,const int8_t*,...)
GrB_Matrix_build(...,const uint8_t*,...)	GrB_Matrix_build_UINT8(...,const uint8_t*,...)
GrB_Matrix_build(...,const int16_t*,...)	GrB_Matrix_build_INT16(...,const int16_t*,...)
GrB_Matrix_build(...,const uint16_t*,...)	GrB_Matrix_build_UINT16(...,const uint16_t*,...)
GrB_Matrix_build(...,const int32_t*,...)	GrB_Matrix_build_INT32(...,const int32_t*,...)
GrB_Matrix_build(...,const uint32_t*,...)	GrB_Matrix_build_UINT32(...,const uint32_t*,...)
GrB_Matrix_build(...,const int64_t*,...)	GrB_Matrix_build_INT64(...,const int64_t*,...)
GrB_Matrix_build(...,const uint64_t*,...)	GrB_Matrix_build_UINT64(...,const uint64_t*,...)
GrB_Matrix_build(...,const float*,...)	GrB_Matrix_build_FP32(...,const float*,...)
GrB_Matrix_build(...,const double*,...)	GrB_Matrix_build_FP64(...,const double*,...)
GrB_Matrix_build(...,const <i>other</i> *,...)	GrB_Matrix_build_UDT(...,const void*,...)
GrB_Matrix_setElement(...,GrB_Scalar,...)	GrB_Matrix_setElement_Scalar(...,const GrB_Scalar,...)
GrB_Matrix_setElement(...,bool,...)	GrB_Matrix_setElement_BOOL(..., bool,...)
GrB_Matrix_setElement(...,int8_t,...)	GrB_Matrix_setElement_INT8(..., int8_t,...)
GrB_Matrix_setElement(...,uint8_t,...)	GrB_Matrix_setElement_UINT8(..., uint8_t,...)
GrB_Matrix_setElement(...,int16_t,...)	GrB_Matrix_setElement_INT16(..., int16_t,...)
GrB_Matrix_setElement(...,uint16_t,...)	GrB_Matrix_setElement_UINT16(..., uint16_t,...)
GrB_Matrix_setElement(...,int32_t,...)	GrB_Matrix_setElement_INT32(..., int32_t,...)
GrB_Matrix_setElement(...,uint32_t,...)	GrB_Matrix_setElement_UINT32(..., uint32_t,...)
GrB_Matrix_setElement(...,int64_t,...)	GrB_Matrix_setElement_INT64(..., int64_t,...)
GrB_Matrix_setElement(...,uint64_t,...)	GrB_Matrix_setElement_UINT64(..., uint64_t,...)
GrB_Matrix_setElement(...,float,...)	GrB_Matrix_setElement_FP32(..., float,...)
GrB_Matrix_setElement(...,double,...)	GrB_Matrix_setElement_FP64(..., double,...)
GrB_Matrix_setElement(..., <i>other</i> ,...)	GrB_Matrix_setElement_UDT(...,const void*,...)
GrB_Matrix_extractElement(GrB_Scalar,...)	GrB_Matrix_extractElement_Scalar(GrB_Scalar,...)
GrB_Matrix_extractElement(bool*,...)	GrB_Matrix_extractElement_BOOL(bool*,...)
GrB_Matrix_extractElement(int8_t*,...)	GrB_Matrix_extractElement_INT8(int8_t*,...)
GrB_Matrix_extractElement(uint8_t*,...)	GrB_Matrix_extractElement_UINT8(uint8_t*,...)
GrB_Matrix_extractElement(int16_t*,...)	GrB_Matrix_extractElement_INT16(int16_t*,...)
GrB_Matrix_extractElement(uint16_t*,...)	GrB_Matrix_extractElement_UINT16(uint16_t*,...)
GrB_Matrix_extractElement(int32_t*,...)	GrB_Matrix_extractElement_INT32(int32_t*,...)
GrB_Matrix_extractElement(uint32_t*,...)	GrB_Matrix_extractElement_UINT32(uint32_t*,...)
GrB_Matrix_extractElement(int64_t*,...)	GrB_Matrix_extractElement_INT64(int64_t*,...)
GrB_Matrix_extractElement(uint64_t*,...)	GrB_Matrix_extractElement_UINT64(uint64_t*,...)
GrB_Matrix_extractElement(float*,...)	GrB_Matrix_extractElement_FP32(float*,...)
GrB_Matrix_extractElement(double*,...)	GrB_Matrix_extractElement_FP64(double*,...)
GrB_Matrix_extractElement(<i>other</i> ,...)	GrB_Matrix_extractElement_UDT(void*,...)
GrB_Matrix_extractTuples(..., bool*,...)	GrB_Matrix_extractTuples_BOOL(..., bool*,...)
GrB_Matrix_extractTuples(..., int8_t*,...)	GrB_Matrix_extractTuples_INT8(..., int8_t*,...)
GrB_Matrix_extractTuples(..., uint8_t*,...)	GrB_Matrix_extractTuples_UINT8(..., uint8_t*,...)
GrB_Matrix_extractTuples(..., int16_t*,...)	GrB_Matrix_extractTuples_INT16(..., int16_t*,...)
GrB_Matrix_extractTuples(..., uint16_t*,...)	GrB_Matrix_extractTuples_UINT16(..., uint16_t*,...)
GrB_Matrix_extractTuples(..., int32_t*,...)	GrB_Matrix_extractTuples_INT32(..., int32_t*,...)
GrB_Matrix_extractTuples(..., uint32_t*,...)	GrB_Matrix_extractTuples_UINT32(..., uint32_t*,...)
GrB_Matrix_extractTuples(..., int64_t*,...)	GrB_Matrix_extractTuples_INT64(..., int64_t*,...)
GrB_Matrix_extractTuples(..., uint64_t*,...)	GrB_Matrix_extractTuples_UINT64(..., uint64_t*,...)
GrB_Matrix_extractTuples(..., float*,...)	GrB_Matrix_extractTuples_FP32(..., float*,...)
GrB_Matrix_extractTuples(..., double*,...)	GrB_Matrix_extractTuples_FP64(..., double*,...)
GrB_Matrix_extractTuples(..., <i>other</i> *,...)	GrB_Matrix_extractTuples_UDT(..., void*,...)

Table 5.5: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_import(...,const bool*,...)	GrB_Matrix_import_BOOL(...,const bool*,...)
GrB_Matrix_import(...,const int8_t*,...)	GrB_Matrix_import_INT8(...,const int8_t*,...)
GrB_Matrix_import(...,const uint8_t*,...)	GrB_Matrix_import_UINT8(...,const uint8_t*,...)
GrB_Matrix_import(...,const int16_t*,...)	GrB_Matrix_import_INT16(...,const int16_t*,...)
GrB_Matrix_import(...,const uint16_t*,...)	GrB_Matrix_import_UINT16(...,const uint16_t*,...)
GrB_Matrix_import(...,const int32_t*,...)	GrB_Matrix_import_INT32(...,const int32_t*,...)
GrB_Matrix_import(...,const uint32_t*,...)	GrB_Matrix_import_UINT32(...,const uint32_t*,...)
GrB_Matrix_import(...,const int64_t*,...)	GrB_Matrix_import_INT64(...,const int64_t*,...)
GrB_Matrix_import(...,const uint64_t*,...)	GrB_Matrix_import_UINT64(...,const uint64_t*,...)
GrB_Matrix_import(...,const float*,...)	GrB_Matrix_import_FP32(...,const float*,...)
GrB_Matrix_import(...,const double*,...)	GrB_Matrix_import_FP64(...,const double*,...)
GrB_Matrix_import(...,const other,...)	GrB_Matrix_import_UDT(...,const void*,...)
GrB_Matrix_export(...,bool*,...)	GrB_Matrix_export_BOOL(...,bool*,...)
GrB_Matrix_export(...,int8_t*,...)	GrB_Matrix_export_INT8(...,int8_t*,...)
GrB_Matrix_export(...,uint8_t*,...)	GrB_Matrix_export_UINT8(...,uint8_t*,...)
GrB_Matrix_export(...,int16_t*,...)	GrB_Matrix_export_INT16(...,int16_t*,...)
GrB_Matrix_export(...,uint16_t*,...)	GrB_Matrix_export_UINT16(...,uint16_t*,...)
GrB_Matrix_export(...,int32_t*,...)	GrB_Matrix_export_INT32(...,int32_t*,...)
GrB_Matrix_export(...,uint32_t*,...)	GrB_Matrix_export_UINT32(...,uint32_t*,...)
GrB_Matrix_export(...,int64_t*,...)	GrB_Matrix_export_INT64(...,int64_t*,...)
GrB_Matrix_export(...,uint64_t*,...)	GrB_Matrix_export_UINT64(...,uint64_t*,...)
GrB_Matrix_export(...,float*,...)	GrB_Matrix_export_FP32(...,float*,...)
GrB_Matrix_export(...,double*,...)	GrB_Matrix_export_FP64(...,double*,...)
GrB_Matrix_export(...,other,...)	GrB_Matrix_export_UDT(...,void*,...)
GrB_free(GrB_Type*)	GrB_Type_free(GrB_Type*)
GrB_free(GrB_UnaryOp*)	GrB_UnaryOp_free(GrB_UnaryOp*)
GrB_free(GrB_IndexUnaryOp*)	GrB_IndexUnaryOp_free(GrB_IndexUnaryOp*)
GrB_free(GrB_BinaryOp*)	GrB_BinaryOp_free(GrB_BinaryOp*)
GrB_free(GrB_Monoid*)	GrB_Monoid_free(GrB_Monoid*)
GrB_free(GrB_Semiring*)	GrB_Semiring_free(GrB_Semiring*)
GrB_free(GrB_Scalar*)	GrB_Scalar_free(GrB_Scalar*)
GrB_free(GrB_Vector*)	GrB_Vector_free(GrB_Vector*)
GrB_free(GrB_Matrix*)	GrB_Matrix_free(GrB_Matrix*)
GrB_free(GrB_Descriptor*)	GrB_Descriptor_free(GrB_Descriptor*)
GrB_wait(GrB_Type, GrB_WaitMode)	GrB_Type_wait(GrB_Type, GrB_WaitMode)
GrB_wait(GrB_UnaryOp, GrB_WaitMode)	GrB_UnaryOp_wait(GrB_UnaryOp, GrB_WaitMode)
GrB_wait(GrB_IndexUnaryOp, GrB_WaitMode)	GrB_IndexUnaryOp_wait(GrB_IndexUnaryOp, GrB_WaitMode)
GrB_wait(GrB_BinaryOp, GrB_WaitMode)	GrB_BinaryOp_wait(GrB_BinaryOp, GrB_WaitMode)
GrB_wait(GrB_Monoid, GrB_WaitMode)	GrB_Monoid_wait(GrB_Monoid, GrB_WaitMode)
GrB_wait(GrB_Semiring, GrB_WaitMode)	GrB_Semiring_wait(GrB_Semiring, GrB_WaitMode)
GrB_wait(GrB_Scalar, GrB_WaitMode)	GrB_Scalar_wait(GrB_Scalar, GrB_WaitMode)
GrB_wait(GrB_Vector, GrB_WaitMode)	GrB_Vector_wait(GrB_Vector, GrB_WaitMode)
GrB_wait(GrB_Matrix, GrB_WaitMode)	GrB_Matrix_wait(GrB_Matrix, GrB_WaitMode)
GrB_wait(GrB_Descriptor, GrB_WaitMode)	GrB_Descriptor_wait(GrB_Descriptor, GrB_WaitMode)
GrB_error(const char**, const GrB_Type)	GrB_Type_error(const char**, const GrB_Type)
GrB_error(const char**, const GrB_UnaryOp)	GrB_UnaryOp_error(const char**, const GrB_UnaryOp)
GrB_error(const char**, const GrB_IndexUnaryOp)	GrB_IndexUnaryOp_error(const char**, const GrB_IndexUnaryOp)
GrB_error(const char**, const GrB_BinaryOp)	GrB_BinaryOp_error(const char**, const GrB_BinaryOp)
GrB_error(const char**, const GrB_Monoid)	GrB_Monoid_error(const char**, const GrB_Monoid)
GrB_error(const char**, const GrB_Semiring)	GrB_Semiring_error(const char**, const GrB_Semiring)
GrB_error(const char**, const GrB_Scalar)	GrB_Scalar_error(const char**, const GrB_Scalar)
GrB_error(const char**, const GrB_Vector)	GrB_Vector_error(const char**, const GrB_Vector)
GrB_error(const char**, const GrB_Matrix)	GrB_Matrix_error(const char**, const GrB_Matrix)
GrB_error(const char**, const GrB_Descriptor)	GrB_Descriptor_error(const char**, const GrB_Descriptor)

Table 5.6: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_eWiseMult(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseMult_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseMult_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseMult_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseMult_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseMult_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseMult_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseAdd_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseAdd_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseAdd_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseAdd_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseAdd_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseAdd_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_extract(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_extract(GrB_Vector,...,GrB_Vector,...)
GrB_extract(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_extract(GrB_Matrix,...,GrB_Matrix,...)
GrB_extract(GrB_Vector,...,GrB_Matrix,...)	GrB_Col_extract(GrB_Vector,...,GrB_Matrix,...)
GrB_assign(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_assign(GrB_Vector,...,GrB_Vector,...)
GrB_assign(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_assign(GrB_Matrix,...,GrB_Matrix,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)	GrB_Col_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)	GrB_Row_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)
GrB_assign(GrB_Vector,...,GrB_Scalar,...)	GrB_Vector_assign_Scalar(GrB_Vector,...,const GrB_Scalar,...)
GrB_assign(GrB_Vector,...,bool,...)	GrB_Vector_assign_BOOL(GrB_Vector,..., bool,...)
GrB_assign(GrB_Vector,...,int8_t,...)	GrB_Vector_assign_INT8(GrB_Vector,..., int8_t,...)
GrB_assign(GrB_Vector,...,uint8_t,...)	GrB_Vector_assign_UINT8(GrB_Vector,..., uint8_t,...)
GrB_assign(GrB_Vector,...,int16_t,...)	GrB_Vector_assign_INT16(GrB_Vector,..., int16_t,...)
GrB_assign(GrB_Vector,...,uint16_t,...)	GrB_Vector_assign_UINT16(GrB_Vector,..., uint16_t,...)
GrB_assign(GrB_Vector,...,int32_t,...)	GrB_Vector_assign_INT32(GrB_Vector,..., int32_t,...)
GrB_assign(GrB_Vector,...,uint32_t,...)	GrB_Vector_assign_UINT32(GrB_Vector,..., uint32_t,...)
GrB_assign(GrB_Vector,...,int64_t,...)	GrB_Vector_assign_INT64(GrB_Vector,..., int64_t,...)
GrB_assign(GrB_Vector,...,uint64_t,...)	GrB_Vector_assign_UINT64(GrB_Vector,..., uint64_t,...)
GrB_assign(GrB_Vector,...,float,...)	GrB_Vector_assign_FP32(GrB_Vector,..., float,...)
GrB_assign(GrB_Vector,...,double,...)	GrB_Vector_assign_FP64(GrB_Vector,..., double,...)
GrB_assign(GrB_Vector,...,other,...)	GrB_Vector_assign_UDT(GrB_Vector,...,const void*,...)
GrB_assign(GrB_Matrix,...,GrB_Scalar,...)	GrB_Matrix_assign_Scalar(GrB_Matrix,...,const GrB_Scalar,...)
GrB_assign(GrB_Matrix,...,bool,...)	GrB_Matrix_assign_BOOL(GrB_Matrix,..., bool,...)
GrB_assign(GrB_Matrix,...,int8_t,...)	GrB_Matrix_assign_INT8(GrB_Matrix,..., int8_t,...)
GrB_assign(GrB_Matrix,...,uint8_t,...)	GrB_Matrix_assign_UINT8(GrB_Matrix,..., uint8_t,...)
GrB_assign(GrB_Matrix,...,int16_t,...)	GrB_Matrix_assign_INT16(GrB_Matrix,..., int16_t,...)
GrB_assign(GrB_Matrix,...,uint16_t,...)	GrB_Matrix_assign_UINT16(GrB_Matrix,..., uint16_t,...)
GrB_assign(GrB_Matrix,...,int32_t,...)	GrB_Matrix_assign_INT32(GrB_Matrix,..., int32_t,...)
GrB_assign(GrB_Matrix,...,uint32_t,...)	GrB_Matrix_assign_UINT32(GrB_Matrix,..., uint32_t,...)
GrB_assign(GrB_Matrix,...,int64_t,...)	GrB_Matrix_assign_INT64(GrB_Matrix,..., int64_t,...)
GrB_assign(GrB_Matrix,...,uint64_t,...)	GrB_Matrix_assign_UINT64(GrB_Matrix,..., uint64_t,...)
GrB_assign(GrB_Matrix,...,float,...)	GrB_Matrix_assign_FP32(GrB_Matrix,..., float,...)
GrB_assign(GrB_Matrix,...,double,...)	GrB_Matrix_assign_FP64(GrB_Matrix,..., double,...)
GrB_assign(GrB_Matrix,...,other,...)	GrB_Matrix_assign_UDT(GrB_Matrix,...,const void*,...)

Table 5.7: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)	GrB_Vector_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)
GrB_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)	GrB_Matrix_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_BOOL(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT8(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT8(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT16(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT16(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT32(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT32(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_INT64(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UINT64(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP32(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_FP64(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp, <i>other</i> ,GrB_Vector,...)	GrB_Vector_apply_BinaryOp1st_UDT(GrB_Vector,...,GrB_BinaryOp,const void*,GrB_Vector,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_BinaryOp2nd_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_BinaryOp2nd_BOOL(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_BinaryOp2nd_INT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_BinaryOp2nd_INT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_BinaryOp2nd_INT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_BinaryOp2nd_INT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_BinaryOp2nd_UINT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)	GrB_Vector_apply_BinaryOp2nd_FP32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)	GrB_Vector_apply_BinaryOp2nd_FP64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_BinaryOp2nd_UDT(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,const void*,...)

Table 5.8: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_BOOL(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT8(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT8(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT16(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT16(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT32(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT32(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_INT64(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UINT64(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP32(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_FP64(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp, <i>other</i> ,GrB_Matrix,...)	GrB_Matrix_apply_BinaryOp1st_UDT(GrB_Matrix,...,GrB_BinaryOp,const void*,GrB_Matrix,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_BinaryOp2nd_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_BinaryOp2nd_BOOL(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_BinaryOp2nd_INT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_BinaryOp2nd_UINT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_BinaryOp2nd_FP32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_BinaryOp2nd_FP64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_BinaryOp2nd_UDT(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,const void*,...)

Table 5.9: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)	GrB_Vector_apply_IndexOp_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)	GrB_Vector_apply_IndexOp_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)	GrB_Vector_apply_IndexOp_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)	GrB_Vector_apply_IndexOp_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)	GrB_Vector_apply_IndexOp_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)	GrB_Vector_apply_IndexOp_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)	GrB_Vector_apply_IndexOp_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)	GrB_Vector_apply_IndexOp_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)	GrB_Vector_apply_IndexOp_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)	GrB_Vector_apply_IndexOp_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)	GrB_Vector_apply_IndexOp_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)	GrB_Vector_apply_IndexOp_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)
GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector, <i>other</i> ,...)	GrB_Vector_apply_IndexOp_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)	GrB_Matrix_apply_IndexOp_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)	GrB_Matrix_apply_IndexOp_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)	GrB_Matrix_apply_IndexOp_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)	GrB_Matrix_apply_IndexOp_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)	GrB_Matrix_apply_IndexOp_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)	GrB_Matrix_apply_IndexOp_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)	GrB_Matrix_apply_IndexOp_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)	GrB_Matrix_apply_IndexOp_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)	GrB_Matrix_apply_IndexOp_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)	GrB_Matrix_apply_IndexOp_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)	GrB_Matrix_apply_IndexOp_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)	GrB_Matrix_apply_IndexOp_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)
GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix, <i>other</i> ,...)	GrB_Matrix_apply_IndexOp_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)

Table 5.10: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

Polymorphic signature	Nonpolymorphic signature
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>	<code>GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>	<code>GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>	<code>GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>	<code>GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>	<code>GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>	<code>GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>	<code>GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>	<code>GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>	<code>GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>	<code>GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>	<code>GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>	<code>GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>
<code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)</code>	<code>GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>	<code>GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>	<code>GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>	<code>GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>	<code>GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>	<code>GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>	<code>GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>	<code>GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>	<code>GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>	<code>GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>	<code>GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>	<code>GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>	<code>GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>
<code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)</code>	<code>GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)</code>

Table 5.11: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_reduce(GrB_Vector,...,GrB_Monoid,...)	GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Vector,...)	GrB_Vector_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Vector,...)	GrB_Vector_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_Vector,...)
GrB_reduce(bool*,...,GrB_Vector,...)	GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)
GrB_reduce(int8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)
GrB_reduce(uint8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)
GrB_reduce(int16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)
GrB_reduce(uint16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)
GrB_reduce(int32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)
GrB_reduce(uint32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)
GrB_reduce(int64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)
GrB_reduce(uint64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)
GrB_reduce(float*,...,GrB_Vector,...)	GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)
GrB_reduce(double*,...,GrB_Vector,...)	GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)
GrB_reduce(<i>other</i> *,...,GrB_Vector,...)	GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)
GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)	GrB_Matrix_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)
GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)	GrB_Matrix_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...)
GrB_reduce(bool*,...,GrB_Matrix,...)	GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)
GrB_reduce(int8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)
GrB_reduce(uint8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)
GrB_reduce(int16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)
GrB_reduce(uint16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)
GrB_reduce(int32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)
GrB_reduce(uint32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)
GrB_reduce(int64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)
GrB_reduce(uint64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)
GrB_reduce(float*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)
GrB_reduce(double*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)
GrB_reduce(<i>other</i> *,...,GrB_Matrix,...)	GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)
GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)

Appendix A

Revision history

Changes in 2.0.1 (Released: ## Xxxxx 2022:

- (Issue GH-69) Fix error in description of contents of matrix constructed from GrB_Matrix_diag.

Changes in 2.0.0 (Released: 15 November 2021:

- Reorganized Chapters 2 and 3: Chapter 2 contains prose regarding the basic concepts captured in the API; Chapter 3 presents all of the enumerations, literals, data types, and predefined objects required by the API. Made short captions for the List of Tables.
- (Issue BB-49, BB-50) Updated and corrected language regarding multithreading and completion, and requirements regarding acquire-release memory orders. Methods that used to force complete no longer do.
- (Issue BB-74, BB-9) Assigned integer values to all return codes as well as all enumerations in the API to ensure run-time compatibility between libraries.
- (Issues BB-70, BB-67) Changed semantics and signature of GrB_wait(obj, mode). Added wait modes for 'complete' or 'materialize' and removed GrB_wait(void). **This breaks backward compatibility.**
- (Issue GH-51) Removed deprecated GrB_SCMP literal from descriptor values. **This breaks backward compatibility.**
- (Issues BB-8, BB-36) Added sparse GrB_Scalar object and its use in additional variants of extract/setElement methods, and reduce, apply, assign and select operations.
- (Issues BB-34, GH-33, GH-45) Added new select operation that uses an index unary operator. Added new variants of apply that take an index unary operator (matrix and vector variants).
- (Issues BB-68, BB-51) Added serialize and deserialize methods for matrices to/from implementation defined formats.

- 7287 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified
7288 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats
7289 have been deferred.
- 7290 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7291 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return
7292 error if duplicate locations encountered.
- 7293 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7294 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7295 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type T (not to
7296 be confused with the proposed unary operator).
- 7297 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added
7298 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to
7299 a method that is not supported by the implementation.
- 7300 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque
7301 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7302 • (Issue BB-45) Removed language about annihilators.
- 7303 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7304 • Updated a number algorithms in the appendix to use new operations and methods.
- 7305 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the
7306 specification.
- 7307 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and
7308 `GRB_SUBVERSION`.
- 7309 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7310 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection
7311 fixed to read `eWiseAdd`/set union.
- 7312 • Typographical error (`NEQ` \rightarrow `NE`) in Description of Table ??.

7313 Changes in 1.3.0 (Released: 25 September 2019):

- 7314 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque
7315 GraphBLAS object as an argument.
- 7316 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7317 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a
7318 scalar.

7319
7320

7321

7322
7323

7324
7325

7326

7327
7328

7329
7330

7331
7332

7333

7334
7335

7336

7337
7338

7339
7340

7341

7342
7343

7344

7345
7346

7347
7348

7349

7350

7351

- (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 ?? 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.

- 7352 • Fixed miscellaneous typographical errors (such as \otimes , \oplus).
- 7353 Changes in 1.2.0:
- 7354 • Removed "provisional" clause.
- 7355 Changes in 1.1.0:
- 7356 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
 - 7357 `assign` operations.
 - 7358 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
 - 7359 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
 - 7360 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
 - 7361 from the domains of the binary operator provided.
 - 7362 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
 - 7363 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
 - 7364 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
 - 7365 which is returned when the capacities of the output arrays are insufficient to hold all of the
 - 7366 tuples.
 - 7367 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
 - 7368 face.
 - 7369 • Added replace flag (`z`) notation to Table ??.
 - 7370 • Updated the “Mathematical Description” of the `assign` operation in Table ??.
 - 7371 • Added triangle counting example.
 - 7372 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
 - 7373 sections of GraphBLAS operations when the respective text was the “standard” text (i.e.,
 - 7374 identical in a majority of the operations).
 - 7375 • Fixed typographical errors.
- 7376 Changes in 1.0.2:
- 7377 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
 - 7378 matrices and avoid casting issues in certain implementations.
 - 7379 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
 - 7380 erased outside the assigned area.
 - 7381 • Changes non-polymorphic interface:
 - 7382 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7383 – Renamed GrB_Vector_reduce_BinaryOp to GrB_Matrix_reduce_BinaryOp.
- 7384 – Renamed GrB_Vector_reduce_Monoid to GrB_Matrix_reduce_Monoid.
- 7385 • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7386 • 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.

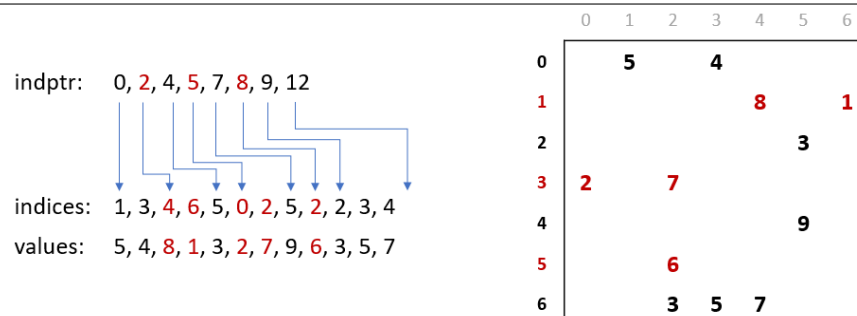


Figure B.1: Data layout for CSR format.

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.

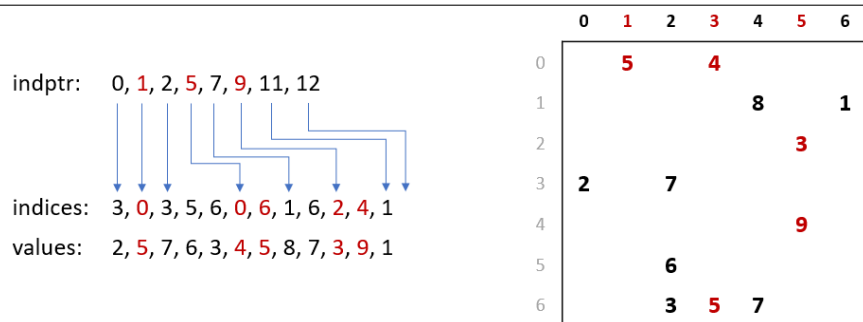


Figure B.2: Data layout for CSC format.

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.

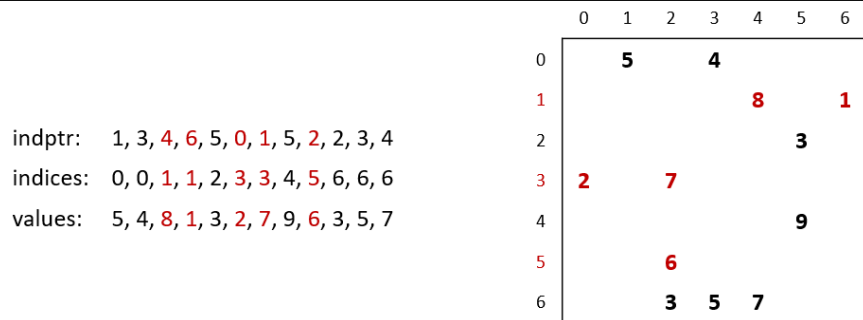


Figure B.3: Data layout for COO format.

7419 **Appendix C**

7420 **Examples**

C.1 Example: Level breadth-first search (BFS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i] = 0$ . (Vector  $v$  should be empty on input.)
11  */
12  GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
13  {
14      GrB_Index n;
15      GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
16
17      GrB_Vector_new(v,GrB_INT32,n);     // Vector<int32_t>  $v(n)$ 
18
19      GrB_Vector q;                      // vertices visited in each level
20      GrB_Vector_new(&q,GrB_BOOL,n);     // Vector<bool>  $q(n)$ 
21      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
22
23      /*
24       * BFS traversal and label the vertices.
25       */
26      int32_t d = 0;                     //  $d = \text{level in BFS traversal}$ 
27      bool succ = false;                  //  $\text{succ} == \text{true}$  when some successor found
28      do {
29          ++d;                            // next level (start with 1)
30          GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL); //  $v[q] = d$ 
31          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
32                  q,A,GrB_DESC_RC);      //  $q[!v] = q \parallel A$ ; finds all the
33                                          // unvisited successors from current  $q$ 
34          GrB_reduce(&succ,GrB_NULL,GrB_LOR_MONOID_BOOL,
35                    q,GrB_NULL);          //  $\text{succ} = \parallel(q)$ 
36      } while (succ);                     // if there is no successor in  $q$ , we are done.
37
38      GrB_free(&q);                       //  $q$  vector no longer needed
39
40      return GrB_SUCCESS;
41  }

```

C.2 Example: Level BFS in GraphBLAS using apply

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i]$  does not have a stored element.
11  * Vector  $v$  should be uninitialized on input.
12  */
13  GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index n;
16      GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
17
18      GrB_Vector_new(v,GrB_INT32,n);     // Vector<int32_t>  $v(n) = 0$ 
19
20      GrB_Vector q;                     // vertices visited in each level
21      GrB_Vector_new(&q,GrB_BOOL,n);     // Vector<bool>  $q(n) = \text{false}$ 
22      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
23
24      /*
25       * BFS traversal and label the vertices.
26       */
27      int32_t level = 0;                 // level = depth in BFS traversal
28      GrB_Index nvals;
29      do {
30          ++level;                       // next level (start with 1)
31          GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,
32                  GrB_SECOND_INT32,q,level,GrB_NULL); //  $v[q] = \text{level}$ 
33          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
34                  q,A,GrB_DESC_RC);     //  $q[!v] = q \ || \ \&\& \ A$ ; finds all the
35                                      // unvisited successors from current  $q$ 
36          GrB_Vector_nvals(&nvals, q);
37      } while (nvals);                   // if there is no successor in  $q$ , we are done.
38
39      GrB_free(&q);                      //  $q$  vector no longer needed
40
41      return GrB_SUCCESS;
42  }

```

C.3 Example: Parent BFS in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a binary  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS
9   * traversal of the graph and sets  $parents[i]$  to the index of vertex  $i$ 's parent.
10  * The parent of the root vertex,  $s$ , will be set to itself ( $parents[s] = s$ ). If
11  * vertex  $i$  is not reachable from  $s$ ,  $parents[i]$  will not contain a stored value.
12  */
13  GrB_Info BFS(GrB_Vector *parents, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index N;
16      GrB_Matrix_nrows(&N, A);                      //  $N = \#$  vertices
17
18      GrB_Vector_new(parents, GrB_UINT64, N);
19      GrB_Vector_setElement(*parents, s, s);          //  $parents[s] = s$ 
20
21      GrB_Vector wavefront;
22      GrB_Vector_new(&wavefront, GrB_UINT64, N);
23      GrB_Vector_setElement(wavefront, 1UL, s);      //  $wavefront[s] = 1$ 
24
25      /*
26       * BFS traversal and label the vertices.
27       */
28      GrB_Index nvals;
29      GrB_Vector_nvals(&nvals, wavefront);
30
31      while (nvals > 0)
32      {
33          // convert all stored values in wavefront to their 0-based index
34          GrB_apply(wavefront, GrB_NULL, GrB_NULL, GrB_ROWINDEX_INT64,
35                  wavefront, 0UL, GrB_NULL);
36
37          // "FIRST" because left-multiplying wavefront rows. Masking out the parent
38          // list ensures wavefront values do not overwrite parents already stored.
39          GrB_vxm(wavefront, *parents, GrB_NULL, GrB_MIN_FIRST_SEMIRING_UINT64,
40                  wavefront, A, GrB_DESC_RSC);
41
42          // Don't need to mask here since we did it in vxm. Merges new parents in
43          // current wavefront with existing parents:  $parents += wavefront$ 
44          GrB_apply(*parents, GrB_NULL, GrB_PLUS_UINT64,
45                  GrB_IDENTITY_UINT64, wavefront, GrB_NULL);
46
47          GrB_Vector_nvals(&nvals, wavefront);
48      }
49
50      GrB_free(&wavefront);
51
52      return GrB_SUCCESS;
53  }

```

C.4 Example: Betweenness centrality (BC) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ ,
9   * compute the BC-metric vector  $\delta$ , which should be empty on input.
10  */
11  GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
12  {
13      GrB_Index n;
14      GrB_Matrix_nrows(&n, A);                //  $n = \#$  of vertices in graph
15
16      GrB_Vector_new(delta, GrB_FP32, n);      // Vector<float>  $\delta(n)$ 
17
18      GrB_Matrix sigma;
19      GrB_Matrix_new(&sigma, GrB_INT32, n, n); //  $\text{Matrix}<\text{int32}_t> \text{sigma}(n, n)$ 
20                                              //  $\text{sigma}[d, k] = \#$  shortest paths to node  $k$  at level  $d$ 
21
22      GrB_Vector q;
23      GrB_Vector_new(&q, GrB_INT32, n);        // Vector<int32_t>  $q(n)$  of path counts
24      GrB_Vector_setElement(q, 1, s);          //  $q[s] = 1$ 
25
26      GrB_Vector p;
27      GrB_Vector_dup(&p, q);                   // Vector<int32_t>  $p(n)$  shortest path counts so far
28                                              //  $p = q$ 
29
30      GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
31              q, A, GrB_DESC_RC);              // get the first set of out neighbors
32
33      /*
34       * BFS phase
35       */
36      GrB_Index d = 0;                          // BFS level number
37      int32_t sum = 0;                          // sum == 0 when BFS phase is complete
38
39      do {
40          GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\text{sigma}[d, :] = q$ 
41          GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
42          GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
43                  q, A, GrB_DESC_RC);          //  $q = \#$  paths to nodes reachable
44                                              // from current level
45          GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
46          ++d;
47      } while (sum);
48
49      /*
50       * BC computation phase
51       * ( $t1, t2, t3, t4$ ) are temporary vectors
52       */
53      GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
54      GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
55      GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
56      GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
57
58      for (int i=d-1; i>0; i--)
59      {
60          GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t1 = 1 + \delta$ 
61          GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, t1, *delta, GrB_NULL);
62          GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t2 = \text{sigma}[i, :]$ 
63          GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t2 = (1 + \delta) / \text{sigma}[i, :]$ 
64          GrB_mvx(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
65                  // add contributions made by

```

```

63         A, t2, GrB_NULL);
64     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, GrB_DESC_T0); // t4 = sigma[i-1,:]
65     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, GrB_TIMES_FP32, t4, t3, GrB_NULL); // t4 = sigma[i-1,:]*t3
66     GrB_eWiseAdd(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, *delta, t4, GrB_NULL); // accumulate into delta
67 }
68
69 GrB_free(&sigma);
70 GrB_free(&q); GrB_free(&p);
71 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
72
73 return GrB_SUCCESS;
74 }

```

C.5 Example: Batched BC in GraphBLAS

```

1  #include <stdlib.h>
2  #include "GraphBLAS.h" // in addition to other required C headers
3
4  // Compute partial BC metric for a subset of source vertices, s, in graph A
5  GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
6  {
7      GrB_Index n;
8      GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
9      GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
10
11     // index and value arrays needed to build numsp
12     GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
13     int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
14     for(int i=0; i<nsver; ++i) {
15         i_nsver[i] = i;
16         ones[i] = 1;
17     }
18
19     // numsp: structure holds the number of shortest paths for each node and starting vertex
20     // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
21     GrB_Matrix numsp;
22     GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
23     GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
24     free(i_nsver); free(ones);
25
26     // frontier: Holds the current frontier where values are path counts.
27     // Initialized to out vertices of each source node in s.
28     GrB_Matrix frontier;
29     GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
30     GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, GrB_DESC_RCT0);
31
32     // sigma: stores frontier information for each level of BFS phase. The memory
33     // for an entry in sigmas is only allocated within the do-while loop if needed.
34     // n is an upper bound on diameter.
35     GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n);
36
37     int32_t d = 0; // BFS level number
38     GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
39
40     // ----- The BFS phase (forward sweep) -----
41     do {
42         // sigmas[d](:,s) = d^th level frontier from source vertex s
43         GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
44
45         GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
46                 GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
47         GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL, GrB_PLUS_INT32,
48                     numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
49         GrB_mxm(frontier, numsp, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
50                 A, frontier, GrB_DESC_RCT0); // f<!numsp> = A' +.* f (update frontier)
51         GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level
52         d++;
53     } while (nvals);
54
55     // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
56     GrB_Matrix nspinv;
57     GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
58     GrB_apply(nspinv, GrB_NULL, GrB_NULL,
59              GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
60
61     // bcu: BC updates for each vertex for each starting vertex in s
62     GrB_Matrix bcu;

```

```

63 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
64 GrB_assign(bcu, GrB_NULL, GrB_NULL,
65           1.0f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
66
67 GrB_Matrix w; // temporary workspace matrix
68 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
69
70 // ----- Tally phase (backward sweep) -----
71 for (int i=d-1; i>0; i--) {
72     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
73                 GrB_TIMES_FP32, bcu, nspinv, GrB_DESC_R); // w<sigmas[i]>=(1 ./ nsp).*bcu
74
75     // add contributions by successors and mask with that BFS level's frontier
76     GrB_mxm(w, sigmas[i-1], GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
77            A, w, GrB_DESC_R); // w<sigmas[i-1]> = (A +.* w)
78     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32, GrB_TIMES_FP32,
79                 w, numsp, GrB_NULL); // bcu += w .* numsp
80 }
81
82 // row reduce bcu and subtract "nsver" from every entry to account
83 // for 1 extra value per bcu row element.
84 GrB_reduce(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, bcu, GrB_NULL);
85 GrB_apply(*delta, GrB_NULL, GrB_NULL, GrB_MINUS_FP32, *delta, (float)nsver, GrB_NULL);
86
87 // Release resources
88 for (int i=0; i<d; i++) {
89     GrB_free(&(sigmas[i]));
90 }
91 free(sigmas);
92
93 GrB_free(&frontier); GrB_free(&numsp);
94 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
95
96 return GrB_SUCCESS;
97 }

```


C.6 Example: Maximal independent set (MIS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  // Assign a random number to each element scaled by the inverse of the node's degree.
8  // This will increase the probability that low degree nodes are selected and larger
9  // sets are selected.
10 void setRandom(void *out, const void *in)
11 {
12     uint32_t degree = *(uint32_t*)in;
13     *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
14 }
15
16 /*
17  * A variant of Luby's randomized algorithm [Luby 1985].
18  *
19  * Given a numeric n x n adjacency matrix A of an unweighted and undirected graph (where
20  * the value true represents an edge), compute a maximal set of independent vertices and
21  * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
22  * of the set (the iset vector should be uninitialized on input.)
23  */
24 GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
25 {
26     GrB_Index n;
27     GrB_Matrix_nrows(&n,A); // n = # of rows of A
28
29     GrB_Vector prob; // holds random probabilities for each node
30     GrB_Vector neighbor_max; // holds value of max neighbor probability
31     GrB_Vector new_members; // holds set of new members to iset
32     GrB_Vector new_neighbors; // holds set of new neighbors to new iset mbrs.
33     GrB_Vector candidates; // candidate members to iset
34
35     GrB_Vector_new(&prob,GrB_FP32,n);
36     GrB_Vector_new(&neighbor_max,GrB_FP32,n);
37     GrB_Vector_new(&new_members,GrB_BOOL,n);
38     GrB_Vector_new(&new_neighbors,GrB_BOOL,n);
39     GrB_Vector_new(&candidates,GrB_BOOL,n);
40     GrB_Vector_new(iset,GrB_BOOL,n); // Initialize independent set vector, bool
41
42     GrB_UnaryOp set_random;
43     GrB_UnaryOp_new(&set_random,setRandom,GrB_FP32,GrB_UINT32);
44
45     // compute the degree of each vertex.
46     GrB_Vector degrees;
47     GrB_Vector_new(&degrees,GrB_FP64,n);
48     GrB_reduce(degrees,GrB_NULL,GrB_NULL,GrB_PLUS_FP64,A,GrB_NULL);
49
50     // Isolated vertices are not candidates: candidates[degrees != 0] = true
51     GrB_assign(candidates,degrees,GrB_NULL,true,GrB_ALL,n,GrB_NULL);
52
53     // add all singletons to iset: iset[degree == 0] = 1
54     GrB_assign(*iset,degrees,GrB_NULL,true,GrB_ALL,n,GrB_DESC_RC) ;
55
56     // Iterate while there are candidates to check.
57     GrB_Index nvals;
58     GrB_Vector_nvals(&nvals, candidates);
59     while (nvals > 0) {
60         // compute a random probability scaled by inverse of degree
61         GrB_apply(prob, candidates,GrB_NULL,set_random,degrees,GrB_DESC_R);
62     }

```

```

63 // compute the max probability of all neighbors
64 GrB_mnv(neighbor_max, candidates, GrB_NULL, GrB_MAX_SECOND_SEMIRING_FP32, A, prob, GrB_DESC_R);
65
66 // select vertex if its probability is larger than all its active neighbors,
67 // and apply a "masked no-op" to remove stored falses
68 GrB_eWiseAdd(new_members, GrB_NULL, GrB_NULL, GrB_GT_FP64, prob, neighbor_max, GrB_NULL);
69 GrB_apply(new_members, new_members, GrB_NULL, GrB_IDENTITY_BOOL, new_members, GrB_DESC_R);
70
71 // add new members to independent set.
72 GrB_eWiseAdd(*iset, GrB_NULL, GrB_NULL, GrB_LOR, *iset, new_members, GrB_NULL);
73
74 // remove new members from set of candidates  $c = c \& \neg \text{new}$ 
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  }

```