GDI: A Graph Database Interface Standard

Maciej Besta*†, Robert Gerstenberger*†, Nils Blach, Marc Fischer‡, Torsten Hoefler†

ETH Zurich

*Alphabetical order †Contact authors
‡The author contributed to the specification while at ETH Zurich

Version 0.9 February 7, 2023

Contents

1 Overview

This section outlines the Graph Database Interface (GDI).

1.1 GDI and Its Goals

GDI is a key ingredient in an effort to solve four key challenges of graph databases: high performance, scalability, programmability, and portability.

GDI is a storage engine layer interface for distributed graph databases. As such, its main purpose is to abstract the low-level storage layer such that higher-level parts (query methods, query planners, execution engines, and others) of the graph database can run vendor agnostic. This also allows to distribute the graph over a network to multiple storage backends (multiple machines) which might rely on main-memory, hard disks, SSDs, and others. The interface provides a short list of methods offering CRUD (create, read, update, delete) functionality for common graph data concepts, including edges, vertices, properties, and labels. The focus lies on methods with clear semantics such that high-performance implementations are possible that scale to thousands of cores. Simultaneously, the provided semantics are rich and support different graph database features such as ACID transaction handling.

This document outlines GDI, consisting of conventions, function definitions and their semantic. The goal of GDI is to help to develop a widely used standard for writing highly scalable distributed graph databases that is established in both academia and industry.

1.2 Labeled Property Graph Model

A graph is a tuple $G = (V, E)$, where an object $v \in V$ is called a *vertex*. An *edge* represents a relation and denotes an unordered pair of vertices that connects said vertices: $e = \{u, v\}$. The set of edges is denoted by $E \subseteq V \times V$. If the relation has an associated direction, an edge is said to be *directed* and denotes the ordered pair $e = (u, v)$ of vertices that connects said vertices. The term *origin* specifies the starting point of a directed edge and *target* the end point. From the origin's perspective, an edge is *outgoing* and from the target's perspective it is *incoming*.

GDI extends graphs and implements the labeled property graph model (LPG) [2]. LPG adds labels and properties to the simple graph model $G = (V, E)$. *Labels* differentiate subsets of vertices and edges. In addition to labels, each vertex and edge can feature a non-negative number of *properties* (sometimes referenced as attributes in scientific literature). A property is a $(key, value)$ pair, where the *key* works as an identifier with *value* being the corresponding value. A label property graph can formally be modeled as a tuple:

$$LPG = (V, E, L, l_V, l_E, K, W, p_V, p_E)$$

V and E are defined in the same way as in the simple graph model above. L denotes the set of labels. l_V and l_E describe labeling functions, which map vertices and edges respectively to a subset of labels: $l_V : V \mapsto \mathcal{P}(L)$ and $l_E : E \mapsto \mathcal{P}(L)$ with $\mathcal{P}(L)$ being the power set of L, meaning all possible subsets of L. In addition to labels, each vertex and edge can have an arbitrary non-negative number of properties, which are key-value pairs $p = (key, value)$. K is the set of all possible keys where as W denotes the set of all possible values. So every property satisfies $key \in K$ and $value \in W$. $p_V(u)$ describes the set of properties of a vertex u for every $u \in V$, and $p_E(e)$ expresses the set of properties for every edge $e \in E$. Note that only the pair $(key, value)$ must be unique. Therefore it is allowed to assign multiple properties with the same key to vertices and edges.

1.2.1 Additions and Restrictions to the LPG Model

To increase the flexibility of the data model, GDI allows to have multiple identical edges between the same vertices. Also, it is possible to have edges that connect a vertex to itself (loop). Further, it is possible to have both directed and undirected edges in the same graph.

Additionally, vertices can have identifiers that are specified by the client; this follows the requirement set by the LDBC council [1] in the SNB benchmark specification [4]. GDI limits the use of identifiers: All vertex identifiers within a label are unique, but vertices of different labels might share the same identifier. An exception to this rule is when vertices do not have a label, then it is possible to have multiple vertices with the same identifier.

1.3 Context of GDI

GDI acts as storage engine interface for a distributed graph database. As such, it is the task of GDI to ensure data consistency, transactional data access, low response times and high throughput. If the data is distributed among multiple machines, implementations must consider the CAP-theorem and explicitly state which properties (consistency, availability, partition tolerance) they offer. Note that due to the ACID guarantee of GDI, implementations must offer consistency. This also implies that data might be stored redundant such that fault tolerance is taken into account.

Table 1 illustrates an example design of a graph database . GDI acts as an API that can be used by any of the higher layers.

(Layer 6) Client
A client queries the graph database. Typically, the client uses a graph query language to run traversals and graph matching requests.
(Layer 5) Query Planner
The query planner works in close cooperation with the execution engine to determine an ordered set of steps to execute the query given by the client.
(Layer 4) Execution Engine
Execution engine distributes workload among multiple machines and aggregate intermediate results that ran on different processes.
(Layer 3) Query Functions
Query functions consume data from the storage engine and return objects like edges, vertices, paths, or subgraphs. Aggregation functions are provided to return aggregated values. Further functionality includes filtering of objects (e.g., by labels or properties).
(Layer 2) Storage Engine (GDI)
Storage engine uses the low-level storage layer to access the graph data. It basically translates from disk dependent storage (for example CSV, JSON, binary format, block format) to generic objects. Therefore, this layer provides a rich set of interfaces to create, read, update and delete (CRUD) vertices, edges and associated labels and properties. This layer should provide ACID guarantees to the upper layers. If not, then a layer above must handle queries in a way that they do not interfere.
(Layer 1) Low-Level Storage Layer (Storage Backend)
This layer provides an abstraction for a low-level storage layer such as hard disks (for example CSV files, JSON, binary formats, block format), RAM, distributed RAM or others. Its goal is to store the data in a reliable way and provide fast data access.

Table 1: An example layering of a graph database. GDI is an interface for the storage engine. Extensions such as query planner and execution engine can rely on the properties that GDI offers.

1.4 Execution and Consistency

GDI is constructed with distributed graph databases in mind, but it can also be used for single-node or single-core databases. Generally, it is assumed that a set of independent processes run concurrently in a (tightly coupled) compute cluster. GDI offers no general functionality to the user to manage the processes as it might be required in a primary-secondary model or a full-fledged graph database. Instead, it is the responsibility of the user to distribute and assign work to the processes in the appropriate way.

GDI uses different consistency models. GDI guarantees serializability for graph data, such as vertices, edges and associated labels and properties. Generally, this data can only be altered by transactions that ensure ACID properties (atomicity, consistency, isolation, durability). Further, GDI guarantees eventual consistency for global elements, such as labels, property types and indexes. Since these objects also affect the graph data, this might lead to cases where graph data becomes inconsistent until the system has converged. Transactions must be able to detect such state and abort accordingly. GDI provides barrier functions to the user to synchronize the system. Some GDI functions provide explicit synchronization, which is generally described in more detail in the function's specification. Note that implementations might provide consistency models for global elements that are more restrictive (stronger) than eventual consistency.

1.5 Following Established and Time-Tested Specifications

In GDI, we follow the practice and style of the Message Passing Interface (MPI) [3], an established API that provides a specification of a communication library for computing clusters. Our main motivation is the fact that the goals of MPI are very similar to those of GDI, i.e., MPI was designed to enable portability, programmability, scalability, and high performance.

1.6 Low-Level of Specification

In GDI, we follow MPI and use an analogous "low-level" style of the interface specification, that is similar to C. This fosters portability across different architectures. However, there is nothing that prevents potential GDI implementations from adapting an object-oriented style, as long as they adhere to the GDI semantics. In fact, MPI implementations such as MPICH also provide "object style" APIs to their communication functions.

1.7 Document Structure

First, terms and conditions are provided (Section 2). Then, we list functions for the general GDI management such as initialization or completion (Section 3) and for database creation and destruction (Section 4). Later, we list functions related to the graph metadata, namely labels (Section 5) and properties (Section 6), and to the graph data, namely vertices (Section 7) and edges (Section 8). After that, functions for indexes (Section 9), basic datatypes (Section 10), and transactions (Section 11) are provided. Finally, we list functions related to constraints (Section 12), error handling (Section 13), and bulk data loading (Section 15).

2 Terms and Conventions

This section provides an overview of terms and conventions to describe the graph database interface.

2.1 Abbreviations and Terms

- Client: A person that runs graph queries using the GDI interface or functionality provided by the user (e.g., a person implementing applications incorporating graph databases).
- GDI: The graph database interface.
- ID: Identifier.
- Implementor: A person that implements the functions specified by GDI.
- Index: In this context we define an index as a database index. A database index is a structure which allows to quickly locate an object without having to search through the whole database.
- Synchronization[1]: The term is used for process synchronization. It ensures that a set of processes have reached a common point in their instruction flow.
- UID: Unique identifier.
- User: A person that interacts directly with the GDI interface (e.g., a graph database developer).

2.2 Separation of Responsibilities

In GDI, we clearly separate the responsibilities of Client, User, and Implementor. While this is usually obvious, in some cases, we explicitly remark on whether a given aspect of a respective GDI function, is something that lies within the responsibility of a Client, a User, or an Implementor. This facilitates working with GDI and clarifies its semantics.

2.3 Document Notation

Across the specification, paragraphs set in the following formats contain complementary material intended for specific audiences. Some readers may read these sections rigorously, while others may choose to ignore them.

> *Rationale.* These sections provide the reasoning behind the design decisions made and are specifically targeted at readers interested in interface design. (*End of rationale.*)

> *Advice to users.* These sections provide additional information for users, illustrating certain aspects of using GDI, which are specifically targeted at readers interested in developing GDI programs. (*End of advice to users.*)

> *Advice to implementors.* These sections provide further comments for implementors and are specifically targeted at readers interested in GDI implementations. (*End of advice to implementors.*)

[1] In the literature, the term "coordination" is sometimes also used.

2.4 Function Specification

The parameters within function declarations are categorized as IN, OUT, or INOUT, in order to indicate their respective use within the function to the user. Note, however, that this parameter categorization is not fine-grained enough for a direct translation into language bindings, such as `const` in C. The categories have the following meaning:

- IN: The parameter's input value may be used by the function, but is not updated during the execution of the function,
- OUT: The parameter may be updated during the execution of the function, but its input values is not used,
- INOUT: The parameter's input value may be used by the function and the parameter may be updated during the execution of the function.

In several cases GDI uses reference parameters in its function interface. Such references are either addresses to user-provided buffer space or handles to opaque objects (see 2.6.1 for term definitions). These parameters are sometimes treated slightly different: If the target of the reference is modified, then the reference parameter is also categorized as OUT or INOUT, albeit the reference itself is not modified.

> *Rationale.* The GDI specification attempts to avoid the use of the INOUT category as much as possible, because more restrictive categorization of parameters results in better interpretability and fewer erroneous uses. (*End of rationale.*)

All GDI functions are specified in ISO C 99.

2.5 Semantic Terms

The GDI specification uses the following semantic terms, when discussing GDI functions.

local Any function whose completion, be it successful or unsuccessful, is solely depended on the locally executing process, is considered local.

collective Any function whose completion, be it successful or unsuccessful, requires all processes of the database to call said function, is considered collective. Collective functions do not guarantee any synchronization, but may be synchonizing. The execution order of collective functions must be equivalent on all processors.

2.6 Data Types

2.6.1 Opaque Objects

Opaque objects are objects stored in system memory. They are not directly accessible, and their size and shape is hidden. The user can only access opaque objects via handles that reside in user-provided space. These handles can be used to access objects, be passed as arguments, as well as participate in assignments and comparisons, which is essential for validity checks. In GDI, objects involved in transactions or internal representations of objects such as labels, property types, transactions, etc. are stored opaquely in system memory and passed to GDI functions via handle arguments.

Some predefined opaque objects with associated handles are provided by GDI, which must not be deallocated by the user. Other opaque objects must be allocated and deallocated via specific functions that match the type of the object. Each allocator function specifies the handle as an OUT parameter, which will be assigned a valid reference to the allocated opaque object. On the other hand, each deallocator function specifies the handle as an INOUT parameter, as the referenced opaque object will be accessed and a typed "invalid handle" constant will be returned.

From the user's perspective, the opaque object becomes inaccessible as soon as the deallocation function returns, but actual deallocation only occurs once all pending operations, i.e., transactions, that involve the object at the time of the deallocation function call, have completed.

Note that opaque objects and their handles are process specific, so they are only significant at their respective allocating process and cannot be shared with other processes via communication.

> *Rationale.* The distinction between opaque objects (in system space) and their handles (in user space) has two main reasons. Firstly, it conceals the internal representation of GDI data structures. Secondly, it improves usability, as it relinquishes the responsibility of ensuring that there are no pending operations involving out-of-scope, opaque objects to the GDI implementation. This design allows users to deallocate at appropriate times by simply marking objects for deallocation, relying on the GDI implementation to retain the object until all pending operations completed.
>
> Requiring handles to support the frequently used operations, assignment and comparison, restricts the range of possible implementations, for the benefit of reduced complexity. Otherwise, arbitrarily typed handles would require additional functions for these operations, thereby increasing complexity. So, these limitations were placed upon potential GDI implementations to use native-language assignment and comparison operations and contribute to the goal of keeping the GDI interface clean and simple. (*End of rationale.*)

> *Advice to users.* The user is mostly responsible for managing handles, the references, to opaque objects. While GDI offers functionalities to retrieve handles of all opaque objects of a given type from a database, the avoidance of situations such as dangling references is the users purview. (*End of advice to users.*)

> *Advice to implementors.* GDI intends the allocation and deallocation of objects to appear to the user as if the information for those objects were copied, so that semantically opaque objects are separate from each other. However, this does not mean that GDI implementations may not employ optimisations such as references and reference counting, if they are hidden from the user. (*End of advice to implementors.*)

2.6.2 Array Arguments

Some GDI functions accept an array as input parameter. But, whenever an array argument is used, it must be accompanied by an additional length argument `count`, which defines the number of valid entries stored consecutively, at the beginning of the array. It follows that `count` must be smaller than or equal to the size of the entire array. If a regular array stores handles to opaque objects of the same type, it is referred to as an array of handles. Note that in some, always appropriately indicated cases an array of handles containing NULL handles is considered valid.

Other GDI functions return an array. The process of returning an array requires three parameters. Firstly, the user needs to allocate a buffer and pass a pointer to its location via an OUT parameter. This buffer will be used to store the elements of the array. The user also supplies an IN parameter (`count`) to indicate the number of elements the buffer can hold. Lastly, the function declares the actual number of elements written to the buffer, by setting an OUT parameter (`resultcount`). The user can request the function to only return the number of elements (using the `resultcount` parameter) and not the array, by setting either the `count` parameter to 0 or the buffer pointer to null. Passing null as the `resultcount` parameter results in the function to ignore the buffer parameter and return nothing.

2.6.3 State

State information is used as arguments for specific GDI functions. Their values are identified by names, and they do not support any operation on them. Many GDI functions utilize some state information, i.e., the GDI_CreateIndex function has a state argument `itype` with values GDI_INDEXTYPE_HASHTABLE and GDI_INDEXTYPE_BTREE.

2.6.4 Named Constants

GDI provides a set of predefined named constand handles. Named constants do not change their values during execution and can be defined at link-time or compile-time. In either case they can be employed for initializations or assignments, but only compile-time named constants can also be used for array length declarations and as labels in C switch statements.

Similar to the joint treatment of handles and their respective opaque objects in 2.4, opaque objects referenced by constant handles are also treated as constants after GDI initialization and before GDI completion.

The following named constants are required to be defined at compile-time:

GDI_MAX_DECIMAL_SIZE
GDI_MAX_ERROR_STRING
GDI_MAX_OBJECT_NAME

2.6.5 Choice

Some GDI functions accept arguments of choice (or union) data type. This allows users to pass by reference actual arguments of different types for distinct invocations of the same function.

2.6.6 Convention For Strings As Function Parameters

Some GDI functions take strings as input parameters. Those input strings are expected to be null terminated and encoded in UTF-8.

Other GDI functions return a UTF-8 encoded string. The process of returning a string requires three parameters. Firstly, the user needs to allocate a buffer and pass a pointer to its location via an OUT parameter. This buffer will be used to store the string. The user also supplies an IN parameter (`length`) to indicate the number of Bytes (n) that the buffer can hold. Lastly, the function declares the actual length (in Bytes) of the string written to the buffer, by setting an OUT parameter (`resultlength`). Note that this length does not include the null terminator and cannot exceed $n - 1$ Bytes, whereby n denotes the size of the buffer. If the requested string's length exceeds this limit, it will be truncated to a fitting number of characters accordingly, so that at most $n - 1$ Bytes will be written to the buffer (excluding the null terminator). GDI ensures that the returned string is correctly UTF-8 encoded. The user can request the function to only return the length of the string (using the `resultlength` parameter) and not the string itself, by setting either the `length` parameter to 0 or the buffer pointer to null. Passing null as the `resultlength` parameter results in the function to ignore the buffer parameter and return nothing.

2.7 Naming Objects

Naming GDI objects by associating them with printable, human-readable identifiers can prove useful in several scenarios. For instance, when querying the database for labels or property types, or when examining vertices and edges during graph exploration for debugging purposes.

Functions with a name parameter, which is required to be a UTF-8 encoded character string, will associate the name with an object. The GDI library will copy the passed string into its local store, such that the user is free to deallocate it at any time after the call. Note that trailing spaces in the name parameter will be ignored, while leading spaces are relevant.

Setting the name is a collective operation with the requirement of using the same input parameter on every process, so that the name for such an object is the same on all processes.

GDI_MAX_OBJECT_NAME restricts the maximum length of names. Any name with a length of more than GDI_MAX_OBJECT_NAME-1 Bytes (last Byte reserved for the null terminator) will be truncated. Note that the value of GDI_MAX_OBJECT_NAME must be at least 64.

> *Advice to users.* There is no guarantee that names, whose length is smaller than GDI_MAX_OBJECT_NAME, can always be successfully assigned, as their storage requirements might exceed the remaining space of the GDI library. Thus, the constant GDI_MAX_OBJECT_NAME should only be treated as a strict upper bound. (*End of advice to users.*)

Advice to implementors. Name retrieving functions require the user to allocate sufficient space for names of up to GDI_MAX_OBJECT_NAME length. Thus, implementations that utilize the heap to allocate space for names should still define GDI_MAX_OBJECT_NAME to be relatively small. Implementations which preallocate a fixed amount of space for a name should define GDI_MAX_OBJECT_NAME to be the size of that preallocation. (*End of advice to implementors.*)

For a given object, only the last, previously associated name will be returned by a call to a name retrieving function. These functions require the user to pass a name argument, which will be used to store a copy of the set name. The user should allocate enough space to store a string of GDI_MAX_OBJECT_NAME Bytes.

The last Byte of the returned string in the `name` parameter, located at `name[resultlength]`, will be used to store a null terminator. The `resultlength` parameter contains the length of the retrieved string, which can be at most GDI_MAX_OBJECT_NAME-1.

An erroneous name retrieving function call will return an empty string ("" in C).

Advice to users. It is always safe to print the string returned by a name retrieving function call, even during an erroneous execution, as implied by the above definition. (*End of advice to users.*)

2.8 Error Handling

GDI provides reliable data transmission. It is the implementors responsibility to ensure that data is always obtained correctly and no communication failure handling functionality is needed. The implementors need to further ensure that if an unreliable mechanism is used in the GDI subsystem, the user is oblivious to this, or unrecoverable errors are reported as failures. To improve interpretability, such failures will be reported as errors in the relevant call, whenever possible. Similarly, no mechanisms are provided by GDI for handling processor failures.

When errors occur in a GDI program, these errors generally belong to one of two groups. Firstly, the GDI implementation independent **program error**, which can occur when invalid arguments (invalid handle, incorrect buffer size, etc.) are passed to GDI functions. Secondly, depending on the resource requirements of the GDI program, it might exceed the available system resources (system buffers, number of pending operations etc.) and cause a **resource error**. This is system dependent, but high-quality implementations will make the necessary trade-offs in favour of the more important resources, in order to alleviate the portability problem this represents.

All GDI function executions either return a code indicating a successful completion or, whenever possible, one of many error codes indication an erroneous execution.

GDI tries to return meaningful error codes for erroneous executions, but its ability to do so is limited by a multitude of factors. Some errors might be too difficult/expensive to detect in normal execution mode. Others may be more severe and result in inconsistent state, preventing GDI from returning to the caller safely. Errors of this nature are for the most part kernel, hardware, or driver errors.

Further limitations are introduced due to the use of asynchronous operations. Asynchronous operations (e.g., write accesses) may return with a code indicating successful completion before actual completion and result in a belated error (e.g., during a commit call). In order to mitigate this limitation, if possible, GDI will utilize the error argument of a subsequent call, related to the same operation, to indicate the actual origin of the error.

After an erroneous GDI call has occurred, it is usually possible to recover, especially from program errors, by aborting the current operation/transaction, retrying, or following a different execution path. Because of the consistency requirement, data in the database should not have been altered. Ideally, an erroneous GDI call returns the most relevant error code and localizes the impact of the error as effectively as possible. For instance, an erroneous GDI_Get... call should not overwrite any memory, other than the buffer space specified in the function call for receiving data. Section 13 elaborates on the state of the database after an erroneous function call.

GDI Implementations may extend the support of erroneous GDI calls, as defined in this specification, in a meaningful manner. For example, GDI specifies strict type conversion rules:

it is erroneous to convert an integer type to a datetime type. GDI implementations may go beyond these conversion rules, and provide type conversion by interpreting the integer as Unix time. It may be helpful to generate warnings for non-conforming behavior.

2.9 File Path and Access

Some GDI functions require file access, e.g. to load bulk data from CSV like files. The path to such functions is given as null terminated character string and should include the path to the file and the file name as well. Each GDI implementation must document its file path formatting requirements, as they are implementation dependent.

> *Advice to implementors.* Implementations should nevertheless conform to certain standards. For example, on Unix and Unix-like operating systems, a path that starts with `'/'` is treated as an absolute path. Similarly, on operating systems that use the kernel Microsoft Windows NT, a path that starts with the drive letter and continues with `':/'` (e.g. `'C:/'`) is treated as an absolute path. (*End of advice to implementors.*)

It is the responsibility of the user to make sure that GDI has the according privileges to read files.

3 Initialization and Completion

GDI wants to maximize *source code portability.* Portability in this context denotes that no changes to the source code are required, when transitioning GDI source code from one system to another. However there are two exceptions: the initialization function GDI_Init and the function call to create a database GDI_CreateDatabase (chapter 4).

```
int GDI_Init( int *argc, char ***argv )
```

GDI_Init allocates all necessary internal data structures that are needed to support the functionality provided by GDI. It is a collective call, and each process must call GDI_Init before any other GDI functions are called. Any additional calls of GDI_Init afterwards are erroneous. GDI_Init expects as its arguments either null or `argc` and `argv`, the arguments of the main function.

```
int GDI_Finalize()
```

GDI_Finalize deallocates all internal data structures and memory associated with the GDI library, including any remaining graph database objects and their associated objects. It is a collective call, and must be called by each process before the program terminates. Afterwards no GDI function is allowed to be called, which includes GDI_Init.

4 Databases

```
int GDI_CreateDatabase( void* params, size_t size, GDI_Database* graph_db )
```

IN	params	inital address of implementation specific parameters (choice)
IN	size	size of the implementation specific parameters in Bytes (non-negative integer)
OUT	graph_db	graph database object returned by the call (handle)

GDI_CreateDatabase will create a database object and allocate all necessary internal data structures. It is possible to create more than one graph database, which are then identified by different GDI_Database handles.

> *Rationale.* Storing graphs in different graph databases instead of storing them as disconnected components, allows to run OLAP style algorithms on each graph separately and to compute associated metrics with more ease. Another example would be the ease of implementation of graph compression, where one graph database object stores the uncompressed graph and another one the compressed graph. (*End of rationale.*)

The parameters (and their structure) pointed to by `params` are specific to the respective GDI implementation, so the user is advised to consult their documentation for more details. The parameter `size` provides the size in Bytes of the structure to which `params` points to. All input parameters should be the same on all processes.

> *Rationale.* The GDI_CreateDatabase function is not portable, due to the implementation specific parameter `params`. Making the function portable requires the implementation to provide reasonable default values. However, the default values might lead to poor performance and undesired behavior such that the user has to consult the implementation's documentation anyway. (*End of rationale.*)

> *Advice to implementors.* The parameter `params` can be used to pass arbitrary information on database creation, such as graph name, maximum storage size, storage directory, maximum used system memory, fault tolerance behavior and so on. (*End of advice to implementors.*)

GDI_CreateDatabase is a collective call.

> *Advice to implementors.* We do not explicitly prescribe a barrier semantic to enable potential optimizations, but GDI_CreateDatabase may come with such a semantic, if a particular implementation deems it necessary. (*End of advice to implementors.*)

```
int GDI_FreeDatabase( GDI_Database* graph_db )
```

INOUT	graph_db	graph database object (handle)

GDI_FreeDatabase deallocates all internal data structures and memory associated with the given graph database object. GDI_FreeDatabase will set `graph_db` to GDI_DATABASE_NULL. Additionally all objects (labels, property types, indexes and transactions) that are associated with the graph database `graph_db` will be freed as well, as if the user had called the respective destruction function for each of the handles prior to the call of GDI_FreeDatabase. GDI_FreeDatabase is a collective call and `graph_db` should be the same on all processes.

5 Labels

Labels provide the notion of categorization to vertices and edges. A label has a unique name. A label handle acts as identifier and is taken as input for other functions. A vertex or edge can have zero, one or more labels. Labels are opaque objects, which are stored locally on each process.

> *Rationale.* Usually, there is only a small set of labels in a graph database. Storing the label objects locally on each process enables to query labels without relying on other processes. (*End of rationale.*)

Labels represent the set L of the labeled property graph model. Labels and property types can be seen as metadata, since they both describe the respective set of possibilities for a given graph and not what is actually present in the graph data (vertex and edges). GDI only guarantees eventual consistency for graph metadata, so usually one of the first steps of creating a graph database is the creation of labels and property types followed by some form of synchronization before the first single process transactions are initiated.

GDI offers one predefined label which can be accessed by the GDI_LABEL_NONE handle. It can be used to index vertices and edges which do not have a label assigned (see Section 9.2). The name assigned to this object is "GDI_LABEL_NONE".

```
int GDI_CreateLabel( const char* name, GDI_Database graph_db, GDI_Label* label )
```

IN	name	character string stored as the name (string)
INOUT	graph_db	graph database object (handle)
OUT	label	label object returned by the call (handle)

GDI_CreateLabel associates a label with the graph database `graph_db` and allocates a label object. GDI_CreateLabel has to be called before the label will be used for the first time. GDI_CreateLabel is a collective call and all input parameters have to be same on all processes.

The label is named to allow retrieval of handles later. The GDI library will locally store the character string of `name`, so `name` can be allocated on the stack or otherwise immediately deallocated after the call by the caller. Trailing spaces in `name` will be ignored, while leading spaces are relevant. The limit of the name length is GDI_MAX_OBJECT_NAME-1, so that an additional null terminator can be stored. If the user tries to store names longer than this, then the name will be truncated. If the label name already exists in the graph database, the error GDI_ERROR_NAME_EXISTS is returned. If `name` contains an empty string, the error GDI_ERROR_EMPTY_NAME is returned.

The call is erroneous if `label` is a predefined label.

```
int GDI_FreeLabel( GDI_Label* label )
```

INOUT	label	label object (handle)

GDI_FreeLabel removes the given label from its associated graph database. `label` should be the same on all processes. The function will deallocate the label object and set `label` to GDI_LABEL_NULL. All vertices or edges with this given label will get the label removed, but remain in the database. If the label is still associated with any explicit indexes, GDI_FreeLabel will, similarly to GDI_RemoveLabelFromIndex, remove the association of the label with the indexes and entries in the indexes will be updated accordingly (see Section 9.2). Additionally, any GDI_Constraint and GDI_Subconstraint object that contains a condition that uses `label` will get those conditions marked as stale and in turn also the whole object will be marked as stale.

GDI_FreeLabel is a collective call and will synchronize all processes of the associated graph database. GDI_FreeLabel is an implicit collective write transaction, so all transactions on the associated graph database must be finished before a process enters the GDI_FreeLabel call and no other transactions on that graph database may be started until the GDI_FreeLabel call returns.

A call to GDI_FreeLabel has a barrier semantic: a process returns from the call only after all other processes have not only entered their matching call, but also finished the respective changes to the graph database.

The call is erroneous if `label` is a predefined label. If after the label is removed, the graph database contains any vertices without any labels and the same application level ID, GDI_WARNING_NON_UNIQUE_ID is returned.

> *Advice to users.* GDI_FreeLabel might warrant extensive changes to the graph database, so its use might be expensive in terms of performance. (*End of advice to users.*)

> *Advice to implementors.* The functionality of GDI_FreeLabel is provided for completeness of supported graph database operations, but has rarely any use cases in real world scenarios. Because of the extensive necessary changes to the database, it will be complex to implement. (*End of advice to implementors.*)

```
int GDI_UpdateLabel( const char* name, GDI_Label label )
```

IN	name	character string stored as the name (string)
INOUT	label	label object (handle)

GDI_UpdateLabel updates the name of the given label. It is a collective call and all input parameters have to be same on all processes.

The GDI library will locally store the character string of `name`, so `name` can be allocated on the stack or otherwise immediately deallocated after the call by the caller. Trailing spaces in `name` will be ignored, while leading spaces are relevant. The limit of the name length is GDI_MAX_OBJECT_NAME-1, so that an additional null terminator can be stored. If the user tries to store names longer than this, then the name will be truncated. If the name already exists on a different label of the graph database, the error GDI_ERROR_NAME_EXISTS is returned. If `name` contains an empty string, the error GDI_ERROR_EMPTY_NAME is returned.

If `name` and the name already associated with `label` are the same, no action is performed. The call is erroneous if `label` is a predefined label.

```
int GDI_GetLabelFromName( GDI_Label* label, const char* name,
         GDI_Database graph_db )
```

OUT	label	label object (handle)
IN	name	a character string which represents a label name (string)
IN	graph_db	graph database object (handle)

Given the label name as parameter `name` and the graph database handle as parameter `graph_db`, the function GDI_GetLabelFromName looks up the label handle. If the name is not found, `label` contains GDI_LABEL_NULL. The length of the input string `name` should be at most GDI_MAX_OBJECT_NAME-1. GDI_GetLabelFromName is a local call.

```
int GDI_GetNameOfLabel( char* name, size_t length, size_t* resultlength,
         GDI_Label label )
```

OUT	name	stored name for the label (string)
IN	length	maximum length of name (non-negative integer)
OUT	resultlength	length of the returned name (non-negative integer)
IN	label	label whose associated name is retrieved (handle)

GDI_GetNameOfLabel retrieves the name associated with `label`. `length` denotes the length of the allocated string `name`. The buffer to which `name` points to should be able to hold at least GDI_MAX_OBJECT_NAME Bytes. `resultlength` contains on return the length of the retrieved string. `name[resultlength]` contains an additional null terminator. Therefore the returned value of `resultlength` is at most GDI_MAX_OBJECT_NAME-1. If the allocated string is smaller than the actual label name, the string will be filled, such that a valid UTF-8 string is returned, and the remaining characters will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. If any other error occurs, GDI_GetNameOfLabel will return an empty string. GDI_GetNameOfLabel is a local call.

> *Rationale.* Vertices and edges store only the handle of a label. GDI_GetLabelName allows the user to get the associated label name for easier identification. (*End of rationale.*)

```
int GDI_GetAllLabelsOfDatabase( GDI_Label array_of_labels[], size_t count,
        size_t* resultcount, GDI_Database graph_db )
```

OUT	array_of_labels	array of labels (array of handles)
IN	count	length of array_of_labels (non-negative integer)
OUT	resultcount	number of retrieved labels (non-negative integer)
IN	graph_db	graph database object (handle)

A user might not know what labels are available in a certain graph database object. The function GDI_GetAllLabelsOfDatabase will retrieve all labels currently associated to the given graph database `graph_db`. The user provides an array for label handles and the parameter `count`, which contains the maximum number of label handles that can be written to said array. The parameter `resultcount` contains the actual number of label handles written to `array_of_labels`. If the array is smaller than the available number of label handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllLabelsOfDatabase is a local call.

6 Properties

A property is a tuple (key, value), where key $\in K$ and value $\in W$ (cf. Section 1.2), and can be assigned to vertices and edges. The key identifies a property and the value is the according property value. GDI allows that a key can refer to multiple values.

> *Rationale.* GDI bases on the labeled property graph model, in which a tuple (key, value) is unique per object (vertex/edge). This implies that a key can refer to multiple different values. (*End of rationale.*)

A property type describes the key of a property, the datatype of the elements of the property value and possible limitations. Property types are opaque objects, which are stored locally on each process.

> *Rationale.* Usually, there is only a small set of property types in a graph database. Storing the property type objects locally on each process enables to query information about the property types without relying on other processes. (*End of rationale.*)

Property types represent the set K of the labeled property graph model. Labels and property types can be seen as metadata, since they both describe the respective set of possibilities for a given graph and not what is actually present in the graph data (vertex and edges). GDI only guarantees eventual consistency for graph metadata, so usually one of the first steps of creating a graph database is the creation of labels and property types followed by some form of synchronization before the first single process transactions are initiated.

6.1 Property Type Creation, Destruction and Update

```
int GDI_CreatePropertyType( const char* name, int etype, GDI_Datatype dtype,
        int stype, size_t count, GDI_Database graph_db,
        GDI_PropertyType* ptype )
```

IN	name	character string stored as the name (string)
IN	etype	entity type (state)
IN	dtype	datatype object (handle)
IN	stype	type of the size limitation (state)
IN	count	number of elements (positive integer)
INOUT	graph_db	graph database object (handle)
OUT	ptype	property type object returned by the call (handle)

GDI_CreatePropertyType associates a new property type with the graph database `graph_db` and allocates a property type object. GDI_CreatePropertyType has to be called before the property type will be used for the first time. GDI_CreatePropertyType is a collective call and all input parameters should be the same on all processes.

The property type is named to allow retrieval of the handles later. The GDI library will locally store the character string of `name`, so `name` can be allocated on the stack or otherwise immediately deallocated after the call by the caller. Trailing spaces in `name` will be ignored, while leading spaces are relevant. The limit of the name length is GDI_MAX_OBJECT_NAME-1, so that an additional null terminator can be stored. If the user tries to store names longer than this, then the name will be truncated. If the property type name already exists in the graph database, the error GDI_ERROR_NAME_EXISTS is returned. If `name` contains an empty string, the error GDI_ERROR_EMPTY_NAME is returned.

The call is erroneous if `ptype` is a predefined property type.

GDI allows to impose certain limitations on the property type to enable optimizations. The state variable `etype` informs the library on how often entries of the property type will occur. `etype` is restricted to the two values GDI_SINGLE_ENTITY and GDI_MULTIPLE_ENTITY. A property type with the GDI_SINGLE_ENTITY attribute will only have one property entry on a single object (vertex/edge), while a property type with the GDI_MULTIPLE_ENTITY attribute allows multiple entries of the same property type on a single object.

The datatype provided by the parameter `dtype` informs the library about the datatype of the elements of the property value and enables the library to perform operations on property values.

The parameters `stype` and `count` work in conjunction. The state variable `stype` informs the library whether the property type has a certain size limitation. `stype` is restricted to the three values GDI_FIXED_SIZE, GDI_MAX_SIZE and GDI_NO_SIZE_LIMIT. If GDI_FIXED_SIZE is provided, the property value will always occupy a fixed amount of space. Such a property value will always consists of `count` elements (inclusive) of the datatype `dtype`. The library will enforce that limitation and return an error, if the user attempts to add a property of type `ptype` to an object with a different amount of elements. If GDI_MAX_SIZE is provided instead, the library allows the user to store up to `count` elements (inclusive) of the datatype `dtype` as the value of a property of type `ptype`. If the user attempts to store more than `count` elements, the library will return an error. If GDI_NO_SIZE_LIMIT is provided in `stype`, then no limitation on the size of the value of a property of type `ptype` is enforced and `count` is ignored by the library.

In all cases, the elements are stored consecutively.

> *Rationale.* These limitations are provided, so that the library can use the additional information for optimizations. However it imposes no restrictions on the use of the LPG model in GDI, since the user can always provide GDI_MULTIPLE_ENTITY for `etype` and GDI_NO_SIZE_LIMIT for `stype` to fully support the LPG model. (*End of rationale.*)

```
int GDI_FreePropertyType( GDI_PropertyType* ptype )
```

INOUT	ptype	property type object (handle)

GDI_FreePropertyType removes the given property type `ptype` from its associated graph database. `ptype` should be the same on all processes. The function will deallocate the property type object and set the parameter `ptype` to GDI_PROPERTY_TYPE_NULL. All vertices or edges with properties of the given property type will get those properties removed, but remain in the database. If `ptype` is still associated with any explicit indexes, GDI_FreePropertyType will, similarly to GDI_RemovePropertyTypeFromIndex, remove the association of the property type with the indexes and entries in the indexes will be updated accordingly (see Section 9.3). Additionally, any GDI_Constraint and GDI_Subconstraint object that contains a condition that uses `ptype` will get those conditions marked as stale and in turn also the whole object will be marked as stale.

GDI_FreePropertyType is a collective call and will synchronize all processes of the associated graph database. GDI_FreePropertyType is an implicit collective write transaction, so all transactions on the associated graph database must be finished before a process enters the GDI_FreePropertyType call and no other transactions on that graph database may be started until the GDI_FreePropertyType call returns. A call to GDI_FreePropertyType has a barrier semantic: a process returns from the call only after all other processes have not only entered their matching call, but also finished the respective changes to the graph database.

The call is erroneous if `ptype` is a predefined property type.

> *Advice to users.* GDI_FreePropertyType might warrant extensive changes to the graph database, so its use might be expensive in terms of performance. (*End of advice to users.*)

> *Advice to implementors.* The functionality of GDI_FreePropertyType is provided for completeness of supported graph database operations, but has rarely any use cases in real world scenarios. Because of the extensive necessary changes to the database, it will be complex to implement. (*End of advice to implementors.*)

```
int GDI_UpdatePropertyType( const char* name, int etype, GDI_Datatype dtype,
        int stype, size_t count, const void* default_value,
        GDI_PropertyType ptype )
```

IN	name	character string stored as the name (string)
IN	etype	entity type (state)
IN	dtype	datatype object (handle)
IN	stype	type of the size limitation (state)
IN	count	number of elements (positive integer)
IN	default_value	initial address of default value (choice)
INOUT	ptype	property type object (handle)

GDI_UpdatePropertyType updates the attributes of the property type `ptype`. Additionally all property entries on vertices and edges of the given property type will be updated accordingly. All input parameters should be the same on all processes.

The GDI library will locally store the character string of `name`, so `name` can be allocated on the stack or otherwise immediately deallocated after the call by the caller. Trailing spaces in `name` will be ignored, while leading spaces are relevant. The limit of the name length is GDI_MAX_OBJECT_NAME-1, so that an additional null terminator can be stored. If the user tries to store names longer than this, then the name will be truncated. If the name already exists on a different property type of the graph database, the error GDI_ERROR_NAME_EXISTS is returned. If `name` contains an empty string, the error GDI_ERROR_EMPTY_NAME is returned.

If the entity type is updated from GDI_MULTIPLE_ENTITY to GDI_SINGLE_ENTITY and there is more than one property of the given type on an object (vertex/edge), all properties of that type except for one arbitrary one will be removed from said object. In the opposite direction (from GDI_SINGLE_ENTITY to GDI_MULTIPLE_ENTITY), no changes on the objects are necessary.

The updated datatype is given by the parameter `dtype`. The elements of the property values are updated by the conversion rules from Section 10.8. If the requested conversion is not valid in GDI, the error GDI_ERROR_CONVERSION is returned.

The type of the size limitation is given by `stype` and the number of elements by `count`. Certain combinations of the old and new size limitation parameters require changes in the number of elements of the property values. For the ease of the explanation below, we will call old_count the number of elements that was previously associated with `ptype`.

If the property type was a fixed sized type before and will remain so, the change depends entirely on the amount of elements. If `count` is bigger than old_count, then additional elements need to be added at the end of the property value, so that a total number of `count` elements is reached. The value of each newly added element will be taken from `default_value`. So the first old_count elements will retain their value, while the following (`count`-old_count) elements will have the value of `default_value`. If `count` is smaller than old_count, then (old_count-`count`) trailing elements will be dropped, so only `count` elements remain, which will keep their value. The same changes have to be applied to all properties of type `ptype`. If `count` is equal to old_count, no changes to the property values are necessary.

If the property type was a fixed sized type before and will become a maximum sized type, then there will be only changes to the property values if `count` is smaller than old_count. In such a case, the (old_count-`count`) trailing elements of the property value will be dropped, so only `count` elements remain, which will keep their value. The same changes have to be applied to all properties of type `ptype`.

If the property type was a maximum sized type before and will remain so, and `count` is equal to or bigger than old_count, no changes to the property values are necessary. If `count` is smaller than old_count, the number of elements of each property value, that are of property type `ptype`, is significant. If the number of elements is equal or smaller to `count`, no changes are necessary. If however the number of elements is bigger, then the trailing elements will be dropped, so that only `count` elements remain, which will keep their value.

If the property type was a maximum sized type before and will become a fixed sized type, changes might be different for each property of type `ptype`. If `count` is bigger than old_count, then additional elements need to be added at the end of the value of each property of that type, so that the total number of elements reaches `count`. The value of each newly added element will be taken from `default_value`, while the original elements will retain their value. The amount of newly added elements might be different for each property of that type. If `count` is equal to old_count, the same changes as just described have to happen, however if a property of that type has the maximum number of elements possible, no changes to its property value are necessary. If `count` is smaller than old_count, then all kinds of changes are possible. If the number of elements of a property value, that is of property type `ptype`, is smaller than `count`, then additional elements have to be added at the end of the value, so that the total number of elements reaches `count`. The value of each newly added element will be taken from `default_value`, while the original elements will retain their value. If the number of elements is equal to `count`, no changes are necessary. If the number of elements is bigger than `count`, then trailing elements will be dropped, until only `count` elements remain, which will keep their value.

If property type `ptype` had previously no limitations (the previous `stype` had the value GDI_NO_SIZE_LIMIT), and will become either a fixed sized or a maximum sized type, then each property of that type has to be checked, whether it adheres to the new size limitations. If the property type will become maximum sized, and the number of elements of the value of a property of such a type is equal to or smaller than `count`, no changes are necessary. If the number of elements is bigger, then trailing elements will be dropped, until only `count` elements remain, which will keep their value. For a property type that will become fixed sized, changes are more complicated. If the number of elements of the value of a property of such a type is smaller than `count`, new elements with the value of `default_value` will be added at the end of the property value, so that a total number of `count` elements is reached. If the number of elements is equal to `count`, no changes are necessary. If the number of elements is bigger than `count`, then trailing elements will be dropped, until only `count` elements remain, which will keep their value.

If `stype` has the value GDI_NO_SIZE_LIMIT, then no changes are necessary.

In all cases, the elements are stored consecutively.

The entry of vertices and edges in any index associated with `ptype` might be updated. Additionally, any GDI_Constraint and GDI_Subconstraint object that contains a condition that uses `ptype` will get those conditions marked as stale and in turn also the whole object will be marked as stale.

GDI_UpdatePropertyType is a collective call and will synchronize all processes of the associated graph database. GDI_UpdatePropertyType is an implicit collective write transaction, so all transactions on the associated graph database must be finished before a process enters the GDI_UpdatePropertyType call and no other transactions on that graph database may be started until the GDI_UpdatePropertyType call returns. A call to GDI_UpdatePropertyType has a barrier semantic: a process returns from the call only after all other processes have not only entered their matching call, but also finished the respective changes to the graph database.

The call is erroneous if `ptype` is a predefined property type.

> *Advice to users.* GDI_UpdatePropertyType might warrant extensive changes to the graph database, so its use might be expensive in terms of performance. (*End of advice to users.*)

> *Advice to implementors.* The functionality of GDI_UpdatePropertyType is provided for completeness of supported graph database operations, but has rarely any use cases in real world scenarios. Because of the extensive necessary changes to the database, it will be complex to implement. (*End of advice to implementors.*)

6.2 Predefined Property Types

GDI offers a number of predefined property types: for application level ID, degree, indegree and outdegree. Predefined property types work with any graph database and do not have to be associated first.

The predefined property type GDI_PROPERTY_TYPE_ID, which can be accessed by the static handle of the same name, can be used to access the application level ID of an object, mostly of vertices. The property is usually set during the initial GDI_CreateVertex call and it is also used by the implicit index for GDI_TranslateVertexID, which translates application level IDs into internal vertex UIDs. While edges typically do not have (U)IDs, GDI_PROPERTY_TYPE_ID can be used to explicitly set application level IDs for edges. The name assigned to this opaque object is "GDI_PROPERTY_TYPE_ID". GDI_PROPERTY_TYPE_ID is a single entity property type (`etype` is GDI_SINGLE_ENTITY). Its datatype is GDI_BYTE and GDI imposes no limit on the size of the application level ID (`stype` is GDI_NO_SIZE_LIMIT).

In graph theory, the degree specifies the number of edges that a vertex has. It is the sum of the number of incoming edges, the number of outgoing edges and the number of undirected edges. Loops (edges that connect a vertex to itself) will be counted twice. GDI has the predefined property type GDI_PROPERTY_TYPE_DEGREE, which can be accessed by the static handle of the same name, to access the degree of a vertex. GDI_PROPERTY_TYPE_DEGREE can't be used on edges. The name assigned to this opaque object is "GDI_PROPERTY_TYPE_DEGREE". It is a single entity property type (`etype` is GDI_SINGLE_ENTITY), the datatype is GDI_UINT64_T and it is a fixed sized property type (`stype` is GDI_FIXED_SIZE) with exactly one element of GDI_UINT64_T (`count` has the value 1). GDI_PROPERTY_TYPE_DEGREE is a read-only property type and entries are implicitly updated by the library, when edges are added or removed.

The indegree of a vertex is the number of incoming edges. The property type to access the indegree of a vertex is GDI_PROPERTY_TYPE_INDEGREE, which can be accessed by the static handle of the same name. GDI_PROPERTY_TYPE_INDEGREE can't be used on edges. The name assigned to this opaque object is "GDI_PROPERTY_TYPE_INDEGREE". It is a single entity property type (`etype` is GDI_SINGLE_ENTITY), the datatype is GDI_UINT64_T and it is a fixed sized property type (`stype` is GDI_FIXED_SIZE) with exactly one element of GDI_UINT64_T (`count` has the value 1). GDI_PROPERTY_TYPE_INDEGREE is a read-only property type and entries are implicitly updated by the library, when edges are added or removed.

Similarly, the outdegree of a vertex is the number of outgoing edges. The property type to access the outdegree of a vertex is GDI_PROPERTY_TYPE_OUTDEGREE, which can be accessed by the static handle of the same name. GDI_PROPERTY_TYPE_OUTDEGREE can't be used on edges. Its assigned name assigned is "GDI_PROPERTY_TYPE_OUTDEGREE". It is a single entity property type (`etype` is GDI_SINGLE_ENTITY), the datatype is GDI_UINT64_T and it is a fixed sized property type (`stype` is GDI_FIXED_SIZE) with exactly one element of GDI_UINT64_T (`count` has the value 1). GDI_PROPERTY_TYPE_OUTDEGREE is a read-only property type and entries are implicitly updated by the library, when edges are added or removed.

6.3 Property Type Retrieval

```
int GDI_GetPropertyTypeFromName( GDI_PropertyType* ptype, const char* name,
        GDI_Database graph_db )
```

OUT	ptype	property type object (handle)
IN	name	a character string which represents a property type name (string)
IN	graph_db	graph database object (handle)

Given the property type name as parameter `name` and the graph database handle as parameter `graph_db`, the function GDI_GetPropertyTypeFromName looks up the property type handle for that name. If the name is not found, `ptype` contains GDI_PROPERTY_TYPE_NULL. The length of the input string `name` should be at most GDI_MAX_OBJECT_NAME-1 Bytes. GDI_GetPropertyTypeFromName is a local call.

```
int GDI_GetAllPropertyTypesOfDatabase( GDI_PropertyType array_of_ptypes[],
        size_t count, size_t* resultcount, GDI_Database graph_db )
```

OUT	array_of_ptypes	array of property types (array of handles)
IN	count	length of array_of_ptypes (non-negative integer)
OUT	resultcount	number of retrieved property types (non-negative integer)
IN	graph_db	graph database object (handle)

A user might not know what property types are available in a certain graph database object. GDI_GetAllPropertyTypesOfDatabase will retrieve all property types associated to the given graph database `graph_db`. The user provides an array for property type handles and the parameter `count`, which contains the maximum number of property type handles that can be written to said array. On return `resultcount` contains the actual number of property type handles written to `array_of_ptypes`. If the array is smaller than the available number of property type handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllPropertyTypesOfDatabase is a local call.

6.4 Property Type Attributes

```
int GDI_GetNameOfPropertyType( char* name, size_t length, size_t* resultlength,
        GDI_PropertyType ptype )
```

OUT	name	stored name for the property type (string)
IN	length	maximum length of name (non-negative integer)
OUT	resultlength	length of the returned name (non-negative integer)
IN	ptype	property type whose associated name is retrieved (handle)

GDI_GetNameOfPropertyType retrieves the name associated with `ptype`. `length` denotes the length of the allocated string `name`. The buffer to which `name` points to should be able to hold at GDI_MAX_OBJECT_NAME Bytes. `resultlength` contains on return the length of the retrieved string. `name[resultlength]` contains an additional null terminator. Therefore the returned value of `resultlength` is at most GDI_MAX_OBJECT_NAME-1. If the allocated string is smaller than the actual property type name, the string will be filled, such that a valid UTF-8 string is returned, and the remaining characters will be omitted. In such an overflow case the error GDI_ERROR_TRUNCATE will be returned. If any other error occurs, GDI_GetNameOfPropertyType will return an empty string. GDI_GetNameOfPropertyType is a local call.

> *Rationale.* Vertices and edges store only the handle of a property type. The function GDI_GetNameOfPropertyType allows the user to get the associated property type name for easier identification. (*End of rationale.*)

```
int GDI_GetEntityTypeOfPropertyType( int* etype, GDI_PropertyType ptype )
```

OUT	etype	entity type (state)
IN	ptype	property type object (handle)

GDI_GetEntityTypeOfPropertyType returns the entity type of the property type `ptype`. The returned state variable `etype` can have exactly two values: GDI_SINGLE_ENTITY and GDI_MULTIPLE_ENTITY. GDI_GetEntityTypeOfPropertyType is a local call.

```
int GDI_GetDatatypeOfPropertyType( GDI_Datatype* dtype, GDI_PropertyType ptype )
```

OUT	dtype	datatype object (handle)
IN	ptype	property type object (handle)

GDI_GetDatatypeOfPropertyType retrieves the datatype of the given property type `ptype`. It is a local call.

```
int GDI_GetSizeLimitOfPropertyType( int* stype, size_t* count,
         GDI_PropertyType ptype )
```

OUT	stype	size limitation type (state)
OUT	count	number of elements (non-negative integer)
IN	ptype	property type object (handle)

GDI_GetSizeLimitOfPropertyType returns the size limitations associated with the given property type `ptype`. The state variable `stype` can have exactly three values: GDI_FIXED_SIZE, GDI_MAX_SIZE and GDI_NO_SIZE_LIMIT. Additionally the parameter `count` is returned. If GDI_FIXED_SIZE is returned in `stype`, then `count` contains the number of elements that will always make up the value of a property of type `ptype`. If GDI_MAX_SIZE is returned in `stype`, `count` contains the maximum number of elements that can make up the value of a property of type `ptype`. If `stype` contains GDI_NO_SIZE_LIMIT, then no limitation is imposed on the number of elements for the value of a property of type `ptype` and `count` is set to the value 0, but that value can be ignored by the caller.

7 Vertices

Vertices are represented in GDI during a transaction as temporary GDI_VertexHolder objects. GDI_VertexHolder objects are only valid during the transaction, in which they were created. Usually one of the first steps in a transaction is to either create a new vertex via GDI_CreateVertex or to access an existing vertex by associating it with a GDI_VertexHolder object by calling GDI_AssociateVertex. The GDI_VertexHolder object handles all communication that is involved when querying incident edges, vertex properties and vertex labels.

> *Advice to implementors.* GDI_VertexHolder objects serve as access objects, identifying uniquely a vertex in the database during a transaction. They don't mandate a specific implementation, e.g. it is possible to directly access and manipulate the data in the database or to cache a local copy of the respective data to reduce communication. (*End of advice to implementors.*)

All functions in this section can return a transaction-critical error.

7.1 Temporary Vertex Object Creation

```
int GDI_CreateVertex( const void* external_id, size_t size,
         GDI_Transaction transaction, GDI_VertexHolder* vertex )
```

IN	external_id	initial address of application level ID (choice)
IN	size	size of application level ID (non-negative integer)
INOUT	transaction	transaction object (handle)
OUT	vertex	temporary vertex object returned by the call (handle)

GDI_CreateVertex allocates a temporary representation of a vertex. `vertex` is not associated with any edges or labels yet. GDI_CreateVertex should only be called inside a transaction.

If an application level ID is provided, it will be stored as a property with the predefined type GDI_PROPERTY_TYPE_ID. Additionally an implicit index used for GDI_TranslateVertexID will be updated on commit of the transaction. Other properties are not yet associated with `vertex`.

If the application does not use IDs for its vertices, it should provide NULL in the parameter `external_id` and the value 0 in the parameter `size`. The application will not be able to access these vertices directly using GDI_TranslateVertexID during a later transaction, instead it either has to find the vertices through graph exploration or by using explicit indexes.

```
int GDI_AssociateVertex( GDI_Vertex_uid internal_uid,
         GDI_Transaction transaction, GDI_VertexHolder* vertex )
```

IN	internal_uid	internal vertex UID (UID)
INOUT	transaction	transaction object (handle)
OUT	vertex	temporary vertex object returned by the call (handle)

GDI_AssociateVertex allocates a GDI_VertexHolder object and associates said object with a (remote) vertex location, provided by `internal_uid`. Afterwards the GDI_VertexHolder object can be used to query its edges, labels and properties. GDI_AssociateVertex should only be called inside a transaction. If `internal_uid` does not belong to the same graph database as `transaction` does, the error GDI_ERROR_OBJECT_MISMATCH is returned.

Each GDI_VertexHolder object is associated with a transaction and will be invalidated, once that transaction is committed or aborted.

7.2 Vertex Destruction

`int GDI_FreeVertex( GDI_VertexHolder* vertex )`

INOUT	vertex	vertex object (handle)

GDI_FreeVertex removes the vertex from the graph database upon transaction commit. It deallocates the temporary vertex object and sets `vertex` to GDI_VERTEX_NULL. The temporary vertex object can't be queried afterwards, even if the transaction is still ongoing. Additionally all edges that have this vertex as origin or target will be removed. If the transaction contains GDI_EdgeHolder objects representing any of those edges, those objects will be invalidated and can't be accessed while the transaction is still ongoing. GDI_FreeVertex should only be called during a transaction. The function removes the vertex and associated edges from all associated indexes during the commit call of the transaction.

7.3 Vertex Edge Handling

```
int GDI_GetEdgesOfVertex( GDI_Edge_uid array_of_uids[], size_t count,
         size_t* resultcount, GDI_Constraint constraint,
         int edge_orientation, GDI_VertexHolder vertex )
```

OUT	array_of_uids	array of internal edge UIDs (array of UIDs)
IN	count	length of array_of_uids (non-negative integer)
OUT	resultcount	number of retrieved UIDs (non-negative integer)
IN	constraint	constraint object (handle)
IN	edge_orientation	edge orientation (integer)
IN	vertex	vertex object (handle)

GDI_GetEdgesOfVertex queries the temporary vertex object `vertex` and returns the internal edge UIDs of all incident edges that satisfy the edge orientation given by the parameter `edge_orientation` and the conditions set by the constraint object that `constraint` points to. The internal edge UIDs then can be used to access the edges with GDI_EdgeHolder objects. The internal edge UIDs will be returned in the array `array_of_uids`, where `count` contains the maximum number of internal edge UIDs that can be written to said array. On return `resultcount` contains the actual number of internal edge UIDs written to `array_of_uids`. If the array is smaller than the available number of internal edge UIDs, the array will be filled and the remaining internal edge UIDs will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. If `constraint` does not belong to the same graph database as `vertex` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_GetEdgesOfVertex should only be called during a transaction and all retrieved internal edge UIDs are only valid during the transaction from which GDI_GetEdgesOfVertex is called.

GDI provides a way to filter the edges incident to `vertex` with the use of a constraint object and their orientation. The `edge_orientation` is a bitwise OR combination of the following integer constants to provide various modes of edge filtering:

- GDI_EDGE_INCOMING: keep all incoming edges
- GDI_EDGE_OUTGOING: keep all outgoing edges
- GDI_EDGE_UNDIRECTED: keep all undirected edges

It is possible to retrieve all edges of `vertex` by supplying the value GDI_CONSTRAINT_NULL in the parameter `constraint` and a bitwise OR of the constants GDI_EDGE_INCOMING, GDI_EDGE_OUTGOING and GDI_EDGE_UNDIRECTED as argument for `edge_orientation`. GDI_ERROR_EDGE_ORIENTATION is returned as error, in case `edge_orientation` is not valid. If a stale GDI_Constraint object is passed, GDI_ERROR_STALE is returned as error.

```
int GDI_GetNeighborVerticesOfVertex( GDI_Vertex_uid array_of_uids[],
         size_t count, size_t* resultcount, GDI_Constraint constraint,
         int edge_orientation, GDI_VertexHolder vertex )
```

OUT	array_of_uids	array of internal vertex UIDs (array of UIDs)
IN	count	length of array_of_uids (non-negative integer)
OUT	resultcount	number of retrieved UIDs (non-negative integer)
IN	constraint	constraint object (handle)
IN	edge_orientation	edge orientation (integer)
IN	vertex	vertex object (handle)

GDI_GetNeighborVerticesOfVertex queries the temporary vertex object `vertex` and returns the internal vertex UIDs of the set of all vertices adjacent to `vertex` that are incident to edges which satisfy the edge orientation given by the parameter `edge_orientation` and the conditions set by the constraint object that `constraint` points to. The internal vertex UIDs then can be used to access the vertices with GDI_VertexHolder objects. The internal vertex UIDs will be returned in the array `array_of_uids`, where `count` contains the maximum number of internal vertex UIDs that can be written to said array. On return `resultcount` contains the actual number of internal vertex UIDs written to `array_of_uids`. If the array is smaller than the available number of internal vertex UIDs, the array will be filled and the remaining internal vertex UIDs will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. If `constraint` does not belong to the same graph database as `vertex` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_GetNeighborVerticesOfVertex should only be called during a transaction and all retrieved internal vertex UIDs are only valid during the transaction from which GDI_GetNeighborVerticesOfVertex is called.

The edges incident to `vertex` are filtered according to their orientation with additional limitations imposed by the use of the constraint object. The `edge_orientation` is a bitwise OR combination of the following integer constants to provide various modes of edge filtering:

- GDI_EDGE_INCOMING: keep all incoming edges
- GDI_EDGE_OUTGOING: keep all outgoing edges
- GDI_EDGE_UNDIRECTED: keep all undirected edges

The resulting set of edges is queried for the other incident vertex and then further refined, so that the result in `array_of_uids` is a set of internal vertex UIDs, meaning each vertex only appears once.

Retrieval of all adjacent vertices is enabled by supplying the value GDI_CONSTRAINT_NULL in the parameter `constraint` and a bitwise OR of the constants GDI_EDGE_INCOMING, GDI_EDGE_OUTGOING and GDI_EDGE_UNDIRECTED as argument for `edge_orientation`. GDI_ERROR_EDGE_ORIENTATION is returned as error, in case `edge_orientation` is not valid. If a stale GDI_Constraint object is passed, GDI_ERROR_STALE is returned as error.

7.4 Vertex Label Handling

A vertex can have an arbitrary number of labels, including no labels at all.

```
int GDI_AddLabelToVertex( GDI_Label label, GDI_VertexHolder vertex )
```

IN	label	label object (handle)
INOUT	vertex	vertex object (handle)

GDI_AddLabelToVertex adds `label` to `vertex`. If `vertex` has already the given label or the predefined label GDI_LABEL_NONE is supplied, no action is performed. If `label` does not belong to the same graph database as `vertex` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. If `label` is already associated with an object with the same application level ID, then the label will not be added to `vertex` and the error GDI_ERROR_NON_UNIQUE_ID is returned. GDI_AddLabelToVertex should only be called during a transaction. The function might prompt an update of explicit indexes. The vertex will be added to the indexes associated with `label`, in case the vertex is not already part of said indexes (because of other labels). In case the vertex didn't have any label before this function call, the vertex will also be removed from the indexes associated with GDI_LABEL_NONE if said indexes do not have `label` associated. All of those operations will be done during the commit call of the transaction.

```
int GDI_RemoveLabelFromVertex( GDI_Label label, GDI_VertexHolder vertex )
```

IN	label	label object (handle)
INOUT	vertex	vertex object (handle)

GDI_RemoveLabelFromVertex removes `label` from `vertex`. If the specified label is not associated with the vertex or the predefined label GDI_LABEL_NONE is supplied, no action is performed. GDI_RemoveLabelFromVertex should only be called during a transaction. The function might prompt an update of explicit indexes. The vertex will be removed from the indexes associated with the label, if there are no additional common labels. If `label` was the only label of the vertex, the vertex will be added to any indexes associated with GDI_LABEL_NONE. All of those operations will be done during the commit call of the transaction.

If `label` was the only label of `vertex`, then the vertex will have no label afterwards. GDI allows that multiple vertices without any labels have the same application level ID. If the call to GDI_RemoveLabelFromVertex will result in such a case, it will remove the label, but also return GDI_WARNING_NON_UNIQUE_ID.

> *Advice to users.* If the user program relies on the fact, that vertices have to have unique IDs for querying the implicit index, it is possible to remedy such a situation by adding a (dummy) label to the vertex during the same transaction. (*End of advice to users.*)

```
int GDI_GetAllLabelsOfVertex( GDI_Label array_of_labels[], size_t count,
         size_t* resultcount, GDI_VertexHolder vertex )
```

OUT	array_of_labels	array of label objects (array of handles)
IN	count	length of array_of_labels (non-negative integer)
OUT	resultcount	number of retrieved labels (non-negative integer)
IN	vertex	vertex object (handle)

GDI_GetAllLabelsOfVertex will retrieve all labels that are currently associated with `vertex`. The user provides an array for label handles and the parameter `count`, which contains the maximum number of label handles that can be written to said array. On return, `resultcount` contains the actual number of label handles written to `array_of_labels`. If the array is smaller than the available number of label handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllLabelsOfVertex should only be called during a transaction.

7.5 Vertex Property Handling

```
int GDI_AddPropertyToVertex( const void* value, size_t count,
         GDI_PropertyType ptype, GDI_VertexHolder vertex )
```

IN	value	initial address to the value (choice)
IN	count	number of elements (non-negative integer)
IN	ptype	property type object (handle)
INOUT	vertex	vertex object (handle)

GDI_AddPropertyToVertex adds a new property of type `ptype` to the given vertex. `count` elements of the datatype associated with `ptype` will be read from the address given by the parameter `value` and stored in the vertex. Any size limitation of the property type `ptype` will be enforced. If `ptype` is a single entity property type, and a property of that type already exists on the vertex, the error GDI_ERROR_PROPERTY_TYPE_EXISTS will be returned. If `ptype` does not belong to the same graph database as `vertex` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. If there is a property with the same property type and value already present on the vertex, no action is performed (multiple entries of the same (key,value)-pair are not allowed in the LPG model). GDI_AddPropertyToVertex should only be called during a transaction. The function might prompt an update of explicit indexes. The vertex will be added to the indexes associated with `ptype`, in case the vertex is not already part of said indexes (because of other properties). The update of the indexes will be done during the commit call of the transaction.

```
int GDI_GetAllPropertyTypesOfVertex( GDI_PropertyType array_of_ptypes[],
         size_t count, size_t* resultcount, GDI_VertexHolder vertex )
```

OUT	array_of_ptypes	array of property type objects (array of handles)
IN	count	length of array_of_ptypes (non-negative integer)
OUT	resultcount	number of retrieved property types (non-negative integer)
IN	vertex	vertex object (handle)

GDI_GetAllPropertyTypesOfVertex will retrieve all property types, that have at least one property of that type present on `vertex`. The user provides an array for property type handles `array_of_ptypes` and the parameter `count`, which contains the maximum number of property type handles that can be written to said array. On return `resultcount` contains the actual number of property type handles written to `array_of_ptypes`. If the array is smaller than the available number of property type handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllPropertyTypesOfVertex should only be called during a transaction.

GDI_GetAllPropertyTypesOfVertex does not return the predefined degree property types, since they are present on each vertex by default.

```
int GDI_GetPropertiesOfVertex( void* buf, size_t buf_count,
         size_t* buf_resultcount, size_t array_of_offsets[], size_t offset_count,
         size_t* offset_resultcount, GDI_PropertyType ptype,
         GDI_VertexHolder vertex )
```

OUT	buf	initial address of buffer (choice)
IN	buf_count	length of buf (non-negative integer)
OUT	buf_resultcount	number of retrieved elements in buf (non-negative integer)
OUT	array_of_offsets	array of buffer offsets (array of non-negative integers)
IN	offset_count	length of array_of_offsets (non-negative integer)
OUT	offset_resultcount	number of retrieved offsets (non-negative integer)
IN	ptype	property type object (handle)
IN	vertex	vertex object (handle)

GDI_GetPropertiesOfVertex retrieves all properties of type `ptype` from the given vertex. The values of the properties will be stored in the buffer `buf` with `buf_count` specifying the maximum number of elements of the datatype associated with `ptype` that will fit into `buf`. On return `buf_resultcount` contains the actual number of elements of the datatype associated with `ptype` that are written to `buf`. The offset of each property value will be returned in `array_of_offsets`. The offsets will be specified in number of elements of the datatype associated with the property type `ptype`. The parameter `offset_count` contains the maximum number of offsets, that can be written to `array_of_offsets`. On return, `offset_resultcount` contains the actual number of entries written to `array_of_offsets`. If `vertex` contains n properties of type `ptype`, then `offset_resultcount` will be set to $n + 1$. The first n entries in `array_of_offsets` contain the offset where the respective property value in `buf` begins. The last entry of `array_of_offsets` contains the total number of elements written. This construction enables to determine the number of elements of the i-th property value in `buf` by calculating `array_of_offsets[`$i+1$`] - array_of_offsets[`i`]`.

> *Rationale.* The last entry of `array_of_offsets` and `buf_resultcount` both determine the total number of elements written to `buf`. The parameter `buf_resultcount` is required in the function interface to return the number of elements when a null pointer is given for `buf`, or `array_of_offset` or 0 is provided for `offset_count`, or `buf_count` (Section 2.6.2). (*End of rationale.*)

If no properties of type `ptype` are present on the given vertex, `offset_resultcount` will be set to value 0 and nothing will be written to `buf` and `array_of_offsets`. If `array_of_offsets` is smaller than the number of property values to be returned, the array will be filled and the remaining offsets will be omitted. Similarly, if buffer `buf` is too small to hold all property values, the buffer will be filled and the remaining property values will be omitted. The error GDI_ERROR_TRUNCATE will be returned in both overflow cases. If `ptype` does not belong to the same graph database as `vertex` does, the error GDI_ERROR_OBJECT_MISMATCH is returned.

GDI_GetPropertiesOfVertex should only be called during a transaction.

```
int GDI_RemovePropertiesFromVertex( GDI_PropertyType ptype,
         GDI_VertexHolder vertex )
```

IN	ptype	property type object (handle)
INOUT	vertex	vertex object (handle)

GDI_RemovePropertiesFromVertex will remove all properties of type `ptype` from the given vertex. If there is no property of type `ptype` on the vertex, no action will be performed. GDI_RemovePropertiesFromVertex should only be called during a transaction. If any properties were removed, the function might prompt an update of explicit indexes. The vertex will be removed from the indexes associated with `ptype`, if there are no common additional properties. The update of the indexes will be done during the commit call of the transaction.

```
int GDI_RemoveSpecificPropertyFromVertex( const void* value, size_t count,
         GDI_PropertyType ptype, GDI_VertexHolder vertex )
```

IN	value	initial address to the value (choice)
IN	count	number of elements (non-negative integer)
IN	ptype	property type object (handle)
INOUT	vertex	vertex object (handle)

GDI_RemoveSpecificPropertyFromVertex will remove a specific property of type `ptype`. The function will remove a property only if the number of elements of its property value, where one element is of the datatype associated with `ptype`, is equal to `count` and the value of each element of that property value matches exactly the respective element in the data pointed to by the parameter `value`. If there is no property of type `ptype` or no property of that type, whose property value matches `value`, no action will be performed. At most one property will be removed, since multiple entries of the same (key, value)-pair are not allowed in the LPG model. GDI_RemoveSpecificPropertyFromVertex should only be called during a transaction. The function might prompt an update of explicit indexes. If only one property of type `ptype` was present on the vertex and that property was removed, the vertex will be removed from the indexes associated with the property type, if there are no common additional property types associated with said indexes. If there were multiple properties of that type present and a property was removed, the associated indexes will be updated. All of those operations will be done during the commit call of the transaction.

```
int GDI_UpdatePropertyOfVertex( const void* value, size_t count,
        GDI_PropertyType ptype, GDI_VertexHolder vertex )
```

IN	value	initial address to the value (choice)
IN	count	number of elements (non-negative integer)
IN	ptype	single entity property type object (handle)
INOUT	vertex	vertex object (handle)

GDI_UpdatePropertyOfVertex updates the property of a single entity property type `ptype` on `vertex`. If `ptype` is a multi entity property type, the error GDI_ERROR_WRONG_TYPE is returned. The new property value is copied from the data pointed to by `value`. The number of elements to be copied, which are of the datatype associated with `ptype`, is provided by the `count` parameter. Any size limitation of the property type `ptype` regarding the new value will be enforced. If no property of type `ptype` exists on the given vertex, the error GDI_ERROR_NO_PROPERTY is returned. If `ptype` does not belong to the same graph database as `vertex` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. If the existing property of type `ptype` has the same property value as `value`, no action is performed. GDI_UpdatePropertyOfVertex should only be called during a transaction. The entry of the vertex in any associated index will be updated during the commit call of the transaction.

```
int GDI_UpdateSpecificPropertyOfVertex( const void* old_value, size_t old_count,
        const void* new_value, size_t new_count, GDI_PropertyType ptype,
        GDI_VertexHolder vertex )
```

IN	old_value	initial address to the value for comparison (choice)
IN	old_count	number of elements in old_value (non-negative integer)
IN	new_value	initial address to the new value (choice)
IN	new_count	number of elements in new_value (non-negative integer)
IN	ptype	property type object (handle)
INOUT	vertex	vertex object (handle)

GDI_UpdateSpecificPropertyOfVertex updates a property of type `ptype` on `vertex`, only if its property value matches the content of `old_value`. The function will update a property only if the number of elements of its property value, where one element is of the datatype associated with `ptype`, is equal to `old_count` and the value of each element of that property value matches exactly the respective element in the data pointed to by the parameter `old_value`. If such a property is found, the old property value is removed and the new property value is copied from the data to which `new_value` points to, with the number of elements of the new property value being specified by `new_count`. If no property is updated, the error GDI_ERROR_NO_PROPERTY is returned. Any size limitation of the property type `ptype` regarding the new property value will be enforced. At most one property will be updated, since multiple entries of the same (key,value)-pair are not allowed in the LPG model. For the same reason, the error GDI_ERROR_PROPERTY_EXISTS will be returned, in case there is a property with the same property type and value, matching `new_value`, already present on the vertex. If `ptype` does not belong to the same graph database as `vertex` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_UpdateSpecificPropertyOfVertex should only be called during a transaction. The entry of the vertex in any associated index will be updated during the commit call of the transaction.

> *Rationale.* GDI_UpdateSpecificPropertyOfVertex returns an error, if the new property already exists on the vertex, instead of performing no action, to avoid non-intuitive situations, where a successful update would decrease the total number of properties on a vertex instead of keeping it constant. (*End of rationale.*)

```
int GDI_SetPropertyOfVertex( const void* value, size_t count,
          GDI_PropertyType ptype, GDI_VertexHolder vertex )
```

IN	value	initial address to the value (choice)
IN	count	number of elements (non-negative integer)
IN	ptype	single entity property type object (handle)
INOUT	vertex	vertex object (handle)

GDI_SetPropertyOfVertex sets the property of a single entity property type `ptype` on `vertex`. If `ptype` is a multi entity property type, the error GDI_ERROR_WRONG_TYPE is returned. The new value of the property is copied from the data pointed to by `value`. The number of elements to be copied, which are of the datatype associated with `ptype`, is provided by the `count` parameter. If a property of type `ptype` already exists on the given vertex, then the value of that property will be overwritten. Otherwise the property will be added to the vertex. Any size limitation of the property type `ptype` regarding the new property value will be enforced. If `ptype` does not belong to the same graph database as `vertex` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_SetPropertyOfVertex should only be called during a transaction. The function might prompt an update of explicit indexes. In case a property of the vertex was updated, the entry of the vertex in the associated indexes will be updated as well. In case a property was added, the vertex will be added to the indexes associated with the property type `ptype`, in case the vertex is not already part of said indexes (because of other properties). All of those operations will be done during the commit call of the transaction.

> *Advice to implementors.* An implementation might map GDI_SetPropertyOfVertex to a sequence of function calls: Remove the property from a vertex and then insert the property again. However, there might be a faster way to achieve this. (*End of advice to implementors.*)

8 Edges

Edges are represented in GDI during a transaction as temporary GDI_EdgeHolder objects. GDI_EdgeHolder objects are only valid during the transaction, in which they were created. Edges can either be obtained by querying vertices or explicit indexes. Those edges then can be associated with a GDI_EdgeHolder object by calling GDI_AssociateEdge. Another possibility is to create a new edge with GDI_CreateEdge. The GDI_EdgeHolder object handles all communication that is involved when querying incident vertices, edge properties and edge labels.

> *Rationale.* GDI_EdgeHolder objects serve as access objects, identifying uniquely an edge in the database during a transaction. They don't mandate a specific implementation, e.g. it is possible to directly access and manipulate the data in the database or to cache a local copy of the respective data to reduce communication. (*End of rationale.*)

All functions in this section can return a transaction-critical error.

8.1 Temporary Edge Object Creation

```
int GDI_CreateEdge( int dtype, GDI_VertexHolder origin, GDI_VertexHolder target,
        GDI_EdgeHolder* edge )
```

IN	dtype	direction type (state)
IN	origin	vertex object (handle)
IN	target	vertex object (handle)
OUT	edge	temporary edge object returned by the call (handle)

GDI_CreateEdge allocates a temporary representation of an edge, which represents a newly created connection from vertex `origin` to vertex `target`. The state parameter `dtype` is restricted to two values and indicates whether the edge is directed (GDI_EDGE_DIRECTED) or undirected (GDI_EDGE_UNDIRECTED). No labels or properties are associated with `edge` yet. The error GDI_ERROR_OBJECT_MISMATCH is returned, if `origin` and `target` do not belong to the same transaction. GDI_CreateEdge should only be called during a transaction.

```
int GDI_AssociateEdge( GDI_Edge_uid internal_uid, GDI_Transaction transaction,
        GDI_EdgeHolder* edge )
```

IN	internal_uid	internal edge UID (UID)
INOUT	transaction	transaction object (handle)
OUT	edge	temporary edge object returned by the call (handle)

GDI_AssociateEdge allocates a GDI_EdgeHolder object and associates said object with a (remote) edge location, provided by `internal_uid`. Afterwards the GDI_EdgeHolder object can be used to query its incident vertices, labels and properties. If `internal_uid` does not belong to the same graph database as `transaction` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_AssociateEdge should only be called during a transaction.

Each GDI_EdgeHolder object is associated with a transaction and will be invalidated, once that transaction is committed or aborted.

8.2 Edge Destruction

```
int GDI_FreeEdge( GDI_EdgeHolder* edge )
```

INOUT	edge	edge object (handle)

GDI_FreeEdge removes the edge from the graph database upon transaction commit. It deallocates the temporary edge object and sets `edge` to GDI_EDGE_NULL. The temporary edge object can't be queried afterwards, even if the transaction is still ongoing. GDI_FreeEdge should only be called during a transaction. The function removes the edge from all associated indexes during the commit call of the transaction.

8.3 Edge Attributes

```
int GDI_GetVerticesOfEdge( GDI_Vertex_uid* origin_uid, GDI_Vertex_uid* target_uid,
        GDI_EdgeHolder edge )
```

OUT	origin_uid	internal vertex UID of the origin (UID)
OUT	target_uid	internal vertex UID of the target (UID)
IN	edge	edge object (handle)

GDI_GetVerticesOfEdge will return in `origin_uid` the internal vertex UID of the vertex, from which `edge` originates and in `target_uid` the internal vertex UID of the vertex, which `edge` targets. If `edge` is undirected, a fixed order of the two vertices will be returned, such that multiple calls of GDI_GetVerticesOfEdge with the same unchanged edge will always return the same result. If a null pointer is passed for either UID parameter, the function does not return the respective internal vertex UID. GDI_GetVerticesOfEdge should only be called during a transaction.

```
int GDI_GetDirectionTypeOfEdge( int* dtype, GDI_EdgeHolder edge )
```

OUT	dtype	direction type (state)
IN	edge	edge object (handle)

GDI_GetDirectionTypeOfEdge returns whether `edge` is directed (GDI_EDGE_DIRECTED) or undirected (GDI_EDGE_UNDIRECTED). The state parameter `dtype` is restricted to those two values. GDI_GetDirectionTypeOfEdge should only be called during a transaction.

```
int GDI_SetOriginVertexOfEdge( GDI_VertexHolder origin_vertex,
        GDI_EdgeHolder edge )
```

IN	origin_vertex	vertex object (handle)
INOUT	edge	edge object (handle)

GDI_SetOriginVertexOfEdge updates the origin vertex of `edge`. If `edge` is undirected, a fixed one of the two vertices will be replaced, namely the one that would have been returned as origin internal vertex UID by a previous call to GDI_GetVerticesOfEdge. If `origin_vertex` and `edge` do not belong to the same transaction, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_SetOriginVertexOfEdge should only be called during a transaction. The function might prompt an update of explicit indexes.

```
int GDI_SetTargetVertexOfEdge( GDI_VertexHolder target_vertex,
        GDI_EdgeHolder edge )
```

IN	target_vertex	vertex object (handle)
INOUT	edge	edge object (handle)

GDI_SetTargetVertexOfEdge updates the target vertex of `edge`. If `edge` is undirected, a fixed one of the two vertices will be replaced, namely the one that would have been returned as target internal vertex UID by a previous call to GDI_GetVerticesOfEdge. If `target_vertex` and `edge` do not belong to the same transaction, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_SetTargetVertexOfEdge should only be called during a transaction. The function might prompt an update of explicit indexes.

`int GDI_SetDirectionTypeOfEdge( int dtype, GDI_EdgeHolder edge )`

IN	dtype	direction type (state)
INOUT	edge	edge object (handle)

GDI_SetDirectionTypeOfEdge updates, whether `edge` is directed (GDI_EDGE_DIRECTED) or undirected (GDI_EDGE_UNDIRECTED). The state parameter `dtype` is restricted to those two values. GDI_SetDirectionTypeOfEdge should only be called during a transaction. The function might prompt an update of explicit indexes.

8.4 Edge Label Handling

An edge can have an arbitrary number of labels, including no labels at all.

`int GDI_AddLabelToEdge( GDI_Label label, GDI_EdgeHolder edge )`

IN	label	label object (handle)
INOUT	edge	edge object (handle)

GDI_AddLabelToEdge adds `label` to `edge`. If the edge has already the given label or the predefined label GDI_LABEL_NONE is supplied, no action is performed. If `label` does not belong to the same graph database as `edge` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. If there is already an object with the same `label` and application level ID present in the database, then the label will not be added to `edge` and the error GDI_ERROR_NON_UNIQUE_ID is returned.

> *Advice to users.* GDI does not enforce that an edge needs an application level ID. A user can add an application level ID by adding a property (see Section 8.5) using the predefined property type GDI_PROPERTY_TYPE_ID. (*End of advice to users.*)

GDI_AddLabelToEdge should only be called during a transaction. The function might prompt an update of explicit indexes. The edge will be added to the indexes associated with `label`, in case the edge is not already part of said indexes (because of other labels). In case the edge didn't have any label before this function call, the edge will also be removed from the indexes associated with GDI_LABEL_NONE if said indexes do not have `label` associated. All of those operations will be done during the commit call of the transaction.

`int GDI_RemoveLabelFromEdge( GDI_Label label, GDI_EdgeHolder edge )`

IN	label	label object (handle)
INOUT	edge	edge object (handle)

GDI_RemoveLabelFromEdge removes `label` from `edge`. If the specified label is not present in the edge object or the predefined label GDI_LABEL_NONE is supplied, no action is performed. GDI_RemoveLabelFromEdge should only be called during a transaction. The function might prompt an update of explicit indexes. The edge will be removed from the indexes associated with the label, if there are no additional common labels. If `label` was the only label of the edge, the edge will be added to any indexes associated with GDI_LABEL_NONE. All of those operations will be done during the commit call of the transaction.

```
int GDI_GetAllLabelsOfEdge( GDI_Label array_of_labels[], size_t count,
        size_t* resultcount, GDI_EdgeHolder edge )
```

OUT	array_of_labels	array of label objects (array of handles)
IN	count	length of array_of_labels (non-negative integer)
OUT	resultcount	number of retrieved labels (non-negative integer)
IN	edge	edge object (handle)

GDI_GetAllLabelsOfEdge will retrieve all labels that are currently associated with the temporary edge object `edge`. The user provides an array for label handles and the parameter `count`, which contains the maximum number of label handles that can be written to said array. On return, `resultcount` contains the actual number of label handles written to `array_of_labels`. If the array is smaller than the available number of label handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllLabelsOfEdge should only be called during a transaction.

8.5 Edge Property Handling

```
int GDI_AddPropertyToEdge( const void* value, size_t count,
        GDI_PropertyType ptype, GDI_EdgeHolder edge )
```

IN	value	initial address to the value (choice)
IN	count	number of elements (non-negative integer)
IN	ptype	property type object (handle)
INOUT	edge	edge object (handle)

GDI_AddPropertyToEdge adds a new property of type `ptype` to the given edge. `count` elements of the datatype associated with `ptype` will be read from the address given by the parameter `value` and stored in the edge. Any size limitation of the property type `ptype` will be enforced. If `ptype` is a single entity property type, and a property of that type already exists on the edge, the error GDI_ERROR_PROPERTY_TYPE_EXISTS will be returned. If `ptype` does not belong to the same graph database as `edge` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. If there is a property with the same property type and value already present on the edge, no action is performed (multiple entries of the same (key,value)-pair are not allowed in the LPG model). GDI_AddPropertyToEdge should only be called during a transaction. The function might prompt an update of explicit indexes. The edge will be added to the indexes associated with `ptype`, in case the edge is not already part of said indexes (because of other properties). The update of the indexes will be done during the commit call of the transaction.

```
int GDI_GetAllPropertyTypesOfEdge( GDI_PropertyType array_of_ptypes[],
        size_t count, size_t* resultcount, GDI_EdgeHolder edge )
```

OUT	array_of_ptypes	array of property types (array of handles)
IN	count	length of array_of_ptypes (non-negative integer)
OUT	resultcount	number of retrieved property types (non-negative integer)
IN	edge	edge object (handle)

GDI_GetAllPropertyTypesOfEdge will retrieve all property types, that have at least one property of that type present on `edge`. The user provides an array for property type handles `array_of_ptypes` and the parameter `count`, which contains the maximum number of property type handles that can be written to said array. On return `resultcount` contains the actual number of property type handles written to `array_of_ptypes`. If the array is smaller than the available number of property type handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllPropertyTypesOfEdge should only be called during a transaction.

```
int GDI_GetPropertiesOfEdge( void* buf, size_t buf_count,
        size_t* buf_resultcount, size_t array_of_offsets[], size_t offset_count,
        size_t* offset_resultcount, GDI_PropertyType ptype,
        GDI_VertexEdge edge )
```

OUT	buf	initial address of buffer (choice)
IN	buf_count	length of buf (non-negative integer)
OUT	buf_resultcount	number of retrieved elements in buf (non-negative integer)
OUT	array_of_offsets	array of buffer offsets (array of non-negative integers)
IN	offset_count	length of array_of_offsets (non-negative integer)
OUT	offset_resultcount	number of retrieved offsets (non-negative integer)
IN	ptype	property type object (handle)
IN	edge	edge object (handle)

GDI_GetPropertiesOfEdge retrieves all properties of type `ptype` from the given edge. The values of the properties will be stored in the buffer `buf` with `buf_count` specifying the maximum number of elements of the datatype associated with `ptype` that will fit into `buf`. On return `buf_resultcount` contains the actual number of elements of the datatype associated with `ptype` that are written to `buf`. The offset of each property value will be returned in `array_of_offsets`. The offsets will be specified in number of elements of the datatype associated with the property type `ptype`. The parameter `offset_count` contains the maximum number of offsets, that can be written to `array_of_offsets`. On return, `offset_resultcount` contains the actual number of entries written to `array_of_offsets`. If `edge` contains n properties of type `ptype`, then `offset_resultcount` will be set to $n+1$. The first n entries in `array_of_offsets` contain the offset where the respective property value in `buf` begins. The last entry of `array_of_offsets` contains the total number of elements written. This construction enables to determine the number of elements of the i-th property value in `buf` by calculating `array_of_offsets`[$i+1$] - `array_of_offsets`[i].

> *Rationale.* The last entry of `array_of_offsets` and `buf_resultcount` both determine the total number of elements written to `buf`. The parameter `buf_resultcount` is required in the function interface to return the number of elements when a null pointer is given for `buf`, or `array_of_offset` or 0 is provided for `offset_count`, or `buf_count` (Section 2.6.2). (*End of rationale.*)

If no properties of type `ptype` are present on the given edge, `offset_resultcount` will be set to value 0 and nothing will be written to `buf` and `array_of_offsets`. If `array_of_offsets` is smaller than the number of property values to be returned, the array will be filled and the remaining offsets will be omitted. Similarly, if buffer `buf` is too small to hold all property values, the buffer will be filled and the remaining property values will be omitted. The error GDI_ERROR_TRUNCATE will be returned in both overflow cases. If `ptype` does not belong to the same graph database as `edge` does, the error GDI_ERROR_OBJECT_MISMATCH is returned.

GDI_GetPropertiesOfEdge should only be called during a transaction.

```
int GDI_RemovePropertiesFromEdge( GDI_PropertyType ptype, GDI_EdgeHolder edge )
```

IN	ptype	property type object (handle)
INOUT	edge	edge object (handle)

GDI_RemovePropertiesFromEdge will remove all properties of type `ptype` from the given edge. If there is no property of type `ptype` on the edge, no action will be performed. The function GDI_RemovePropertiesFromEdge should only be called during a transaction. If any properties were removed, the function might prompt an update of explicit indexes. The edge will be removed from the indexes associated wh `ptype`, if there are no common additional properties. The update of the indexes will be done during the commit call of the transaction.

```
int GDI_RemoveSpecificPropertyFromEdge( const void* value, size_t count,
        GDI_PropertyType ptype, GDI_EdgeHolder edge )
```

IN	value	initial address to the value (choice)
IN	count	number of elements (non-negative integer)
IN	ptype	property type object (handle)
INOUT	edge	edge object (handle)

GDI_RemoveSpecificPropertyFromEdge will remove a specific property of type `ptype`. The function will remove a property only if the number of elements of its property value, where one element is of the datatype associated with `ptype`, is equal to `count` and the value of each element of that property value matches exactly the respective element in the data pointed to by the parameter `value`. If there is no property of type `ptype` or no property of that type, whose property value matches `value`, no action will be performed. At most one property will be removed, since multiple entries of the same (key, value)-pair are not allowed in the LPG model. GDI_RemoveSpecificPropertyFromEdge should only be called during a transaction. The function might prompt an update of explicit indexes. If only one property of type `ptype` was present on the edge and that property was removed, the vertex will be removed from the indexes associated with the property type, if there are no common additional property types associated with said indexes. If there were multiple properties of that type present and a property was removed, the associated indexes will be updated. All of those operations will be done during the commit call of the transaction.

```
int GDI_UpdatePropertyOfEdge( const void* value, size_t count,
        GDI_PropertyType ptype, GDI_EdgeHolder edge )
```

IN	value	initial address to the value (choice)
IN	count	number of elements (non-negative integer)
IN	ptype	single entity property type object (handle)
INOUT	edge	edge object (handle)

GDI_UpdatePropertyOfEdge updates the property of a single entity property type `ptype` on `edge`. If `ptype` is a multi entity property type, the error GDI_ERROR_WRONG_TYPE is returned. The new property value is copied from the data pointed to by `value`. The number of elements to be copied, which are of the datatype associated with `ptype`, is provided by the `count` parameter. Any size limitation of the property type `ptype` regarding the new value will be enforced. If no property of type `ptype` exists on the given edge, the error GDI_ERROR_NO_PROPERTY is returned. If `ptype` does not belong to the same graph database as `edge` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. If the existing property of type `ptype` has the same property value as `value`, no action is performed. GDI_UpdatePropertyOfEdge should only be called during a transaction. The entry of the edge in any associated index will be updated during the commit call of the transaction.

```
int GDI_UpdateSpecificPropertyOfEdge( const void* old_value, size_t old_count,
        const void* new_value, size_t new_count, GDI_PropertyType ptype,
        GDI_EdgeHolder edge )
```

IN	old_value	initial address to the value for comparison (choice)
IN	old_count	number of elements in old_value (non-negative integer)
IN	new_value	initial address to the new value (choice)
IN	new_count	number of elements in new_value (non-negative integer)
IN	ptype	property type object (handle)
INOUT	edge	edge object (handle)

GDI_UpdateSpecificPropertyOfEdge updates a property of type `ptype` on `edge`, only if its property value matches the content of `old_value`. The function will update a property only if the number of elements of its property value, where one element is of the datatype associated with `ptype`, is equal to `old_count` and the value of each element of that property value matches exactly the respective element in the data pointed to by the parameter `old_value`. If such a property is found, the old property value is removed and the new property value is copied from the data to which `new_value` points to, with the number of elements of the new property value being specified by `new_count`. If no property is updated, the error GDI_ERROR_NO_PROPERTY is returned. Any size limitation of the property type `ptype` regarding the new property value will be enforced. At most one property will be updated, since multiple entries of the same (key,value)-pair are not allowed in the LPG model. For the same reason, the error GDI_ERROR_PROPERTY_EXISTS will be returned, in case there is a property with the same property type and value, matching `new_value`, already present on the edge. If `ptype` does not belong to the same graph database as `edge` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_UpdateSpecificPropertyOfEdge should only be called during a transaction. The entry of the edge in any associated index will be updated during the commit call of the transaction.

> *Rationale.* GDI_UpdateSpecificPropertyOfEdge returns an error, if the new property already exists on the edge, instead of performing no action, to avoid non-intuitive situations, where a successful update would decrease the total number of properties on an edge instead of keeping it constant. (*End of rationale.*)

```
int GDI_SetPropertyOfEdge( const void* value, size_t count,
         GDI_PropertyType ptype, GDI_EdgeHolder edge )
```

IN	value	initial address to the value (choice)
IN	count	number of elements (non-negative integer)
IN	ptype	single entity property type object (handle)
INOUT	edge	edge object (handle)

GDI_SetPropertyOfEdge sets the property of a single entity property type `ptype` on `edge`. If `ptype` is a multi entity property type, the error GDI_ERROR_WRONG_TYPE is returned. The new value of the property is copied from the data pointed to by `value`. The number of elements to be copied, which are of the datatype associated with `ptype`, is provided by the `count` parameter. If a property of type `ptype` already exists on the given edge, then the value of that property will be overwritten. Otherwise the property will be added to the edge. Any size limitation of the property type `ptype` regarding the new property value will be enforced. If `ptype` does not belong to the same graph database as `edge` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_SetPropertyOfEdge should only be called during a transaction. The function might prompt an update of explicit indexes. In case a property of the edge was updated, the entry of the edge in the associated indexes will be updated as well. In case a property was added, the edge will be added to the indexes associated with the property type `ptype`, in case the edge is not already part of said indexes (because of other properties). All of those operations will be done during the commit call of the transaction.

> *Advice to implementors.* An implementation might map GDI_SetPropertyOfEdge to a sequence of function calls: Remove the property from an edge and then insert the property again. However, there might be a faster way to achieve this. (*End of advice to implementors.*)

9 Indexes

For certain types of workloads, graph databases must provide a fast look-up of objects (vertices and edges) given a label or a property. To solve this use case, GDI provides an interface for indexes for fast look-up, similar to indexes in relational databases. These interfaces are meant to provide explicit indexes: An application might know the graph and its queries better than the graph database. Therefore providing an interface to set the indexes explicitly instead of implicitly (by assumptions of the implementation of GDI) might improve performance.

> *Advice to users.* Indexes are "not free": they use memory and updating them may result in additional complexity or communication. (*End of advice to users.*)

> *Advice to implementors.* Explicit indexes impose no restrictions on what indexes the implementation of GDI might use internally. (*End of advice to implementors.*)

GDI provides access to vertices and their attributes (labels, properties) with a two step mechanism. In the first step an internal UID, a GDI specific UID, of datatype GDI_Vertex_uid is obtained. This internal UID identifies a vertex in the GDI database uniquely. In the second step, that internal vertex UID is used to create a temporary GDI_VertexHolder object, that represents the vertex on the process. The GDI_VertexHolder object then can be used to query the edges, labels and properties of the vertex. Internal vertex UIDs are only valid during the transaction from which they are obtained to allow the relocation of vertices. The process of accessing is split in two parts to make it possible to distribute internal vertex UIDs, which are relatively small, to other processes during a collective read transaction for load-balancing purposes, while GDI_VertexHolder objects are opaque, making them local unshareable objects. Additionally the internal vertex UIDs are also used to identify the target vertex of an edge.

Similarly, GDI also uses a two step mechanism to access edges and their attributes. The first step is an internal UID (of datatype GDI_Edge_uid). This internal UID identifies an edge in the GDI database uniquely. In the second step, that internal edge UID is used to create a temporary GDI_EdgeHolder object, that represents the edge on the process. The GDI_EdgeHolder object then can be used to query the origin and target vertices, labels and properties of the edge. Internal edge UIDs are only valid during the transaction from which they are obtained to allow the relocation of edges. Again this two step mechanism allows the distribution of internal edge UIDs to other processes.

With graph databases there are usually two use cases: 1) smaller queries that usually only visit a small part of the graph and 2) complex queries that usually involve the complete graph or at least a big part of the graph.

In the first case one usually starts from one vertex and then explores the graph by following edges. To have access to that first vertex, GDI provides a function to translate application level IDs to internal vertex UIDs with an implicit index provided by the library (see Section 9.5.1). Afterwards the graph can be explored without querying indexes by following other internal vertex UIDs which are provided by edges.

Complex queries usually involve most of the graph and can be implemented by using collective read transactions in conjunction with explicit indexes. Those queries typically start with a set of objects that fit certain conditions. The application can select the make up of explicit indexes to fit its needs and then query an application selected index in a scalable way to filter objects to meet those conditions, and get returned that starting set of objects, upon which it can start its graph algorithm/exploration. If the application uses a query optimizer, it is that optimizer's responsibility to choose the best index for the task.

GDI has the following index model: An explicit index I has an associated set of labels L and a set of associated property types P. An object o (a vertex or an edge) has a set of associated labels o_L and a set of associated property types o_P. The object o is indexed by I iff $L \cap o_L \neq \emptyset$ (or $L = \emptyset$) and $P \cap o_P \neq \emptyset$ (or $P = \emptyset$) and $L \cup P \neq \emptyset$.

> *Rationale.* The index model is an intuitive extension of indexes in RDBMS: One can think that an object o is stored in tables given by the labels o_L assigned to o. The values properties of o_P are stored in columns. An index I with only labels L denotes the collection of the objects over the tables L. If the index has only associated properties P,

then the index is over all columns P (ignoring NULL values). If the index has associated labels L and property types P, then the index I can be seen as an index over the columns P of the tables L (ignoring NULL values). (*End of rationale.*)

9.1 Explicit Index Creation and Destruction

```
int GDI_CreateIndex( size_t obj_count, int itype, GDI_Database graph_db,
         GDI_Index* index )
```

IN	obj_count	hint on how many objects will be indexed (non-negative integer)
IN	itype	type of index to create (state)
IN	graph_db	graph database object (handle)
OUT	index	index object returned by the call (handle)

GDI_CreateIndex allocates an index object of the given type `itype`. Objects of the graph database `graph_db` will be indexed by `index`. `obj_count` should be considered a hint to the library, on the global number of objects, that will be indexed by this index object. If the value 0 is passed as `obj_count`, the library will consider this as no hint given. The state parameter `itype` is restricted to the two values GDI_INDEXTYPE_HASHTABLE and GDI_INDEXTYPE_BTREE. GDI_INDEXTYPE_HASHTABLE is intended to be implemented as a hash-table for 1:1 relations with an equal and not equal look-up. GDI_INDEXTYPE_BTREE is intended to be implemented as a B-tree for range and comparison functions. On return of the GDI_CreateIndex call, the index object contains no entries. It is a collective call and all input parameters should be the same on all processes. A call to GDI_CreateIndex has a barrier semantic: a process returns from the call only after all other processes have entered their matching call.

The addition and removal of entries to the index(es) is handled implicitly during the commit call of a transaction.

```
int GDI_FreeIndex( GDI_Index* index )
```

INOUT	index	index object (handle)

GDI_FreeIndex deallocates an index object, and sets `index` to GDI_INDEX_NULL. `index` should be the same on all processes. It is a collective call with a barrier semantic: a process returns from the call only after all other processes have entered their matching call.

If `index` is still associated with any labels and/or property types, then calling GDI_FreeIndex will have the same effect, as if the user would have removed all labels/property types from `index` by himself.

9.2 Index Label Handling

```
int GDI_AddLabelToIndex( GDI_Label label, GDI_Index index )
```

INOUT	label	label object (handle)
INOUT	index	index object (handle)

GDI_AddLabelToIndex will associate a given label with an explicit index. If `label` is already associated with `index`, no action is performed. If `label` does not belong to the same graph database as `index` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. All parameters should be the same on all processes. GDI_AddLabelToIndex is a collective call.

If `index` is only associated with other labels, then all objects with the label `label` will be added as entries to the index, unless they are already present in this index, because of other labels. However if there are also property types associated with `index`, then only objects with the label `label`, which also contain at least one property of the property types associated with `index`, will be added as entries to the index, unless they are already present. If `index` is only associated with property types, adding a label can reduce the amount of objects that are indexed, because an object has to be part of both sets, the label set and the property type set, to be indexed. It is possible to have indexes that are only associated with one of the sets, making them pure label indexes or pure property type indexes.

It is possible to index objects with no label using the predefined label GDI_LABEL_NONE.

Explicit indexes will be updated implicitly during the commit of a transaction, if objects got labels, associated with `index`, added or removed during the transaction.

The index can be used for a fast look-up of objects with labels and properties associated with the index.

> *Advice to users.* Indexes introduce additional complexity during transactions, so if a label is assigned to too many indexes, there might be too much overhead without any performance gain. However this depends on the behavior of the implementation, so the user is advised to check the documentation of the implementation. (*End of advice to users.*)

`int GDI_RemoveLabelFromIndex( GDI_Label label, GDI_Index index )`

INOUT	label	label object (handle)
INOUT	index	index object (handle)

GDI_RemoveLabelFromIndex removes the association of the given label with an explicit index it was previously registered to. If `label` was not associated with `index`, no action is performed. GDI_RemoveLabelFromIndex is a collective call and all parameters should be the same on all processes.

All objects with the label `label`, which have an entry in the index `index`, will be removed from the index, unless they have at least one additional label, that is still associated with `index`. If `label` is the only label associated with `index`, but there are still property types associated with this index, then no entries will be removed from the index. Instead all objects, which contain at least one property of the property types associated with `index`, but are currently not present in this index, because they didn't have the label `label`, will be added as entries to `index`. If `label` is the only label associated with `index` and additionally no property types are associated with this index, then `index` contains no entries after the return of the call.

If the predefined label GDI_LABEL_NONE is supplied, objects with no label will be removed from the index according to the rules mentioned above.

9.3 Index Property Type Handling

`int GDI_AddPropertyTypeToIndex( GDI_PropertyType ptype, GDI_Index index )`

INOUT	ptype	property type object (handle)
INOUT	index	index object (handle)

GDI_AddPropertyTypeToIndex will associate a given property type with an explicit index. If `ptype` is already associated with `index`, no action is performed. If `ptype` does not belong to the same graph database as `index` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_AddPropertyTypeToIndex is a collective call and all parameters should be the same on all processes.

If `index` is only associated with other property types, then all objects with properties of property type `ptype` will be added as entries to the index, unless they are already present in this index, because of other property types. However if there are also labels associated with `index`, then only objects with properties of the property type `ptype`, which also have at least one label of the labels associated with `index`, will be added as entries to the index, unless they are already present. If `index` is only associated with labels, adding a property type can reduce the amount of objects that are indexed, because an object has to be part of both sets, the label set and the property type set, to be indexed. It is possible to have indexes that are only associated with one of the sets, making them pure label indexes or pure property type indexes.

Explicit indexes will be updated implicitly during the commit of a transaction, if objects got properties, whose property type is associated with `index`, added or removed during the transaction.

The index can be used for a fast look-up of objects with labels and properties associated with the index.

> *Advice to users.* Indexes introduce additional complexity during transactions, so if a property type is assigned to too many indexes, there might be too much overhead without any performance gain. However this depends on the behavior of the implementation, so the user is advised to check the documentation of the implementation. (*End of advice to users.*)

`int GDI_RemovePropertyTypeFromIndex( GDI_PropertyType ptype, GDI_Index index )`

INOUT	ptype	property type object (handle)
INOUT	index	index object (handle)

GDI_RemovePropertyTypeFromIndex removes the association of the given property type with an explicit index, it was previously registered to. If `ptype` was not associated with `index`, no action is performed. GDI_RemovePropertyTypeFromIndex is a collective call and all parameters should be the same on all processes.

All objects with the property type `ptype`, which have an entry in the index `index`, will be removed from the index, unless they have at least one additional property, whose property type is still associated with `index`. If `ptype` is the only property type associated with this index, but there are still labels associated with this index, then no entries will be removed from the index. Instead all objects, which contain at least one label of the labels associated with `index`, but are currently not present in this index, because they didn't have properties of property type `ptype`, will be added as entries to `index`. If `ptype` is the only property type associated with index and additionally no labels are associated with this index, then `index` contains no entries after the return of the call.

9.4 Index Bulk Update

```
int GDI_AddLabelsAndPropertyTypesToIndex( GDI_Label array_of_labels[],
         size_t label_count, GDI_PropertyType array_of_ptypes[],
         size_t ptype_count, GDI_Index index )
```

INOUT	array_of_labels	array of labels (array of handles)
IN	label_count	length of array_of_labels (non-negative integer)
INOUT	array_of_ptypes	array of property types (array of handles)
IN	ptype_count	length of array_of_ptypes (non-negative integer)
INOUT	index	index object (handle)

GDI_AddLabelsAndPropertyTypesToIndex is a function to associate multiple labels and property types with an explicit index. The labels that will be newly associated with `index` are located in `array_of_labels`, with the parameter `label_count` providing the number of labels that can be found in said array. Similarly the property types, that will be additionally associated with the index, are found in `array_of_ptypes` with `ptype_count` specifying the number of property type handles in said array. If one of the handles in either array is already associated with `index`, then that handle is ignored. If a label of `array_of_labels` or a property type of `array_of_ptypes` does not belong to the same graph database as `index` does, neither labels nor property types are added to the index and the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_AddLabelsAndPropertyTypesToIndex is a collective call and all parameters should be the same on all processes.

If `index` is only associated with labels, and only new labels will be added, meaning the array `array_of_ptypes` is empty, then all objects with at least one label present in `array_of_labels` will be added as entries to the index, unless they are already present in this index, because of other labels. However if `array_of_ptypes` is not empty, the amount of objects, that are indexed, can be reduced, because an object has to be part of both sets, the label set and the property type set, to be indexed.

Similarly if `index` is only associated with property types, and only new property types will be added, meaning `array_of_labels` is empty, then all objects with at least one property of any property type present in `array_of_ptypes` will be added as entries to the index, unless they are already present in this index, because of other property types. If however `array_of_labels` is not empty, it is possible, that the amount of indexed objects will be reduced, because an object has to be part of both sets to be indexed.

If `index` is associated with labels and property types, the amount of indexed objects can only grow, when labels and/or property types are additionally associated with the index.

It is possible to have indexes that are only associated with one of the sets, making them pure label indexes or pure property type indexes.

Objects with no label can be indexed by using the predefined label GDI_LABEL_NONE.

Explicit indexes will be updated implicitly during the commit of a transaction, if objects got labels or properties, associated with `index`, added or removed during the transaction.

The index can be used for a fast look-up of objects with labels and properties associated with the index.

> *Advice to users.* The function GDI_AddLabelsAndPropertyTypesToIndex allows to efficiently add multiple labels and/or multiple property types to an explicit index since the index is only recomputed after all labels and property types are added. (*End of advice to users.*)

```
int GDI_RemoveLabelsAndPropertyTypesFromIndex( GDI_Label array_of_labels[],
         size_t label_count, GDI_PropertyType array_of_ptypes[],
         size_t ptype_count, GDI_Index index )
```

INOUT	array_of_labels	array of labels (array of handles)
IN	label_count	length of array_of_labels (non-negative integer)
INOUT	array_of_ptypes	array of property types (array of handles)
IN	ptype_count	lengtof array_of_ptypes (non-negative integer)
INOUT	index	index object (handle)

GDI_RemoveLabelsAndPropertyTypesFromIndex is a function to remove the association of multiple labels and multiple property types with an explicit index they were previously registered to. The labels that will be removed from `index` are located in `array_of_labels`, with the parameter `label_count` providing the number of labels that can be found in said array. Similarly the property types, that will be removed from the index, are found in `array_of_ptypes` with `ptype_count` specifying the number of property type handles in said array. If a label from `array_of_labels` or a property type from `array_of_properties` was not associated with `index`, then that handle is ignored.

GDI_RemoveLabelsAndPropertyTypesFromIndex is a collective call and all parameters should be the same on all processes.

If `index` is a pure label index, then only labels can be removed. All objects, which have at least one label present in `array_of_labels`, will be removed from the index, unless they still have at least one additional label, that is still associated with `index`.

Similarly if `index` is a pure property type index, then only property types can be removed. All objects, which have at least one property of any property type found in `array_of_ptypes`, will be removed from the index, unless they still have at least one additional property of a property type, that is still associated with `index`.

If `index` is associated with both labels and property types before the call, and the call will remove all labels, but there will be still property types associated afterwards, it is possible that the number of indexed object will increase, since objects will not be filtered by labels anymore. In such a case, all objects, which contain at least one property of the property types still associated with `index`, but are currently not present in this index, because they didn't have any of the labels previously associated with `index`, will be added as entries to `index`.

Similarly the amount of indexed objects might also increase, if instead the call will remove all property types, but there will be still labels associated afterwards. In this case, all objects, which contain at least one label of the labels still associated with `index`, but are currently not present in this index, because they didn't have properties of any property type previously associated with `index`, will be added as entries to `index`.

If `index` is still associated with labels and property types after the call, the number of indexed objects can only decline, when labels and/or property types are removed from the index, because an object has to be part of both sets, the label set and the property type set, to be indexed.

If all labels and property types are removed from `index` during the call, then the index contains no entries after the return of the call.

If the predefined label GDI_LABEL_NONE is supplied in `array_of_labels`, objects with no label will be removed from the index according to the rules mentioned above.

> *Advice to users.* The function GDI_RemoveLabelsAndPropertyTypesFromIndex allows to efficiently remove multiple labels and/or multiple property types from an explicit index since the index is only recomputed after all labels and property types are removed. (*End of advice to users.*)

9.5 Querying Indexes

9.5.1 Implicit Indexes

```
int GDI_TranslateVertexID( bool* found_flag, GDI_Vertex_uid* internal_uid,
        GDI_Label label, const void* external_id, size_t size,
        GDI_Transaction transaction )
```

OUT	found_flag	flag to indicate whether the application-level ID was found (bool)
OUT	internal_uid	internal vertex UID (UID)
IN	label	label object (handle)
IN	external_id	initial address of application level ID (choice)
IN	size	size of application level ID (non-negative integer)
INOUT	transaction	transaction object (handle)

GDI_TranslateVertexID translates an application level ID of a vertex within the given label to an internal vertex UID, which can be used to access the vertex with a GDI_VertexHolder object. The size of the application level ID in Bytes is provided by the parameter `size`. An implicit index provided by the library is used to retrieve the internal vertex UID. If the application level ID is not found, then `false` is returned in `found_flag` and nothing is written to `internal_uid`. Otherwise `true` is returned in `found_flag` and the internal vertex UID is written to `internal_uid`. If `label` does not belong to the same graph as `transaction`, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_TranslateVertexID should only be called during a transaction and the retrieved internal vertex UID is only valid during the transaction from which GDI_TranslateVertexID is called. GDI_TranslateVertexID is usually the starting point for any kind of graph exploration, if not a new vertex is created or an explicit index is queried.

The predefined label GDI_LABEL_NONE can be used to retrieve the internal vertex UID of a vertex which has no labels. If there are multiple vertices, which do not have any labels and have the application level ID `external_id`, then `internal_uid` will contain the internal vertex UID of an arbitrary vertex with that ID and GDI_WARNING_NON_UNIQUE_ID is returned.

If the application does not use IDs for its vertices, it can't use GDI_TranslateVertexID and has to rely on querying explicit indexes instead.

> *Rationale.* It would be possible to provide the functionality of translating application level IDs to internal vertex UIDs with explicit indexes: One would create an explicit index for each label and add the predefined property type handle GDI_PROPERTY_TYPE_ID

to each of those indexes. However it is such a basic functionality for any query, that GDI_TranslateVertexID was introduced as a short hand and to allow the implementation additional optimizations, since the implicit index is fixed to those two associations, for such a heavily used use case.

No short hand for edges is provided, since edges typically do not have (U)IDs and indexes over edges introduce more overhead than indexes over vertices, since the number of edges is usually at least an order of magnitude higher than the number of vertices in a graph. However if the user needs such a functionality, he can use explicit indexes with the above mentioned scheme and a label that is exclusively used for edges. (*End of rationale.*)

9.5.2 Explicit Indexes

```
int GDI_GetVerticesOfIndex( GDI_Vertex_uid array_of_uids[], size_t count,
        size_t* resultcount, GDI_Constraint constraint, GDI_Index index,
        GDI_Transaction transaction )
```

OUT	array_of_uids	array of internal vertex UIDs (array of UIDs)
IN	count	length of array_of_uids (non-negative integer)
OUT	resultcount	number of retrieved UIDs (non-negative integer)
IN	constraint	constraint object (handle)
IN	index	explicit index object (handle)
INOUT	transaction	transaction object (handle)

GDI_GetVerticesOfIndex queries the index `index` and returns the internal vertex UIDs of all vertices that satisfy the conditions set by the constraint object that `constraint` points to. The internal vertex UIDs then can be used to access the vertices with GDI_VertexHolder objects. The internal vertex UIDs will be returned in the array `array_of_uids`, where `count` contains the maximum number of internal vertex UIDs that can be written to said array. On return `resultcount` contains the actual number of internal vertex UIDs written to `array_of_uids`. If the array is smaller than the available number of internal vertex UIDs, the array will be filled and the remaining internal vertex UIDs will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. If `constraint`, `index` and `transaction` do not belong to the same graph database, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_GetVerticesOfIndex should only be called during a transaction and all retrieved internal vertex UIDs are only valid during the transaction from which GDI_GetVerticesOfIndex is called.

GDI provides a way to filter the vertices found in `index` with the use of a constraint object. The conditions set in the constraint object `constraint` should only concern labels and entries of property types that are associated with `index`. It is possible to retrieve all vertices that are indexed in `index` by supplying the value GDI_CONSTRAINT_NULL as the parameter `constraint`. If a stale GDI_Constraint object is passed, GDI_ERROR_STALE is returned as error.

```
int GDI_GetLocalVerticesOfIndex( GDI_Vertex_uid array_of_uids[], size_t count,
        size_t* resultcount, GDI_Constraint constraint, GDI_Index index,
        GDI_Transaction transaction )
```

OUT	array_of_uids	array of internal vertex UIDs (array of UIDs)
IN	count	length of array_of_uids (non-negative integer)
OUT	resultcount	number of retrieved UIDs (non-negative integer)
IN	constraint	constraint object (handle)
IN	index	explicit index object (handle)
INOUT	transaction	transaction object (handle)

GDI_GetLocalVerticesOfIndex offers the same functionality as GDI_GetVerticesOfIndex, but instead returns only the internal vertex UIDs of vertices that are local to the calling process. The function is meant to be used in combination with collective read transactions as a scalable way to query explicit indexes. GDI_GetLocalVerticesOfIndex is a local call.

```
int GDI_GetEdgesOfIndex( GDI_Edge_uid array_of_uids[], size_t count,
        size_t* resultcount, GDI_Constraint constraint, GDI_Index index,
        GDI_Transaction transaction )
```

OUT	array_of_uids	array of internal edge UIDs (array of UIDs)
IN	count	length of array_of_uids (non-negative integer)
OUT	resultcount	number of retrieved UIDs (non-negative integer)
IN	constraint	constraint object (handle)
IN	index	explicit index object (handle)
INOUT	transaction	transaction object (handle)

GDI_GetEdgesOfIndex queries the index `index` and returns the internal edge UIDs of all edges that satisfy the conditions set by the constraint object that `constraint` points to. The internal edge UIDs then can be used to access the edges with GDI_EdgeHolder objects. The internal edge UIDs will be returned in the array `array_of_uids`, where `count` contains the maximum number of internal edge UIDs that can be written to said array. On return `resultcount` contains the actual number of internal edge UIDs written to `array_of_uids`. If the array is smaller than the available number of internal edge UIDs, the array will be filled and the remaining internal edge UIDs will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. If `constraint`, `index` and `transaction` do not belong to the same graph database, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_GetEdgesOfIndex should only be called during a transaction and all retrieved internal edge UIDs are only valid during the transaction from which GDI_GetEdgesOfIndex is called.

GDI provides a way to filter the edges found in `index` with the use of a constraint object. The conditions set in the constraint object `constraint` should only concern labels and entries of property types that are associated with `index`. It is possible to retrieve all edges that are indexed in `index` by supplying the value GDI_CONSTRAINT_NULL as the parameter `constraint`. If a stale GDI_Constraint object is passed, GDI_ERROR_STALE is returned as error.

```
int GDI_GetLocalEdgesOfIndex( GDI_Edge_uid array_of_uids[], size_t count,
        size_t* resultcount, GDI_Constraint constraint, GDI_Index index,
        GDI_Transaction transacti )
```

OUT	array_of_uids	array of internal edge UIDs (array of UIDs)
IN	count	length of array_of_uids (non-negative integer)
OUT	resultcount	number of retrieved UIDs (non-negative integer)
IN	constraint	constraint object (handle)
IN	index	explicit index object (handle)
INOUT	transaction	transaction object (handle)

GDI_GetLocalEdgesOfIndex offers the same functionality as GDI_GetEdgesOfIndex, but instead returns only the internal edge UIDs of edges that are local to the calling process. The function is meant to be used in combination with collective read transactions as a scalable way to query explicit indexes. GDI_GetLocalEdgesOfIndex is a local call.

9.6 Index Attributes

```
int GDI_GetAllIndexesOfDatabase( GDI_Index array_of_indexes[], size_t count,
        size_t* resultcount, GDI_Database graph_db )
```

OUT	array_of_indexes	array of indexes (array of handles)
IN	count	length of array_of_indexes (non-negative integer)
OUT	resultcount	number of retrieved indexes (non-negative integer)
IN	graph_db	graph database object (handle)

A user might not know what indexes are available in a certain graph database object.

GDI_GetAllIndexesOfDatabase will retrieve all indexes currently present in the given graph database. The user provides an array for index handles and the parameter `count`, which contains the maximum number of index handles that can be written to said array. On return `resultcount` contains the actual number of index handles written to `array_of_indexes`. If the array is smaller than the available number of index handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllIndexesOfDatabase is a local call.

```
int GDI_GetAllLabelsOfIndex( GDI_Label array_of_labels[], size_t count,
         size_t* resultcount, GDI_Index index )
```

OUT	array_of_labels	array of labels (array of handles)
IN	count	length of array_of_labels (non-negative integer)
OUT	resultcount	number of retrieved labels (non-negative integer)
IN	index	index object (handle)

GDI_GetAllLabelsOfIndex will retrieve all labels currently associated with the given index. The user provides an array for label handles `array_of_labels` and the parameter `count`, which contains the maximum number of label handles that can be written to said array. On return `resultcount` contains the actual number of label handles written to `array_of_labels`. If the array is smaller than the available number of label handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllLabelsOfIndex is a local call.

```
int GDI_GetAllPropertyTypesOfIndex( GDI_PropertyType array_of_ptypes[],
         size_t count, size_t* resultcount, GDI_Index index )
```

OUT	array_of_ptypes	array of property types (array of handles)
IN	count	length of array_of_ptypes (non-negative integer)
OUT	resultcount	number of retrieved property types (non-negative integer)
IN	index	index object (handle)

GDI_GetAllPropertyTypesOfIndex will retrieve all property types currently associated with the given index. The user provides an array for property type handles `array_of_ptypes` and the parameter `count`, which contains the maximum number of property type handles that can be written to said array. On return `resultcount` contains the actual number of property type handles written to `array_of_ptypes`. If the array is smaller than the available number of property handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. It is a local call.

```
int GDI_GetTypeOfIndex( int* itype, GDI_Index index )
```

OUT	itype	type of index (state)
IN	index	index object (handle)

GDI_GetTypeOfIndex returns the type of `index`. The returned state parameter `itype` can have exactly two values: GDI_INDEXTYPE_HASHTABLE and GDI_INDEXTYPE_BTREE. GDI_GetTypeOfIndex is a local call.

10 Basic Datatypes

Datatypes give context to property types by letting the graph database know the size of the elements of a property value and by enabling the graph database to run operations on the elements of property values. Together with the count parameter and the size limitations of properties, it enables the composition of fixed and variable datatypes.

GDI supports the following datatypes:

GDI datatype	C datatype
GDI_CHAR	char (treated as printable character)
GDI_BOOL	bool
GDI_INT8_T	int8_t
GDI_INT16_T	int16_t
GDI_INT32_T	int32_t
GDI_INT64_T	int64_t
GDI_UINT8_T	uint8_t
GDI_UINT16_T	uint16_t
GDI_UINT32_T	uint32_t
GDI_UINT64_T	uint64_t
GDI_FLOAT	float
GDI_DOUBLE	double
GDI_DECIMAL	GDI_Decimal
GDI_TIME	GDI_Time
GDI_DATE	GDI_Date
GDI_DATETIME	GDI_Datetime
GDI_BYTE	

Direct addresses instead of handles are used for all non-native datatypes to ensure that they can be used in the same way as native datatypes.

Advice to users. Compared to RDBMS, GDI does not support common datatypes of SQL like Timestamp and Year. However Timestamp can be represented in GDI by GDI_Datetime and Year by uint8_t. (*End of advice to users.*)

10.1 Character Datatype

To represent characters, GDI supports char. The type is implemented by the according C datatype, such that no functions are required for the character datatype. The required storage and value range is summarized in the following table:

GDI datatype	Storage (in Bytes)	Possible Values
GDI_CHAR	1	A character (ASCII Code)

Advice to users. The user is advised to use GDI_CHAR as the basic datatype to store UTF-8 strings, but has to keep in mind that the representation of a single UTF-8 character might use up to 4 Bytes (at the time of writing this document). (*End of advice to users.*)

10.2 Integer Numeric Datatypes

The integer numeric datatypes are categorized into signed integer types (int8_t, int16_t, int32_t and int64_t) which store negative and positive integers, and unsigned integer types (bool, uint8_t, uint16_t, uint32_t, uint64_t) which store only non-negative integers. The integer numeric datatypes are implemented by the according C datatypes, such that no functions are required for integer numeric datatypes. The required storage and value range is summarized in the following table:

GDI datatype	Storage (in Bytes)	Minimum Value	Maximum Value
GDI_INT8_T	1	-128	127
GDI_INT16_T	2	-32768	32767
GDI_INT32_T	4	-2147483648	2147483647
GDI_INT64_T	8	-2^{63}	$2^{63}-1$
GDI_BOOL	1 Bit (at most 1 Byte)	0 (false)	1 (true)
GDI_UINT8_T	1	0	255
GDI_UINT16_T	2	0	65535
GDI_UINT32_T	4	0	4294967295
GDI_UINT64_T	8	0	$2^{64}-1$

Rationale. GDI relies on the ISO C 99 standard and treats therefore a boolean value as an unsigned integer type which can have either the value 0 (false) or 1 (true). GDI allows to represent a boolean value with up to 8 Bits to conform to most programming languages in which the smallest addressable value is a Byte. (*End of rationale.*)

10.3 Floating Point Numeric Datatypes

GDI supports two floating point datatypes: float and double. The types use IEEE 754-1989 for representation. The floating point datatypes are implemented by the according C datatypes, such that no functions are required for floating point numeric datatypes. The required storage and value range is summarized in the following table:

GDI datatype	Storage (in Bytes)	Minimum Value	Maximum Value
GDI_FLOAT	4	-3.40282^{38}	3.40282^{38}
GDI_DOUBLE	8	-1.79769^{308}	1.79769^{308}

Advice to users. The floating point numeric datatypes do not provide exact representation. It is recommended to use GDI_Decimal if exact representations are required (for example for monetary values). (*End of advice to users.*)

10.4 Fixed Point Numeric Datatype

The datatype GDI_Decimal stores exact numeric data values. The datatype supports a maximum number of 65 digits in total and a maximum number of 30 digits after the decimal point. The notation $\{x\}^y$, where x denotes a set of symbols and y is a positive integer number, denotes that symbols of x can occur consecutively up to y times.

The required storage and value range is summarized in the following table:

GDI datatype	Storage (in Bytes)	Minimum Value	Maximum Value
GDI_DECIMAL	At most 67 Bytes	$-10^{65}-1$	$10^{65}-1$

Advice to implementors. A decimal number is stored in at most 67 Bytes. This restriction allows to store the decimal number as string (without terminating null character). (*End of advice to implementors.*)

```
int GDI_SetDecimal(const char* decimal_str, GDI_Decimal* decimal )
```

IN	decimal_str	character string representing the decimal value (string)
OUT	decimal	initial address to the decimal object returned by the call (choice)

GDI_SetDecimal sets the decimal for a GDI_Decimal object, specified by its initial address read from the parameter `decimal`, with the numeric value passed as a character string. The string has the format $\{-\}v\{w\}^{m-d-1}.\{w\}^d$ or $\{-\}0.\{w\}^d$, where negative numbers have a leading minus sign, w represents the digits from 0 to 9, v represents the digits from 1 to 9, m denotes the total number of digits (non-negative integer between 1 and 65), and d denotes the number of digits after the decimal point (non-negative integer between 0 and 30). GDI_SetDecimal is a local call.

Rationale. The decimal value is read from a character string to prevent conversion errors. (*End of rationale.*)

```
int GDI_GetDecimal( char* decimal_str, size_t length,
         size_t* resultlength, const GDI_Decimal* decimal )
```

OUT	decimal_str	character string representing the decimal value (string)
IN	length	maximum length of decimal_str (non-negative integer)
OUT	resultlength	length of the returned character string (non-negative integer)
IN	decimal	initial address to a decimal object (choice)

GDI_GetDecimal retrieves a decimal from an existing GDI_Decimal object and returns said value to an output string. The output string will conform to the format $\{-\}v\{w\}^{(m-d-1)}.\{w\}^d$, if $m - d > 0$. If $m - d \leq 0$, the output string will have the format $\{-\}0.\{w\}^d$. v represents the digits from 1 to 9, w the digits from 0 to 9, m is in the range between 1 and 65 and d is in the range between 0 and 30. Negative numbers have a leading minus sign. `decimal_str` should be allocated so that it can hold a resulting string of length GDI_MAX_DECIMAL_SIZE characters. The parameter `length` indicates the length in Bytes of the allocated string `decimal_str`. The parameter `resultlength` indicates the length (in Bytes) of the string actually written. A null terminator is additionally stored at `decimal_str[resultlength]`. The value of `resultlength` cannot be larger than GDI_MAX_DECIMAL_SIZE-1. If the size of the allocated string is smaller than the size of the decimal, then the string will be filled, such that a valid UTF-8 string is returned, and the remaining characters will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetDecimal is a local call.

10.5 Time Datatypes

GDI offers built-in datatypes to handle time (GDI_Time), date (GDI_Date) as well as date and time (GDI_Datetime). These datatypes do not support negative years and additionally GDI_Datetime can't express daylight saving time (DST).

Rationale. GDI provides no functions to parse a string to time, date or datetime object. Due to many different standards, it is the user's responsibility to parse the values of such strings. Time, date and datetime representations are provided since date and time allow a more compressed representation than datetime. In certain applications, this allows to reduce the memory usage. (*End of rationale.*)

The required storage and value range is summarized in the following table:

GDI datatype	Storage (in Bytes)	Minimum Value	Maximum Value
GDI_Time	At most 5 Bytes	00:00:00.000	23:59:59.999
GDI_Date	At most 4 Bytes	0000-01-01	65535-12-31
GDI_Datetime	At most 11 Bytes	-1200 0000-01-01 00:00:00.000	+1400 65535-12-31 23:59:59.999

```
int GDI_SetTime( uint8_t hour, uint8_t minute, uint8_t second,
        uint16_t fraction, GDI_Time* time )
```

IN	hour	numeric value for the hour (non-negative integer between 0 and 23)
IN	minute	numeric value for the minute (non-negative integer between 0 and 59)
IN	second	numeric value for the second (non-negative integer between 0 and 59)
IN	fraction	numeric value for the fraction of a second (non-negative integer between 0 and 999)
OUT	time	initial address to the time object returned by the call (choice)

GDI_SetTime sets the time for a GDI_Time object, specified by its initial address read from the parameter `time`, with the numeric values for hour, minute, second and fraction of a second. The `fraction` parameter allows for milliseconds precision (between 0 and 999). GDI_SetTime is a local call.

```
int GDI_GetTime( uint8_t* hour, uint8_t* minute, uint8_t* second,
        uint16_t* fraction, const GDI_Time* time )
```

OUT	hour	numeric value for the hour (non-negative integer between 0 and 23)
OUT	minute	numeric value for the minute (non-negative integer between 0 and 59)
OUT	second	numeric value for the second (non-negative integer between 0 and 59)
OUT	fraction	numeric value for the fraction of a second (non-negative integer between 0 and 999)
IN	time	initial address of a time object (choice)

GDI_GetTime retrieves a time from an existing GDI_Time object and returns the numeric values for hour, minute, second and fraction of a second. The `fraction` parameter has milliseconds precision (between 0 and 999). GDI_GetTime is a local call.

```
int GDI_SetDate( uint16_t year, uint8_t month, uint8_t day,
        GDI_Date* date )
```

IN	year	numeric value for the year (non-negative integer)
IN	month	numeric value for the month (positive integer between 1 and 12)
IN	day	numeric value for the day (positive integer between 1 and 31)
OUT	date	initial address to the date object returned by the call (choice)

GDI_SetDate sets the date for a GDI_Date object, specified by its initial address read from `date`, with the numeric values for day, month and year. GDI_SetDate is a local call.

```
int GDI_GetDate( uint16_t* year, uint8_t* month, uint8_t* day,
        const GDI_Date* date )
```

OUT	year	numeric value for the year (non-negative integer)
OUT	month	numeric value for the month (positive integer between 1 and 12)
OUT	day	numeric value for the day (positive integer between 1 and 31)
IN	date	initial address of a date object (choice)

GDI_GetDate retrieves a date from an existing GDI_Date object and returns the numeric values for day, month and year. GDI_GetDate is a local call.

```
int GDI_SetDatetime( uint16_t year, uint8_t month, uint8_t day,
         uint8_t hour, uint8_t minute, uint8_t second,
         uint16_t fraction, int16_t timezone,
         GDI_Datetime* datetime )
```

IN	year	numeric value for the year (non-negative integer)
IN	month	numeric value for the month (positive integer between 1 and 12)
IN	day	numeric value for the day (positive integer between 1 and 31)
IN	hour	numeric value for the hour (non-negative integer between 0 and 23)
IN	minute	numeric value for the minute (non-negative integer between 0 and 59)
IN	second	numeric value for the second (non-negative integer between 0 and 59)
IN	fraction	numeric value for the fraction of a second (non-negative integer between 0 and 999)
IN	timezone	UTC time offset (integer)
OUT	datetime	initial address to the datetime object returned by the call (choice)

GDI_SetDatetime sets the date and time of a GDI_Datetime object. The `fraction` parameter allows for milliseconds precision (between 0 and 999). The `timezone` parameter specifies the offset in the format `{+|-}hhmm` for the UTC time zones. It can range from `-1200` to `+1400`. GDI_SetDatetime is a local call.

```
int GDI_GetDatetime( uint16_t* year, uint8_t* month, uint8_t* day,
         uint8_t* hour, uint8_t* minute, uint8_t* second,
         uint16_t* fraction, int16_t* timezone,
         const GDI_Datetime* datetime )
```

OUT	year	numeric value for the year (non-negative integer)
OUT	month	numeric value for the month (positive integer between 1 and 12)
OUT	day	numeric value for the day (positive integer between 1 and 31)
OUT	hour	numeric value for the hour (non-negative integer between 0 and 23)
OUT	minute	numeric value for the minute (non-negative integer between 0 and 59)
OUT	second	numeric value for the second (non-negative integer between 0 and 59)
OUT	fraction	numeric value for the fraction of a second (non-negative integer between 0 and 999)
OUT	timezone	UTC time offset (integer)
IN	datetime	initial address of a datetime object (choice)

GDI_GetDatetime retrieves date and time from an existing GDI_Datetime object. The same constraints as for GDI_SetDatetime apply for the `fraction` and `timezone` parameters. GDI_GetDatetime is a local call.

10.6 Arbitrary Data

The datatype GDI_BYTE allows one to store the binary value of a Byte in memory unchanged. Together with the count parameter of properties, it is possible to store an arbitrary amount of binary data.

GDI datatype	Storage (in Bytes)	Explanation
GDI_BYTE	1	Binary data

10.7 Datatype Size

`int GDI_GetSizeOfDatatype( size_t* size, GDI_Datatype dtype )`

OUT	size	datatype size (non-negative integer)
IN	dtype	datatype object (handle)

GDI_GetSizeOfDatatype sets the value of `size` to the number of Bytes that the datatype `dtype` occupies. GDI_GetSizeOfDatatype is a local call.

10.8 Conversion

GDI allows to convert datatypes. But not all conversions are allowed and the conversion might result in loss of information. We define the following groups of GDI datatypes:

C integer	GDI_INT8_T, GDI_INT16_T, GDI_INT32_T, GDI_INT64_T, GDI_BOOL, GDI_UINT8_T, GDI_UINT16_T, GDI_UINT32_T, GDI_UINT64_T
Floating point	GDI_FLOAT, GDI_DOUBLE
Time	GDI_TIME, GDI_DATE, GDI_DATETIME

The following conversions are allowed:

Source datatype	Destination datatype
C integer	C integer, Floating point, GDI_DECIMAL
Floating point	C integer, Floating point, GDI_DECIMAL
GDI_DECIMAL	C integer, Floating point
GDI_TIME	GDI_DATETIME
GDI_DATE	GDI_DATETIME
GDI_DATETIME	GDI_TIME, GDI_DATE

If a datatype from the C integer group is converted to a datatype from the C integer or Floating point group, the C arithmetic rules apply. If a datatype from the C integer type is converted to the datatype GDI_DECIMAL, then the integer number is stored without loss of precision. If a datatype from the Floating point group is converted to a datatype from the C integer or Floating point group, the C arithmetic rules apply. If a datatype from the Floating point group is converted to the datatype GDI_DECIMAL, the number is first converted to a string with up to 65 digits in total and 30 digits after the decimal point and then stored as GDI_DECIMAL. This conversion might result in loss of precision. If the datatype GDI_DECIMAL is converted to a datatype from the C integer group, the value after the decimal point is cut off (round down). If the datatype GDI_DECIMAL is converted to a datatype from the Floating point group, the value is converted in best effort and might result in loss of precision. If the datatype GDI_TIME is converted to GDI_DATETIME, the date part is set to 0000-01-01 and the time zone is set to 0000. If the datetype GDI_DATE is converted to GDI_DATETIME, the time part is set to 00:00:00.000 and the time zone is set to 0000. If the datatype GDI_DATETIME is converted to GDI_TIME, the date part is cut off and if converted to GDI_DATE, the time part is cut off. In both cases, the timezone information of the GDI_DATETIME datatype is ignored.

Advice to users. If a datatype from the Floating point group is converted to the datatype GDI_DECIMAL and vice versa, it is advised to check if the conversion resulted in loss of precision. (*End of advice to users.*)

10.9 GDI Operations

GDI has the concept of operations, which are primarily used for comparison, to enable filtering of sets of vertices and edges by their properties or labels. Depending on the nature of the datatype, with whom a property type is associated, GDI allows the use of certain operations (GDI_Op).

The following operations are defined:

Name	Meaning
GDI_EQUAL	operand is equal to the value
GDI_NOTEQUAL	operand is not equal to the value
GDI_GREATER	operand is greater than the value
GDI_EQGREATER	operand is greater than the value or equal to the value
GDI_SMALLER	operand is smaller than the value
GDI_EQSMALLER	operand is smaller than the value or equal to the value

Not every datatype supports all operations. Using the GDI datatype groups specified in Section 10.8, the allowed combinations of GDI_Op and GDI_Datatype parameters are specified below. Further, GDI allows to compare labels using GDI_EQUAL and GDI_NOTEQUAL.

GDI_EQUAL, GDI_NOTEQUAL	C integer, Time, GDI_CHAR, GDI_DECIMAL, GDI_BYTE, Labels
GDI_GREATER, GDI_SMALLER	C integer, Floating point, Time, GDI_DECIMAL
GDI_EQGREATER, GDI_EQSMALLER	C integer, Time, GDI_DECIMAL

It is possible to compare datatypes that are in the same group for the C integer and Floating point groups. The C arithmetic rules apply for comparison within these two groups.

It is also possible to compare datatypes within the Time group, however certain limitations apply: When comparing GDI_TIME with GDI_DATETIME, only the time portion of GDI_DATETIME is significant. When comparing GDI_DATE with GDI_DATETIME, only the date portion of GDI_DATETIME is significant. It is not possible to compare GDI_DATE with GDI_TIME.

It also possible to compare datatypes that are from different groups: When comparing datatypes from the groups C integer and Floating point, then the C arithmetic rules apply.

GDI_Decimal can be compared to C integer and Floating point, where the C integer and Floating point are converted to GDI_Decimal. C integer are converted to GDI_DECIMAL without loss of precision. Floating point are converted to GDI_DECIMAL using highest precision (65 digits in total and 30 digits after the decimal point). Precision of the floating point value might be lost during conversion. If two GDI_Decimal values with different precision are compared, the values are compared using 65 digits in total and 30 digits after the decimal point.

In contrast, GDI_BYTE values should only be compared to other GDI_BYTE values.

For comparison of values that consist of more than one element, the number of elements has to be same for both operands. If the number of elements does not match, the comparison will return false. However, if it is the same, then each element will be compared with its respective counterpart of the other operand. Only if each of the element comparison returns true, the whole comparison will return true. Otherwise false will be returned for the whole comparison. When comparing operands of different datatypes, the rules as specified above apply.

11 Transactions

Transactions are a core concept of GDI. Similar to transactions in RDBMS, a transaction consists of a sequence of operations on the graph. A transaction must guarantee Atomicity, Consistency, Isolation and Durability (ACID). Atomicity ensures that the operations are treated as single unit and either all succeed or completely fail. Consistency ensures that before and after a transaction, the database is always in a consistent state. Isolation ensures that concurrent transactions behave as if they were run in some sequential order. Durability ensures that a committed transaction will remain even in the case of a system failure.

> *Rationale.* Atomicity, Consistency and Isolation are generally required by business database queries. Further, transactions must guarantee Durability such that changes remain. (*End of rationale.*)

> *Advice to implementors.* Locking algorithms like Two Phase Locking provide ACI. However, GDI poses no restriction to the algorithm used to ensure ACI. (*End of advice to implementors.*)

GDI differentiates between transaction-critical and transaction-non-critical errors. If a function returns a transaction-critical error, the transaction is guaranteed to fail. GDI does not offer functions to retry a transaction or to recover from a transaction-critical error: The user must start a new transaction.

> *Advice to users.* Transaction-critical error codes guarantee the transaction to fail. Therefore the user is advised to compare the return code of functions called in a transaction with GDI_ERROR_TRANSACTION_CRITICAL to abort as early as possible. (*End of advice to users.*)

There are three kind of transactions: (1) Single process transactions are transactions that a single process has started. This kind of transaction is meant for simple transactions which touch only a small set of the graph. Note that multiple processes might be involved. For example, the process that started the transaction might offload a write access to a different process. However such cases will be handled transparently by the library, so from the point of view of the process that started the transaction no other process is involved. (2) Collective read transactions are read-only transactions which involve all processes. The collective use of all processes allows to perform more complex queries which might involve all vertices or edges of the graph.

11.1 Single Process Transactions

```
int GDI_StartTransaction( GDI_Database graph_db, GDI_Transaction* transaction )
```

INOUT	graph_db	graph database object to query (handle)
OUT	transaction	transaction object returned by the call (handle)

GDI_StartTransaction will start a transaction and allocate all necessary internal data structures. The transaction is tied to a single graph database, provided by the parameter `graph_db`. Multiple processes can be in different transactions concurrently. Also, a single process can be in multiple transactions concurrently.

> *Rationale.* Running multiple transactions allows to share intermediate results of concurrent transactions. (*End of rationale.*)

```
int GDI_CloseTransaction( GDI_Transaction* transaction, int ctype )
```

INOUT	transaction	transaction object (handle)
IN	ctype	commit type (state)

GDI_CloseTransaction ends the transaction, which is referenced by the `transaction` parameter. `transaction` should be a handle returned by GDI_StartTransaction. The state parameter `ctype` is restricted to two values, so that the user can indicate, whether the transaction should commit (by passing the constant GDI_TRANSACTION_COMMIT) or abort (by passing the constant GDI_TRANSACTION_ABORT). The library will overwrite a requested commit, in case a transaction-critical error occurred during the transaction. If a commit is requested and succeeds, all changes applied during this transaction are committed to the graph database, so those changes will become globally visible. If commit is requested and fails, the function returns GDI_ERROR_TRANSACTION_COMMIT_FAIL and all changes will be rolled back, such that no changes made during the transaction will become globally visible. Similarly a requested abort will also roll back all changes, so that no changes made during the transaction will become globally visible. In all cases, temporary data structures like the GDI_VertexHolder objects (objects that handle accesses to vertices), which were created during the transaction, will be deallocated and are no longer accessible afterwards. The function deallocates the transaction object and sets `transaction` to GDI_TRANSACTION_NULL.

> *Advice to implementors.* In case a locking algorithm is internally used, all acquired locks will be released. If the transaction is read-only, there will be no changes to the database and only the read locks will be released (if locks are used). (*End of advice to implementors.*)

> *Advice to users.* The user is advised to check the return code of GDI_CloseTransaction to see if the transaction committed successfully or failed. If previously a function returned a transaction-critical error, trying to commit the transaction is guaranteed to fail, so it is recommended to abort the transaction instead, so that resources can be deallocated and (if required) a new transaction can be started. (*End of advice to users.*)

> *Rationale.* GDI does not provide an explicit abort function to conform to the interface of GDI_CloseCollectiveTransaction. (*End of rationale.*)

11.2 Collective Read Transactions

```
int GDI_StartCollectiveTransaction( GDI_Database graph_db,
        GDI_Transaction* transaction )
```

INOUT	graph_db	graph database object to query (handle)
OUT	transaction	transaction object returned by the call (handle)

GDI_StartCollectiveTransaction starts a transaction that is shared by all processes associated with the database `graph_db`. The transaction is tied to that single graph database. `graph_db` should be the same on all processes. It is a collective call and will synchronize all processes of that database. All transactions on that graph database must be finished before a process enters the GDI_StartCollectiveTransaction call and no other transactions on that graph database must be started while the collective read transaction is active. A call to GDI_StartCollectiveTransaction has a barrier semantic: a process returns from the call only after all other processes have entered their matching call. Collective read transactions must only involve read accesses and are meant to be used, when the whole graph is going to be queried. Usually such a transaction is used in conjunction with calls to GDI_GetLocalVerticesOfIndex and GDI_GetLocalEdgesOfIndex.

> *Rationale.* Collective read transactions are read-only to support complex business and graph analytic queries. Use cases for massive write transactions are rare and might induce write conflicts which can only be solved with complex algorithms, that impose high performance penalties. (*End of rationale.*)

```
int GDI_CloseCollectiveTransaction( GDI_Transaction* transaction, int ctype )
```

INOUT	transaction	transaction object (handle)
IN	ctype	commit type (state)

GDI_CloseCollectiveTransaction ends the collective read transaction, which is referenced by the `transaction` parameter. `transaction` should be the same on all processes and should be a handle, which was returned by GDI_StartCollectiveTransaction. The user can indicate by `ctype`, if the transaction should commit (by passing the constant GDI_TRANSACTION_COMMIT) or abort (by passing the constant GDI_TRANSACTION_ABORT). The state parameter `ctype` is restricted to those two values. The library will overwrite a requested commit, in case a transaction-critical error occurred during the transaction. The commit type is then exchanged with other processes, such that either all commit successfully or all abort. Note that the transaction aborts if a function on any process returned a transaction-critical error. If a commit was request by the calling process, but the transaction actually aborts, the error GDI_ERROR_TRANSACTION_COMMIT_FAIL is returned. There will be no changes to the database, as collective read transactions are read-only. GDI_CloseCollectiveTransaction is a collective call and will synchronize the processes of the database with a barrier semantic: a process returns from the call only after all other processes have entered their matching call. Other transactions on this graph database must only be started after GDI_CloseCollectiveTransaction returns. Temporary data structures like for example the GDI_VertexHolder objects, which were created during the transaction, will be deallocated and are no longer accessible afterwards. GDI_CloseCollectiveTransaction deallocates the transaction object and sets `transaction` to GDI_TRANSACTION_NULL.

> *Advice to implementors.* In case locks are used, all acquired read locks will be released. (*End of advice to implementors.*)

> *Rationale.* GDI does not offer an explicit collective abort function. An abort has inherently a single process notion. However when participating in a collective read transaction, the other processes still have to be informed about that decision, which would force the abort function to be collective. So the library would have to check periodically during the collective read transaction, whether a process has called abort. Additionally a deadlock situation could occur, in case another process has already entered his commit call. GDI combines both calls (the abort and the commit call) into one function to guarantee progress and avoid performance issues. (*End of rationale.*)

11.3 Transaction Attributes

```
int GDI_GetAllTransactionsOfDatabase( GDI_Transaction array_of_transactions[],
        size_t count, size_t* resultcount, GDI_Database graph_db )
```

OUT	array_of_transactions	array of transactions (array of handles)
IN	count	length of array_of_transactions (non-negative integer)
OUT	resultcount	number of retrieved transactions (non-negative integer)
IN	graph_db	graph database object (handle)

A user might not know what transactions are available in a certain graph database object. GDI_GetAllTransactionsOfDatabase will retrieve all transactions currently present in the given graph database. The user provides an array for transaction handles and the parameter `count`, which contains the maximum number of transaction handles that can be written to said array. On return the parameter `resultcount` contains the actual number of transaction handles written to `array_of_transactions`. If the array is smaller than the available number of transaction handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. The function GDI_GetAllTransactionsOfDatabase is a local call.

`int GDI_GetTypeOfTransaction( int* ttype, GDI_Transaction transaction )`

OUT	ttype	type of transaction (state)
IN	transaction	transaction object (handle)

The function GDI_GetTypeOfTransaction returns the type of `transaction`. The returned state parameter `ttype` can have exactly two values: GDI_SINGLE_PROCESS_TRANSACTION and GDI_COLLECTIVE_READ_TRANSACTION. GDI_GetTypeOfTransaction is a local call.

12 Constraints

GDI constraint objects enable topological queries to be aware of certain property and label conditions, enabling to filter datasets in an early stage. Also, constraint objects can be used to query indexes to access edges and vertices.

A GDI constraint object describes a logical formula that operates on labels and properties of a given object and, once evaluated, returns a boolean value. The basic unit of a constraint is a condition, which expresses a requirement that an object must fulfill. GDI distinguishes label and property conditions. Label conditions operate only on labels and express the need that an object must either have or not have a specified label. Property conditions operate only on properties and allow to compare values of properties (of specified objects) to given values using GDI operations. GDI groups a conjunction of conditions in subconstraints. The conjunction of conditions allows a short-circuit evaluation of the subconstraint to false when a condition evaluates to false. Multiple subconstraints can be grouped as a disjunction into a constraint object. The disjunction of subconstraints allows a short-circuit evaluation of the constraint to true when a subconstraint evaluates to true.

Overall, the construction of a constraint describes a logical formula in disjunctive normal form (DNF), which can also be visualized as a tree. An example that also highlights the similarity of DNF to the construction of constraints is shown in Figure 1.

$$(\text{property "name"} = \text{"John"} \wedge \text{label} = \text{"Employee"}) \vee (\text{label} = \text{"Supervisor"})$$

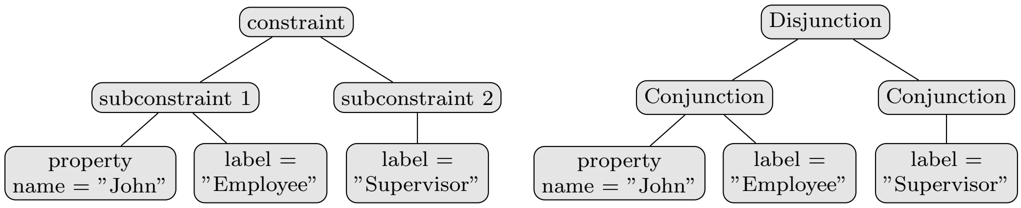

Figure 1: An example to visualize the similarity of GDI's constraint object (left) to a boolean logic in disjunctive normal form (right). The constraint evaluates to true on any object that has a property of type "name" with the value "John", and the label "Employee" or just has the label "Supervisor".

> *Rationale.* The close relationship of GDI constraints to DNF makes it easier to write and understand complex formulas. (*End of rationale.*)

> *Rationale.* An explicit approach in constructing constraint objects was chosen to avoid implementing complex string parsing at a low-level layer of the graph database stack. Such parsing should integrated into the general parsing of the graph query language. (*End of rationale.*)

GDI constraints rely on a static model, which means that it is not possible to update or remove arguments that are already added to conditions, subconstraints or constraints. Furthermore, if a subconstraint is added to a constraint, the original subconstraint can be deallocated afterwards, since the subconstraint is copied upon addition. Similarly when the list of subconstraints of a constraint object is retrieved, the returned objects are copies of the original objects.

> *Rationale.* The static model allows to define a clear behavior in special cases such as the removal of labels and property types from the graph database which are still associated with conditions in existing subconstraint and constraint objects. (*End of rationale.*)

If a condition operates on labels, or property types that have been removed from the database or property types that have been updated using GDI_UpdatePropertyType, then the condition is marked as stale. Associated subconstraints and (transitively) associated constraints also become stale and cannot be used anymore as argument for certain functions (e.g., to query for vertices).

Rationale. GDI offers no function to fix a constraint or subconstraint object that became stale to prevent unexpected behavior, when such an object is used, while it was implicitly changed in the meantime. (*End of rationale.*)

Advice to users. It is the user's responsibility to explicitly create a new constraint or subconstraint object, when the user wants to continue to use the functionality provided by a constraint or subconstraint object marked as stale. The user is still allowed to query stale constraint and subconstraint objects to get the list of conditions that are still valid. (*End of advice to users.*)

12.1 Creation and Destruction

`int GDI_CreateConstraint( GDI_Database graph_db, GDI_Constraint* constraint )`

INOUT	graph_db	graph database object (handle)
OUT	constraint	constraint object returned by the call (handle)

GDI_CreateConstraint allocates a new constraint object. The newly created object has no subconstraints. GDI_CreateConstraint is a local call.

`int GDI_FreeConstraint( GDI_Constraint* constraint )`

INOUT	constraint	constraint object (handle)

GDI_FreeConstraint deallocates the constraint object, and sets the argument `constraint` to GDI_CONSTRAINT_NULL. GDI_FreeConstraint is a local call.

```
int GDI_GetAllConstraintsOfDatabase( GDI_Constraint array_of_constraints[],
         size_t count, size_t* resultcount, GDI_Database graph_db )
```

OUT	array_of_constraints	array of constraints (array of handles)
IN	count	length of array_of_constraints (non-negative integer)
OUT	resultcount	number of retrieved constraints (non-negative integer)
IN	graph_db	graph database object (handle)

A user might not know what constraints are locally available in a certain graph database object. The function GDI_GetAllConstraintsOfDatabase will retrieve all constraints, that are locally associated with the given graph database `graph_db` at the time of the call. The user provides an array for constraint handles and the parameter `count`, which contains the maximum number of constraint handles that can be written to said array. The parameter `resultcount` contains the actual number of constraint handles written to `array_of_constraints`. If the array is smaller than the available number of constraint handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllConstraintsOfDatabase is a local call.

GDI_GetAllConstraintsOfDatabase does not change the staleness state of the returned constraints, so those constraints may or may not be stale.

`int GDI_IsConstraintStale( int* staleness, GDI_Constraint constraint )`

OUT	staleness	returned staleness state of constraint (state)
IN	constraint	constraint object (handle)

GDI_IsConstraintStale checks the state of the given constraint object. If `constraint` is not stale, GDI_FALSE is returned in argument `staleness`. If a stale constraint object is passed, GDI_TRUE is returned in `staleness`. The state parameter `staleness` is restricted to those two values. GDI_IsConstraintStale is a local call.

```
int GDI_CreateSubconstraint( GDI_Database graph_db,
         GDI_Subconstraint* subconstraint )
```

INOUT	graph_db	graph database object (handle)
OUT	subconstraint	subconstraint object returned by the call (handle)

GDI_CreateSubconstraint allocates a new subconstraint object. The newly created object has no conditions. GDI_CreateSubconstraint is a local call.

```
int GDI_FreeSubconstraint( GDI_Subconstraint* subconstraint )
```

INOUT	subconstraint	subconstraint object (handle)

GDI_FreeSubconstraint deallocates the constraint object and sets argument `subconstraint` to GDI_SUBCONSTRAINT_NULL. GDI_FreeSubconstraint is a local call.

```
int GDI_GetAllSubconstraintsOfDatabase(
         GDI_Subconstraint array_of_subconstraints[], size_t count,
         size_t* resultcount, GDI_Database graph_db )
```

OUT	array_of_subconstraints	array of subconstraints (array of handles)
IN	count	length of array_of_subconstraints (non-negative integer)
OUT	resultcount	number of retrieved subconstraints (non-negative integer)
IN	graph_db	graph database object (handle)

A user might not know what subconstraints are locally available in a certain graph database object. The function GDI_GetAllSubconstraintsOfDatabase will retrieve all subconstraints, that are locally associated with the given graph database `graph_db` at the time of the call. The user provides an array for subconstraint handles and the parameter `count`, which contains the maximum number of subconstraint handles that can be written to said array. `resultcount` contains the actual number of subconstraint handles written to `array_of_subconstraints`. If the array is smaller than the available number of subconstraint handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllSubconstraintsOfDatabase is a local call.

GDI_GetAllSubconstraintsOfDatabase does not change the staleness state of the returned subconstraints, so those subconstraints may or may not be stale.

```
int GDI_IsSubconstraintStale( int* staleness, GDI_Subconstraint subconstraint )
```

OUT	staleness	returned staleness state of subconstraint (state)
IN	subconstraint	subconstraint object (handle)

GDI_IsSubconstraintStale checks the state of the given subconstraint object. If argument `subconstraint` is not stale, GDI_FALSE is returned in argument `staleness`. If a stale subconstraint object is passed, GDI_TRUE is returned in `staleness`. The state parameter `staleness` is restricted to those two values. GDI_IsSubconstraintStale is a local call.

12.2 Label Conditions

```
int GDI_AddLabelConditionToSubconstraint( GDI_Label label, GDI_Op op,
        GDI_Subconstraint subconstraint )
```

IN	label	label object (handle)
IN	op	operation (op)
INOUT	subconstraint	subconstraint object (handle)

GDI_AddLabelConditionToSubconstraint adds a label condition to `subconstraint`. The label condition consists of the label handle `label` and the operation `op`. Label conditions restrict `op` to the values GDI_EQUAL and GDI_NOTEQUAL. If a different GDI operation is given, the error GDI_ERROR_OP_DATATYPE_MISMATCH is returned. The function creates a condition that, on evaluation, is true if and only if the object, on which the evaluation is performed, is associated with `label` when `op` is set to GDI_EQUAL and not associated with `label` when `op` is set to GDI_NOTEQUAL. Otherwise, the condition evaluates to false and due to conjunction of the conditions of a subconstraint, the whole subconstraint then evaluates to false. If `label` does not belong to the same graph database as `subconstraint` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. If a stale subconstraint object is passed, GDI_ERROR_STALE is returned as error. GDI_AddLabelConditionToSubconstraint is a local call.

> *Advice to users.* GDI allows to add label conditions with the same label, even the same label condition, multiple times to a subconstraint. However, this should be avoided since it might increase the time required for evaluation. (*End of advice to users.*)

```
int GDI_GetAllLabelConditionsFromSubconstraint( GDI_Label array_of_labels[],
        GDI_Op array_of_ops[], size_t count, size_t* resultcount,
        GDI_Subconstraint subconstraint )
```

OUT	array_of_labels	array of labels (array of handles)
OUT	array_of_ops	array of operations (array of ops)
IN	count	array length (non-negative integer)
OUT	resultcount	number of retrieved conditions (non-negative integer)
IN	subconstraint	subconstraint object (handle)

GDI_GetAllLabelConditionsFromSubconstraint will retrieve all label conditions, that are not stale, of the subconstraint object `subconstraint`. The user provides an array for label handles, an array for GDI operations and the parameter `count`, which contains the maximum number of label handles and GDI operations that can be written to the respective arrays. On return, `resultcount` contains the actual number of label handles and GDI operations written to `array_of_labels` and `array_of_ops`, respectively. The i-th entries in both arrays form together the i-th label condition. If no label conditions are present on the given subconstraint, `resultcount` will be set to value 0 and nothing will be written to `array_of_labels` and `array_of_ops`. If the same label condition is part of `subconstraint` more than once, then that label condition appears multiple times in the returned arrays. If the arrays are smaller than the available number of label conditions, the arrays will be filled and the remaining label conditions will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllLabelConditionsFromSubconstraint is a local call.

12.3 Property Conditions

```
int GDI_AddPropertyConditionToSubconstraint( GDI_PropertyType ptype,
        GDI_op op, void* value, size_t count, GDI_Subconstraint subconstraint )
```

IN	ptype	property type object (handle)
IN	op	operation (op)
IN	value	initial address to the value (choice)
IN	count	number of elements (non-negative integer)
INOUT	subconstraint	subconstraint object (handle)

GDI_AddPropertyConditionToSubconstraint adds a new property condition to the subconstraint object `subconstraint`. The property condition consists of the property type `ptype`, the operation `op` and the comparison value `value`. `count` elements of the datatype associated with `ptype` will be read from the address given by the parameter `value` and stored in the subconstraint, so that `value` can be freed by the caller after the call. The data pointed to by `value` has to match said associated datatype. The datatype associated with `ptype` has to be valid for `op`, otherwise the error GDI_ERROR_OP_DATATYPE_MISMATCH is returned. Any size limitations of the property type `ptype` will be enforced.

The function creates a condition that, on evaluation, is true if and only if the object, on which the evaluation is performed, has at least one property value of type `ptype` on which the operation specified by `op` evaluates to true when using said property value of the object as first parameter and `value` as second parameter. Otherwise, the condition evaluates to false and due to conjunction of the conditions of a subconstraint, the whole subconstraint then evaluates to false. If `ptype` does not belong to the same graph database as `subconstraint` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. If a stale subconstraint object is passed, GDI_ERROR_STALE is returned as error. GDI_AddPropertyConditionToSubconstraint is a local call.

> *Advice to users.* GDI allows to add the same property condition (the tuple consisting of `ptype`, `op`, `value`) multiple times to a subconstraint. However, this should be avoided since it might increase the time required for evaluation. (*End of advice to users.*)

```
int GDI_GetAllPropertyTypesOfSubconstraint( GDI_PropertyType array_of_ptypes[],
        size_t count, size_t* resultcount, GDI_Subconstraint subconstraint )
```

OUT	array_of_ptypes	array of property types (array of handles)
IN	count	length of array_of_ptypes (non-negative integer)
OUT	resultcount	number of retrieved property types (non-negative integer)
IN	subconstraint	subconstraint object (handle)

GDI_GetAllPropertyTypesOfSubconstraint will retrieve the property types of all property conditions, that are not stale, of the subconstraint object `subconstraint`. However, if a property type is associated with multiple property conditions, that property type will only be returned once. The user provides an array for property type handles `array_of_ptypes` and the parameter `count`, which contains the maximum number of property type handles that can be written to said array. On return `resultcount` contains the actual number of property type handles written to `array_of_ptypes`. If the array is smaller than the available number of property handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. GDI_GetAllPropertyTypesOfSubconstraint is a local call.

```
int GDI_GetPropertyConditionsOfSubconstraint( void* buf, size_t buf_count,
        size_t* buf_resultcount, size_t array_of_offsets[], GDI_Op array_of_ops[],
        size_t offset_count, size_t* offset_resultcount, GDI_PropertyType ptype,
        GDI_Subconstraint subconstraint )
```

OUT	buf	initial address of buffer (choice)
IN	buf_count	length of buf (non-negative integer)
OUT	buf_resultcount	number of retrieved elements in buf (non-negative integer)
OUT	array_of_offsets	array of buffer offsets (array of non-negative integers)
OUT	array_of_ops	array of operations (op)
IN	offset_count	length of array_of_offsets (non-negative integer)
OUT	offset_resultcount	number of retrieved offsets (non-negative integer)
IN	ptype	property type object (handle)
IN	subconstraint	subconstraint object (handle)

GDI_GetPropertyConditionsOfSubconstraint retrieves all property conditions of type `ptype` of the subconstraint object `subconstraint`. The operation for each property condition will be returned in `array_of_ops`. The values of the property conditions will be stored in the buffer `buf` with `buf_count` specifying the maximum number of elements of the datatype associated with `ptype` that will fit into `buf`. On return `buf_resultcount` contains the actual number of elements of the datatype associated with `ptype` that are written to `buf`. The offset of each property condition value will be returned in `array_of_offsets`. The offsets will be scified in number of elements of the datatype associated with the property type `ptype`. The parameter `offset_count` contains the maximum number of offsets, that can be written to `array_of_offsets`. On return, `offset_resultcount` contains the actual number of entries written to `array_of_offsets`. If `subconstraint` contains n property conditions of type `ptype`, then `offset_resultcount` will be set to $n+1$. The first n entries in `array_of_offsets` contain the offset where the respective property condition value in `buf` begins. The last entry of `array_of_offsets` contains the total number of elements written. This construction enables to determine the number of elements of the i-th property condition value in `buf` by calculating `array_of_offsets[`$i+1$`] - array_of_offsets[`i`]`.

> *Rationale.* The last entry of `array_of_offsets` and `buf_resultcount` both determine the total number of elements written to `buf`. The parameter `buf_resultcount` is required in the function interface to return the number of elements when a null pointer is given for `buf`, or `array_of_offsets`, or `array_of_ops` or 0 is provided for `offset_count`, or `buf_count` (Section 2.6.2). (*End of rationale.*)

In contrast to `array_of_offsets`, `array_of_ops` contains only n entries (which is equal to `offset_resultcount`-1) on return: one for each property condition. Additionally the length of `array_of_ops` can only be `offset_count`-1.

> *Rationale.* `array_of_offsets` and `array_of_ops` share both the `offset_count` and `offset_resultcount` parameter, to minimize the number of function parameter and keep the interface consistent with the function to retrieve the label conditions. (*End of rationale.*)

To summarize, the i-th property condition consists of `ptype`, the i-th entry of `array_of_ops` and the value in `buf` at the offset provided by the i-th entry in `array_of_offsets`. The number of elements of that property condition value is determined by calculating `array_of_offsets[`$i+1$`] - array_of_offsets[`i`]`.

If no property conditions of type `ptype` are present on the given subconstraint, the parameters `offset_resultcount` and `buf_resultcount` will be set to the value 0 and nothing will be written to `buf`, `array_of_offsets` and `array_of_ops`. If the same property condition is part of `subconstraint` more than once, then that property condition appears multiple times in the returned data. If the arrays are smaller than the available number of property conditions, the arrays will be filled and the remaining property conditions will be omitted. Similarly, if buffer `buf` is too small to hold all property condition values, the buffer will be filled and the remaining property condition values will be omitted. The error GDI_ERROR_TRUNCATE will be returned in both overflow cases. If `ptype` does not belong to the same graph database as `subconstraint` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. GDI_GetPropertyConditionsOfSubconstraint is a local call.

12.4 Constraint Handling

```
int GDI_AddSubconstraintToConstraint( GDI_Subconstraint subconstraint,
        GDI_Constraint constraint )
```

IN	subconstraint	subconstraint object (handle)
INOUT	constraint	constraint object (handle)

GDI_AddSubconstraintToConstraint adds a copy of `subconstraint` to `constraint`. The given subconstraint is treated as disjunction to the already given subconstraints (if there are any). If `subconstraint` does not belong to the same graph database as `constraint` does, the error GDI_ERROR_OBJECT_MISMATCH is returned. If a stale GDI_Constraint object or a stale GDI_Subconstraint object is passed, GDI_ERROR_STALE is returned as error. GDI_AddSubconstraintToConstraint is a local call.

> *Advice to users.* GDI allows to add the same subconstraint multiple times to a constraint. However, this should be avoided since it might increase the time required for evaluation. (*End of advice to users.*)

```
int GDI_GetAllSubconstraintsOfConstraint(
        GDI_Subconstraint array_of_subconstraints[], size_t count,
        size_t* resultcount, GDI_Constraint constraint )
```

OUT	array_of_subconstraints	array of subconstraints (array of handles)
IN	count	length of array_of_subconstraints (non-negative integer)
OUT	resultcount	number of retrieved subconstraints (non-negative integer)
IN	constraint	constraint object (handle)

GDI_GetAllSubconstraintsOfConstraint will retrieve a copy of all subconstraints associated with the constraint object `constraint`. The user provides an array for subconstraint handles and the parameter `count`, which contains the maximum number of subconstraint handles that can be written to said array. On return, `resultcount` contains the actual number of subconstraint handles written to `array_of_subconstraints`. If the array is smaller than the available number of subconstraint handles, the array will be filled and the remaining handles will be omitted. The error GDI_ERROR_TRUNCATE will be returned in such an overflow case. If the same subconstraint is associated multiple times to `constraint`, it appears accordingly multiple times in `array_of_subconstraints`. GDI_GetAllSubconstraintsOfConstraint is a local call.

> *Advice to users.* Since GDI returns a copy of the subconstraints, it is the callers responsibility to free the resources accordingly. (*End of advice to users.*)

13 Error Handling

GDI functions return *error codes*. Their details (e.g., enumeration, mapping) are fully decided by the implementation, so that as much information as possible is expressed within the error codes. However there is one exception: GDI_SUCCESS. The function GDI_GetErrorString can be used to determine the (implementation specific) error string associated with an error code.

The *error classes* form a subset of standard error codes. Therefore, the values defined for GDI error classes are valid GDI error codes and a GDI function may return an error class as error code. The function GDI_GetErrorClass maps any error code to the corresponding error class to allow for an easier interpretation of error codes inside the application. The tables below show the valid error classes.

GDI_SUCCESS	no error
GDI_ERROR_ASSERT	invalid assert argument
GDI_ERROR_BUFFER	invalid buffer pointer
GDI_ERROR_CONSTRAINT	invalid constraint argument
GDI_ERROR_COUNT	invalid count argument
GDI_ERROR_DATABASE	invalid database argument
GDI_ERROR_DATATYPE	invalid datatype argument
GDI_ERROR_DATE	invalid datatype argument of type GDI_Date
GDI_ERROR_DATETIME	invalid datatype argument of type GDI_Datetime
GDI_ERROR_DECIMAL	invalid datatype argument of type GDI_Decimal
GDI_ERROR_DELIMITER	invalid character provided as delimiter argument
GDI_ERROR_EDGE	invalid edge argument
GDI_ERROR_EDGE_ORIENTATION	invalid edge orientation condition
GDI_ERROR_ERROR_CODE	invalid error code argument
GDI_ERROR_INDEX	invalid index argument
GDI_ERROR_LABEL	invalid label argument
GDI_ERROR_OP	invalid operation argument
GDI_ERROR_OP_DATATYPE_MISMATCH	operation for a given datatype is not defined
GDI_ERROR_PROPERTY_TYPE	invalid property type argument
GDI_ERROR_SIZE	invalid size argument
GDI_ERROR_STALE	an argument is marked as stale
GDI_ERROR_STATE	invalid constant for a state argument
GDI_ERROR_SUBCONSTRAINT	invalid subconstraint argument
GDI_ERROR_TIME	invalid datatype argument of type GDI_Time
GDI_ERROR_TRANSACTION	invalid transaction argument
GDI_ERROR_UID	invalid UID argument
GDI_ERROR_VERTEX	invalid vertex argument
GDI_ERROR_ARGUMENT	invalid argument of some other kind
GDI_ERROR_OBJECT_MISMATCH	objects belong to different graph databases
GDI_ERROR_UNKNOWN	unknown error
GDI_ERROR_TRUNCATE	returned data is truncated
GDI_ERROR_TRANSACTION_COMMIT_FAIL	transaction was to be committed, but aborted instead
GDI_ERROR_READ_ONLY_TRANSACTION	a write action was requested during a read-only transaction
GDI_ERROR_TRANSACTION_CRITICAL	a transactional critical error has occurred

Table 2: Error classes (Part 1)

GDI_ERROR_CONVERSION	conversion of the two specified datatypes is not possible
GDI_ERROR_RANGE	one of the arguments is outside of its valid range
GDI_ERROR_NO_PROPERTY	no such property on the object exists
GDI_ERROR_PROPERTY_EXISTS	a property of the requested type with the same value already exists on the object
GDI_ERROR_PROPERTY_TYPE_EXISTS	a property of a single entity type is already present on the object
GDI_ERROR_READ_ONLY_PROPERTY_TYPE	the property type is read-only, and can only be implicitly changed by the library
GDI_ERROR_NON_UNIQUE_ID	an object with the same application level ID already exists for that label
GDI_WARNING_NON_UNIQUE_ID	multiple objects without labels have the same application level ID
GDI_ERROR_CONSISTENCY	the requested operation would generate an inconsistency
GDI_ERROR_OTHER	known error not in this list
GDI_WARNING_OTHER	a situation occured, which may require attention
GDI_ERROR_INTERN	internal GDI (implementation) error
GDI_ERROR_NO_MEMORY	memory is exhausted
GDI_ERROR_RESOURCE	a necessary resource could not be acquired
GDI_ERROR_EMPTY_NAME	name string is empty
GDI_ERROR_NAME_EXISTS	name string already exits for that object type
GDI_ERROR_NOT_SAME	collective argument(s) not identical on all processes, or collective functions called in a different order by different processes
GDI_ERROR_SIZE_LIMIT	count argument not within size limitation bounds
GDI_ERROR_WRONG_TYPE	type of object is not suited for the requested operation
GDI_ERROR_NO_SUCH_FILE	file does not exist
GDI_ERROR_FILE_EXISTS	file exists
GDI_ERROR_BAD_FILE	invalid file name (e.g., path name too long)
GDI_ERROR_ACCESS	permission denied
GDI_ERROR_NO_SPACE	not enough space
GDI_ERROR_QUOTA	quota exceeded
GDI_ERROR_OUTPUT	an error occured while processing the output
GDI_ERROR_READ_ONLY_FILE	read-only file or file system
GDI_ERROR_FILE_IN_USE	file operation could not be completed, as the file is currently open by some process
GDI_ERROR_FILE_FORMAT	format of the CSV file differs from the described layout
GDI_WARNING_NOT_ALL_DATA_LOADED	not all data was loaded into the database
GDI_ERROR_IO	other I/O error
GDI_ERROR_LASTCODE	last error code

Table 3: Error classes (Part 2)

Rationale. GDI differentiates between GDI_ERROR_UNKNOWN and GDI_ERROR_OTHER. GDI_GetErrorString is a function that may retrieve helpful information about GDI_ERROR_OTHER. (*End of rationale.*)

The error codes satisfy,

$$
\begin{aligned}
0 = \mathit{GDI\_SUCCESS} < \mathit{GDI\_WARNING\_...} < \mathit{GDI\_WARNING\_OTHER} \\
< \mathit{GDI\_ERROR\_...} < \mathit{GDI\_ERROR\_TRANSACTION\_CRITICAL} \\
< \mathit{GDI\_ERROR\_...} \leq \mathit{GDI\_ERROR\_LASTCODE}.
\end{aligned}
$$

All error codes smaller than GDI_ERROR_TRANSACTION_CRITICAL are considered non-critical to transactions, while all errors bigger than GDI_ERROR_TRANSACTION_CRITICAL (and of course the error class itself) are considered critical to transactions, meaning that such a transaction is forced to be aborted/rolled back, in case such an error occurs. Similarly all error codes smaller than GDI_WARNING_OTHER (including the error class itself) are considered warnings, which inform the caller that a situation arose, while executing successfully the requested operation, which may require attention. In contrast, all error codes bigger than GDI_WARNING_OTHER are real errors, meaning the requested operation was not executed successfully.

> *Rationale.* The definition of GDI_SUCCESS as 0 is done to be in line with the common C practice. The introduction of a known GDI_ERROR_LASTCODE allows for handy sanity checks regarding error codes.
>
> Using GDI_ERROR_TRANSACTION_CRITICAL as a threshold allows for a simple check whether a transaction needs to be aborted, in case an error occurs. (*End of rationale.*)

These error classes can be grouped into different categories regarding the state of the data after a function returned such an error.

1) preoperational errors: The erroneous function will not write to any of the output arguments, so the state of any output buffer is the same as before the function was called. If it was a creation call, no new opaque object is created. The state of any existing opaque object is the same as before the function was called. If a write change was requested on a read-only object, no changes to the object are applied. The state of the transaction, if the function was called inside of one, is also unchanged. The graph database remains unchanged.

Most errors in this error category can occur while the input parameters are being parsed. These are usually program errors. Additionally the error category covers errors that result from preconditions (for example the existence of a specific property) or postconditions (for example non-unique application level IDs within a label) not being meet. It also includes situations, where an external resource (like a file) can't be accessed or a resource is exhausted.

The following error classes belong in this category:

GDI_ERROR_ACCESS	GDI_ERROR_EDGE
GDI_ERROR_ARGUMENT	GDI_ERROR_EDGE_ORIENTATION
GDI_ERROR_ASSERT	GDI_ERROR_EMPTY_NAME
GDI_ERROR_BAD_FILE	GDI_ERROR_ERROR_CODE
GDI_ERROR_BUFFER	GDI_ERROR_FILE_EXISTS
GDI_ERROR_CONSISTENCY	GDI_ERROR_FILE_FORMAT
GDI_ERROR_CONSTRAINT	GDI_ERROR_FILE_IN_USE
GDI_ERROR_CONVERSION	GDI_ERROR_INDEX
GDI_ERROR_COUNT	GDI_ERROR_LABEL
GDI_ERROR_DATABASE	GDI_ERROR_NAME_EXISTS
GDI_ERROR_DATATYPE	GDI_ERROR_NO_MEMORY
GDI_ERROR_DATE	GDI_ERROR_NO_PROPERTY
GDI_ERROR_DATETIME	GDI_ERROR_NO_SPACE
GDI_ERROR_DECIMAL	GDI_ERROR_NO_SUCH_FILE
GDI_ERROR_DELIMITER	GDI_ERROR_NON_UNIQUE_ID

Table 4: Preoperational error classes (Part 1)

GDI_ERROR_NOT_SAME	GDI_ERROR_RESOURCE
GDI_ERROR_OBJECT_MISMATCH	GDI_ERROR_SIZE
GDI_ERROR_OP	GDI_ERROR_SIZE_LIMIT
GDI_ERROR_OP_DATATYPE_MISMATCH	GDI_ERROR_STALE
GDI_ERROR_PROPERTY_EXISTS	GDI_ERROR_STATE
GDI_ERROR_PROPERTY_TYPE	GDI_ERROR_SUBCONSTRAINT
GDI_ERROR_PROPERTY_TYPE_EXISTS	GDI_ERROR_TIME
GDI_ERROR_RANGE	GDI_ERROR_TRANSACTION
GDI_ERROR_READ_ONLY_FILE	GDI_ERROR_UID
GDI_ERROR_READ_ONLY_PROPERTY_TYPE	GDI_ERROR_VERTEX
GDI_ERROR_READ_ONLY_TRANSACTION	GDI_ERROR_WRONG_TYPE

Table 5: Preoperational error classes (Part 2)

2) output argument errors: The user supplied output space was not big enough. The function will fill output buffers or write to output files to the extent possible, but not all data will be written. The state of any existing opaque object is the same as before the function was called. The state of the transaction, if the function was called inside of one, is also unchanged. The graph database remains unchanged.

The following error classes belong in this category:

GDI_ERROR_QUOTA	GDI_ERROR_TRUNCATE
GDI_ERROR_OUTPUT	

Table 6: Output argument error classes

3) transaction-critical errors: The function returned a transaction-critical error. The state of any output arguments used with this function is undefined, so the user should not rely on them. The state of any temporary opaque objects associated with this transaction is undefined as well. While it will be still possible to access those objects, any results might not be meaningful. The associated transaction is marked as erroneous. The graph database remains unchanged.

The following error class belongs in this category:

GDI_ERROR_TRANSACTION_CRITICAL

Table 7: Transaction-critical error class

4) transaction commit failed: A transaction that was tried to commit, was actually aborted instead. All temporary GDI_VertexHolder and GDI_EdgeHolder objects associated with this transaction will be invalidated and are no longer accessible. The transaction object itself is also invalidated. The graph database remains unchanged.

The following error class belongs in this category:

GDI_ERROR_TRANSACTION_COMMIT_FAIL

Table 8: Transaction commit failed error class

5) undefined errors: The state of neither local objects nor the graph database itself is defined. The documentation of the implementation might provide more details in such a case.

The following error classes belong in this category:

GDI_ERROR_INTERN	GDI_ERROR_OTHER
GDI_ERROR_IO	GDI_ERROR_UNKNOWN

Table 9: Undefined error classes

6) warnings: The call returned successfully and its functionality was fulfilled according to its description. The database and its objects are consistent with the constraints set out in subsection 1.2. However a certain situation has arisen that the user might want to address.

The following error classes belong in this category:

GDI_WARNING_NON_UNIQUE_ID	GDI_WARNING_OTHER
GDI_WARNING_NOT_ALL_DATA_LOADED	

Table 10: Warning error classes

GDI_ERROR_LASTCODE does not belong to any of those categories. It is only provided to allow for sanity checks regarding the error codes/classes.

> *Advice to users.* If the function call is erroneous in more than one way, the implementation can choose which of those errors will be returned. (*End of advice to users.*)

> *Rationale.* By letting the implementation decide, which error code to return, when there is ambiguity, complexity of the description (prioritization of error classes, consideration of edge/corner cases) was avoided in the specification. The decision gives the implementation more freedom to prioritize its error codes (for example by their severity or the performance costs associated with checking their specific condition). (*End of rationale.*)

```
int GDI_GetErrorClass( int* errorclass, int errorcode )
```

OUT	errorclass	error class associated with errorcode (integer)
IN	errorcode	error code returned by a GDI function (integer)

The function GDI_GetErrorClass maps an error code to its corresponding error class. An error class maps onto itself. If the given error code is unknown, the function returns the error class GDI_ERROR_ERROR_CODE. GDI_GetErrorClass is a local call.

```
int GDI_GetErrorString( char* errorstring, size_t length, size_t* resultlength,
          int errorcode )
```

OUT	errorstring	text that corresponds to errorcode (string)
IN	length	maximum length of errorstring (non-negative integer)
OUT	resultlength	length of the returned error string (non-negative integer)
IN	errorcode	error code returned by a GDI function (integer)

GDI_GetErrorString returns the error string corresponding to an error code/class. `length` denotes the length of the allocated string `errorstring`. The argument `errorstring` should be allocated so that it can hold a buffer space of GDI_MAX_ERROR_STRING Bytes. `resultlength` contains on return the length of the returned string in Bytes. A null terminator is additionally stored at `errorstring[resultlength]`. The value of `resultlength` cannot be larger than GDI_MAX_ERROR_STRING-1. If the allocated string is smaller than the actual error string, the string will be filled, such that a valid UTF-8 string is returned, and the remaining characters will be omitted. The error class GDI_ERROR_TRUNCATE will be returned in such an overflow case. If any other error occurs, GDI_GetErrorString will return an empty string. If the given error code is unknown, the function returns the error class GDI_ERROR_ERROR_CODE. GDI_GetErrorString is a local call.

14 Execution Model: Remarks

GDI provides no interface to the user to orchestrate the processes that make up the graph database. Instead, it is the user's responsibility to provide further functionality to distribute and assign work to the processes in an efficient way. For completeness, this section provides two examples how GDI can be used in a more complex graph database environment. Note that the execution model is not limited to these two options. Further, (hardware) limitations, (software) design decisions and other requirements must be taken into account for highest performance.

14.1 Primary-Secondary

In the primary-secondary model, one distinguishes primary and secondary machines. On the secondary machines, GDI is installed and it handles all accesses (create, read, update, delete) on the graph data. Usually, a server application runs on these machines; it listens for incoming requests that are then executed using GDI function calls. The primary machine acts as a gateway for clients that query the database. The primary machine accepts query requests from a client, applies optimizations, determines a query plan and splits it into sub queries (if needed). The sub queries are passed to the secondary machines that execute the commands using GDI and then return the results to the primary one. The results are aggregated and passed to the client. It is the primary machine's responsibility to guarantee that collective GDI functions are executed on all secondary servers. This execution model also allows to fetch data from the graph database and apply graph analytic algorithms on the secondary machines (or on different compute nodes) for further data processing. Note that the secondary machines nevertheless require paths of communication among each other since they act as a group.

14.2 Distributed Model

In the distributed model, all compute nodes are equal and have the same capabilities (they could both execute the GDI queries and listen for the incoming client requests). All nodes typically have GDI installed to participate in the graph database. There are multiple options to run such a graph database.

In an MPI-like setting, the queries to run are generally known beforehand and can be implemented directly on the compute nodes. Orchestration is done using a technology that allows broadcast and aggregation (such as MPI). The big advantage is that the queries are known such that coordination and data exchange can be optimized accordingly. Similar to the primary-secondary model, the close orchestration allows to implement graph analytic algorithms using the compute nodes as workers.

In a similar setting, one assumes that a remote client issues database queries to the distributed system. The compute nodes require a server application that listens for requests. Offloading and orchestration might become more complex since no compute node has a global view. It is then the user's responsibility to ensure that collective functions are executed by all compute nodes to prevent deadlock situations.

15 Bulk Data Loading

GDI offers the functionality to bulk load vertex and edge data from files. Those files have to be in a format similar to CSV (comma-separated values). The file should be UTF-8 encoded.

GDI assumes the following CSV like format: each object (vertex/edge) occupies one line of the CSV file (one record in the CSV terminology). A newline is indicated either by "\n" (ASCII decimal value 10) or by "\r\n" (ASCII decimal sequence 13 10). Each line is divided into fields of data by a delimiter (a single ASCII character).

These fields of data can be application level ID(s) and properties. It is required that each line contains all fields, even if some of those fields are empty. This requirement allows GDI to see CSV files as a two-dimensional matrix, where a column contains the values of a certain property type. A row of this two-dimensional matrix identifies the properties of a certain object. A non-empty field of data contains a property value of the property type of that column. In addition to the common CSV specification, GDI allows for a second delimiter (a single ASCII character), so that it is possible to identify different elements of a property value. An empty field of data indicates that the object does not have that particular property.

Since the data in the files are given as strings, a conversion to the according data types is applied.

The datatype GDI_CHAR supports ASCII and UTF-8 characters (Section 10.1). If a property type with basic datatype GDI_CHAR is defined to have more than one element, no second delimiter should be used to read multiple characters from the file.

> *Rationale.* This allows to store the UTF-8 encoded character string without the use of the second delimiter. (*End of rationale.*)

GDI also allows to load binary data using the datatype GDI_BYTE. The binary data must be given as Base64 encoded string according to RFC 4648[2] If a property type with basic datatype GDI_BYTE is defined to have more than one element, no second delimiter should be used to read multiple Bytes from the file.

> *Rationale.* The binary data must be Base64 encoded to ensure that the file is always in valid UTF-8. Furthermore, it allows to store the whole Base64 encoded string without the use of the second delimiter. (*End of rationale.*)

Numeric datatypes (integer, float, double and GDI_DECIMAL) given as strings should not contain leading zeros and spaces (eg. between the negative sign and the number). The plus sign should be omitted. The boolean datatype is given by the string "1" or "true" for the value true, and "0" or "false" for the value false.

The datatype GDI_DATE is expected to have the format `yyyy-MM-dd`, where `yyyy` is a place holder for the year (non-negative integer between 0 and 9999), `MM` a place holder for the month (positive integer between 1 and 12), `dd` a place holder for the days (positive integer between 1 and 31).

The datatype GDI_TIME is expected to have the format `hh:mm:ss.SSS`, where `hh` is a place holder for the hours (non-negative integer between 0 and 23), `mm` is a place holder for the minutes (non-negative integer between 0 and 59), `ss` is a place holder for the seconds (non-negative integer between 0 and 59), `SSS` is a place holder for the fraction of seconds (non-negative integer between 0 and 999).

GDI_DATETIME is expected to have the format `{+|-}hhmm yyyy-MM-dd hh:mm:ss.SSS`, where `hhmm` is a place holder for the UTC time zones (ranges from -1200 to +1400).

If a property type with numeric, boolean, time, date, or datetime datatype has more than one element, a second delimiter must be used.

Since data might contain specified delimiters, delimiters must be escaped using a backslash "\" (ASCII decimal value 92) before the delimiter. Furthermore, the following escape sequences are used: A backslash must be escaped by prepending another backslash "\\" (ASCII decimal sequence 92 92), a newline is represented by "\n" (ASCII decimal sequence 92 110), a carriage is represented by "\r" (ASCII decimal sequence 92 114), and a tabulator is represented by "\t" (ASCII decimal sequence 92 116).

[2] https://tools.ietf.org/html/rfc4648

Due to the escape sequences, GDI allows all ASCII values for delimiters, except for the ASCII decimal values 10 ("\n"), 13 ("\r"), 92 ("\"), 110 ("n"), 114 ("r") and 116 ("t").

Escaping data can lead to complex situations. For example, assume a comma "," as field delimiter and a semicolon ";" as delimiter for elements of a property, then the string "123;456,your house" is split into two fields ("123;456" and "your house"), where the first property has two values ("123" and "456") and the second property has the value "your house". The string "123;456\,your house" is just one UTF-8 encoded field ("123;456,your house"), which has two values ("123" and "456,your house"). The string "123\;456\\,your house" is split into two UTF-8 encoded fields ("123;456\" and "your house"). The string "123\;456\\\,your house" is just one UTF-8 encoded field ("123;456\,your house").

15.1 Vertex Loading

```
int GDI_LoadVertexCSVFile( int assert, const char* file_path, int header,
        int stype, char field_delimiter, char element_delimiter,
        GDI_PropertyType array_of_ptypes[], size_t ptype_count,
        GDI_Label array_of_labels[], size_t label_count, GDI_Database graph_db )
```

IN	assert	program assertion (integer)
IN	file_path	character string that contains the path to the vertex file (string)
IN	header	header existance (state)
IN	stype	sort type (state)
IN	field_delimiter	single character to indicate field limits (character)
IN	element_delimiter	single character to indicate element limits (character)
IN	array_of_ptypes	array of property types (array of handles)
IN	ptype_count	length of array_of_ptypes (non-negative integer)
IN	array_of_labels	array of labels (array of handles)
IN	label_count	length of array_of_labels (non-negative integer)
INOUT	graph_db	graph database object (handle)

GDI_LoadVertexCSVFile loads vertices in bulk from the file specified by `file_path` into the graph database `graph_db`. The state parameter `header` is restricted to two values: GDI_TRUE, in case the file contains a header as the first line, or GDI_FALSE, in case the file starts with the first vertex in the first line. `field_delimiter` is a single ASCII character to indicate that a field of data has ended and the next field will begin afterwards. `element_delimiter` is a single ASCII character to differentiate elements within a single property value.

The CSV file should have the following order of columns: In the first column GDI expects the application level ID of the vertex and the following columns should contain the additional properties.

Vertex ID	Property 1	Property 2	...	Property P
1	John	Smith		23
2	Kim	Mould		42
...				
n	Kjetil	Peersen		1991

Table 11: File format for a vertex file. Due to readability the application level IDs are shown in a textual representation instead of a Base64 encoding.

The mapping of the fields of data to property types is done with the help of the array `array_of_ptypes`. The number of entries in `array_of_ptypes` is specified by `ptype_count` and should be one less than the number of columns in the CSV file. The order of entries in the array should match the order of columns in the CSV file, for example that the first entry in `array_of_ptypes` will specify the property type for the property values in the second column and so on.

The labels, that going to be assigned to each vertex, are given by `array_of_labels`. The parameter `label_count` specifies the number of entries in `array_of_labels`.

The state parameter `stype` indicates if the vertices are sorted by their application level ID. The parameter is restricted to the values GDI_NO_SORTING if the vertices are not sorted, GDI_ASC_SORTING if the vertices are sorted in ascending order, or GDI_DESC_SORTING if the vertices are sorted in descending order.

> *Rationale.* An implementation might want to search vertices in the given file. If the file is sorted, the implementation can search in the file for these vertices much faster by applying a binary search scheme. (*End of rationale.*)

If a property type from `array_of_ptypes` and/or a label from `array_of_labels` do not belong to the database `graph_db`, the error GDI_ERROR_OBJECT_MISMATCH is returned. If the file can't be found or opened, the error GDI_ERROR_NO_SUCH_FILE is returned. If the number of columns in the CSV file is not `ptype_count` + 1, no vertices are loaded and the error GDI_ERROR_FILE_FORMAT is returned. If a read property is not within the size limitations of the respective property type, that property is not added to the vertex, while the other data is added to the database and the function returns GDI_WARNING_NOT_ALL_DATA_LOADED.

GDI_LoadVertexCSVFile is a collective call and will synchronize all processes of that database. All transactions on that graph database must be finished before a process enters before a GDI_LoadVertexCSVFile call. The function call has a barrier semantic: a process returns from the call only after all other processes have entered their matching call.

The `assert` argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 15.3. A value of `assert` = `0` is always valid.

```
int GDI_LoadVertexPropertiesCSVFile( int assert, const char* file_path,
        int stype, char field_delimiter, char element_delimiter,
        GDI_PropertyType ptype, GDI_Label label, GDI_Database graph_db )
```

IN	assert	program assertion (integer)
IN	file_path	character string that contains the path to the vertex property file (string)
IN	header	header existance (state)
IN	stype	sort type (state)
IN	field_delimiter	single character to indicate field limits (character)
IN	element_delimiter	single character to indicate element limits (character)
IN	ptype	property type (handle)
IN	label	vertex label (handle)
INOUT	graph_db	graph database object (handle)

GDI_LoadVertexPropertiesCSVFile loads (multiple entity) properties of vertices in bulk from the file specified by `file_path` into the graph database `graph_db`. The state parameter `header` is restricted to two values: GDI_TRUE, in case the file contains a header as the first line, or GDI_FALSE, in case the file starts with the first vertex in the first line. `field_delimiter` is a single ASCII character to indicate that a field of data has ended and the next field will begin afterwards. `element_delimiter` is a single ASCII character to differentiate elements within a single property value.

The CSV file should have the following order of columns: In the first column GDI expects the application level ID of the vertex and the following column should contain a single property value. The layout is illustrated in Table 12.

The property type of the values in the second column is specified by `ptype`. `label` together with the application level ID from the first column allows the database to retrieve the vertex in question, so that the property or properties can be added to that vertex.

The state parameter `stype` indicates if the vertices are sorted by their application level ID. The parameter is restricted to the values GDI_NO_SORTING if the vertices are not sorted, GDI_ASC_SORTING if the vertices are sorted in ascending order, GDI_DESC_SORTING if the

Vertex ID	Property
23	English
23	French
42	English
1	Norwegian
...	
n	Chinese

Table 12: File format for a vertex property file. Due to readability the application level IDs are shown in a textual representation instead of a Base64 encoding.

vertices are sorted in descending order or GDI_GROUPED if the vertices are not sorted, but properties pertaining to the same vertex can be found in consecutive lines.

> *Rationale.* An implementation might want to search vertices in the given file. If the file is sorted, the implementation can search in the file for these vertices much faster by applying a binary search scheme. (*End of rationale.*)

If either the property type `ptype` or `label` do not belong to the database `graph_db`, the error GDI_ERROR_OBJECT_MISMATCH is returned. If the file can't be found or opened, the error GDI_ERROR_NO_SUCH_FILE is returned. If the format of the CSV file differs from the described two column layout, no properties are added and the error GDI_ERROR_FILE_FORMAT is returned. If a vertex with the given `label` and an application level ID read from the CSV file is not found inside the database, the lines with the respective application level ID are going to be ignored, while the rest of the data is added to the database. Similarly if a read property is not within the size limitations of the property type `ptype`, that property is not added to the vertex, while the other data is added to the database. If `ptype` is a single entity property type, but a property of that type is already present on the vertex in question (either because the property was already there before the bulk loading call or the vertex appears on multiple lines of the CSV file), that property is ignored, while the other data is added to the database. In all three cases the function returns GDI_WARNING_NOT_ALL_DATA_LOADED.

GDI_LoadVertexPropertiesCSVFile is a collective call.

The `assert` argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 15.3. A value of `assert` = `0` is always valid.

15.2 Edge Loading

```
int GDI_LoadEdgeCSVFile( int assert, const char* file_path, int header,
         int stype, int dtype, char field_delimiter, char element_delimiter,
         GDI_PropertyType array_of_ptypes[], size_t ptype_count,
         GDI_Label array_of_labels[], size_t label_count, GDI_Label origin_label,
         GDI_Label target_label, GDI_Database graph_db )
```

IN	assert	program assertion (integer)
IN	file_path	character string that contains the path to the edge file (string)
IN	header	header existence (state)
IN	stype	sort type (state)
IN	dtype	direction type (state)
IN	field_delimiter	single character to indicate field limits (character)
IN	element_delimiter	single character to indicate element limits (character)
IN	array_of_ptypes	array of property types (array of handles)
IN	ptype_count	length of array_of_ptypes (non-negative integer)
IN	array_of_labels	array of labels (array of handles)
IN	label_count	length of array_of_labels (non-negative integer)
IN	origin_label	label of the vertices in the first column (handle)
IN	target_label	label of the vertices in the second column (handle)
INOUT	graph_db	graph database object (handle)

GDI_LoadEdgeCSVFile loads edges in bulk from the file specified by `file_path` into the graph database `graph_db`. The state parameter `header` is restricted to two values: GDI_TRUE, in case the file contains a header as the first line, or GDI_FALSE, in case the file starts with the first edge in the first line. `field_delimiter` is a single ASCII character to indicate that a field of data has ended and the next field will begin afterwards. `element_delimiter` is a single ASCII character to differentiate elements within a single property value.

The CSV file should have the following order of columns: In the first column GDI expects the application level ID of the origin vertex, in the second column the application level ID of the target vertex, and the following columns should contain the properties of that edge.

Origin Vertex ID	Target Vertex ID	Property 1	Property 2	...	Property P
1	5	2019-10-17	red		103.22
1	7	1990-12-31			97.88
2	3	2021-02-09	black		95612.12
...					
n	m	2013-11-23	blue		2.33

Table 13: File format for an edge file. Due to readability the application level IDs are shown in a textual representation instead of a Base64 encoding.

The mapping of the fields of data to property types is done with the help of the array `array_of_ptypes`. The number of entries in `array_of_ptypes` is specified by `ptype_count` and should be two less than the number of columns in the CSV file. The order of entries in the array should match the order of columns in the CSV file, for example that the first entry in `array_of_ptypes` will specify the property type for the property values in the first property column (the third column overall) and so on.

The labels are given by `array_of_labs` and assigned to every edge. The parameter `label_count` specifies the number of entries in `array_of_labels`.

`origin_label` and `target_label` are provided to uniquely identify the incident vertices of an edge. All vertices in the first column are expected to have the label `origin_label`, and all vertices in the second column are expected to have the label `target_label`. If no label is necessary to uniquely identify the vertices, GDI_LABEL_NONE can be provided as the respective argument.

The state parameter `dtype` is restricted to two values and indicates whether the edge is directed (GDI_EDGE_DIRECTED) or undirected (GDI_EDGE_UNDIRECTED).

The state parameter `stype` indicates if the edges are sorted. The parameter is restricted to the values GDI_NO_SORTING if the edges are not sorted, GDI_ORIGIN_TARGET if the edges are sorted in ascending order first by the origin and then the target vertex, GDI_TARGET_ORIGIN if the edges are sorted in ascending order first by the target and then the origin vertex.

Rationale. An implementation might want to search edges in the given file. If the file is sorted, the implementation can search in the file for these edges much faster by applying a binary search scheme. (*End of rationale.*)

If a property type from `array_of_ptypes` and/or a label from `array_of_labels` do not belong to the database `graph_db`, the error GDI_ERROR_OBJECT_MISMATCH is returned. If the file can't be found or opened, the error GDI_ERROR_NO_SUCH_FILE is returned. If the number of columns in the CSV file is not `ptype_count` + 2, no edges are loaded into the database and the error GDI_ERROR_FILE_FORMAT is returned. If a vertex, be it either a origin or a target vertex, with the respective label and the application level ID read from the CSV file is not found inside the database, the lines with that vertex are going to be ignored, while the rest of the data is added to the database and GDI_WARNING_NOT_ALL_DATA_LOADED is returned. Similarly if a read property is not within the size limitations of the respective property type, that property is not added to the vertex, while the other data is added to the database and the function returns GDI_WARNING_NOT_ALL_DATA_LOADED as well.

GDI_LoadEdgeCSVFile is a collective call and will synchronize all processes of that database. All transactions on that graph database must be finished before a process enters before a GDI_LoadEdgeCSVFile call. The function call has a barrier semantic: a process returns from the call only after all other processes have entered their matching call.

The `assert` argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 15.3. A value of `assert` = `0` is always valid.

15.3 Assertions

GDI provides an `assert` argument to several calls in the bulk loading chapter, namely GDI_LoadVertexCSVFile, GDI_LoadVertexPropertiesCSVFile, and GDI_LoadEdgeCSVFile. These assertions on the circumstances of a call can allow for performance optimizations in the implementation. If accurate information are given in the `assert` argument, the semantics of the program are not changed. However, it is invalid to provide inaccurate information. It is always possible to use `assert` = 0 to signal a general situation without any guarantees.

Advice to users. Implementations are not forced to take into account the `assert` information, so users should refer to the implementation's documentation to see which `assert` values are actually viable for their specific system. However, a user program, that always specifies accurate assertions, is portable, and optimizations are enabled, if available. (*End of advice to users.*)

Advice to implementors. It is possible for an implementation to disregard any `assert` information. However, implementors are encouraged to document viable `assert` values, so that users of their implementation can take advantage of them. (*End of advice to implementors.*)

An `assert` is a bitwise OR combination of a non-negative number of the following integer constants: . The viable `assert` arguments for each call are listed below.

GDI_LoadVertexCSVFile:

GDI_LoadVertexPropertiesCSVFile:

GDI_LoadEdgeCSVFile:

GDI Constant and Predefined Handle Index

This index lists predefined GDI constants and handles.

GDI Function Index

The underlined page numbers refer to the function definitions.

References

[1] Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. 2014. The Linked Data Benchmark Council: A Graph and RDF Industry Benchmarking Effort. *ACM SIGMOD Record* 43, 1 (March 2014), 27–31. https://doi.org/10.1145/2627692.2627697

[2] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. https://doi.org/10.48550/ARXIV.1910.09017 arXiv:1910.09017 [cs.DB]

[3] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. 1994. The MPI Message Passing Interface Standard. In *Programming Environments for Massively Parallel Distributed Systems*, Karsten M. Decker and René M. Rehmann (Eds.). Birkhäuser Basel, Basel, 213–218.

[4] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2723372.2742786