MySQL® NDB API DEVELOPERS' GUIDE

VERSION 2.0 (2006-10-18)

MySQL® NDB API DEVELOPERS' GUIDE: VERSION 2.0 (2006-10-18)

MySQL® NDB API DEVELOPERS' GUIDE

(revision: 3666)

Provides information for developers wishing to use the low-level C/C++-language NDB API for the MySQL® NDBCLUSTER storage engine. Includes concepts, terminology, programming class and structure references, practical examples, common problems, and tips for using the NDB API in applications.

The NDB and MGM APIs as presented in this guide are current for MySQL 5.1 — note that there have been significant changes in the API as implemented in previous MySQL Cluster versions. The definitions of NDB API classes and MGM API functions reflect the state of the MySQL main development tree as of MySQL 5.1.12 and later.

Please email <docs@mysql.com> for more information or if you are interested in doing a translation.

Table of Contents

Chapter 1. OVERVIEW & CONCEPTS

This chapter provides a general overview of essential MySQL Cluster, NDB API, and MGM API concepts, terminology, and programming constructs.

1.1. Introduction

This section introduces the NDB Transaction and Scanning APIs as well as the NDB Management (MGM) API for use in building applications to run on MySQL Cluster. It also discusses the general theory and principles involved in developing such applications.

1.1.1. The NDB API

The *NDB API* is an object-oriented application programming interface for MySQL Cluster that implements indexes, scans, transactions, and event handling. NDB transactions are ACID-compliant in that they provide a means to group together operations in such a way that they succeed (commit) or fail as a unit (rollback). It is also possible to perform operations in a "no-commit" or deferred mode, to be committed at a later time.

NDB scans are conceptually rather similar to the SQL cursors implemented in MySQL 5.0 and other common enterprise-level database management systems. These allow for high-speed row processing for record retrieval purposes. (MySQL Cluster naturally supports set processing just as does MySQL in its non-Cluster distributions. This can be accomplished via the usual MySQL APIs discussed in the MySQL Manual and elsewhere.) The NDB API supports both table scans and row scans; the latter can be performed using either unique or ordered indexes. Event detection and handling is discussed in Section 3.5, “The `NdbEventOperation` Class”, as well as Section 6.6, “NDB API Event Handling Example”.

In addition, the NDB API provides object-oriented error-handling facilities in order to provide a means of recovering gracefully from failed operations and other problems. See Section 6.3, “Handling Errors and Retrying Transactions”, for a detailed example.

The NDB API provides a number of classes implementing the functionality described above. The most important of these include the `Ndb`, `Ndb_cluster_connection`, `NdbTransaction`, and `NdbOperation` classes. These model (respectively) database connections, cluster connections, transactions, and operations. These classes and their subclasses are listed in Chapter 3, *NDB API CLASSES*. Error conditions in the NDB API are handled using `NdbError`, a structure which is described in Section 4.1, “The `NdbError` Structure”.

1.1.2. The MGM API

The *MySQL Cluster Management API*, also known as the *MGM API*, is a C-language programming interface intended to provide administrative services for the cluster. These include starting and stopping Cluster nodes, handling Cluster logging, backups, and restoration from backups, as well as various other management tasks. A conceptual overview of MGM and its uses can be found in Chapter 5, *THE MGM API*.

The MGM API's principal structures model the states of individual modes (`ndb_mgm_node_state`), the state of the Cluster as a whole (`ndb_mgm_cluster_state`), and management server response messages (`ndb_mgm_reply`). See Section 5.4, “MGM Structures”, for detailed descirptions of these.

1.2. Terminology

Provides a glossary of terms which are unique to the NDB and MGM APIs, or have a specialised mean-

ing when applied therein.

The following terms are useful to an understanding of MySQL Cluster, the NDB API, or have a specialised meaning when used in one of these contexts. In addition, you may find the MySQL Manual's MySQL Cluster Glossary [http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-glossary.html] to be useful as well.

- **Backup**: A complete copy of all cluster data, transactions and logs, saved to disk.

- **Restore**: Returning the cluster to a previous state as stored in a backup.

- **Checkpoint**: Generally speaking, when data is saved to disk, it is said that a checkpoint has been reached. When working with the `NDB` storage engine, there are two sorts of checkpoints which work together in order to ensure that a consistent view of the cluster's data is maintained:

  - **Local Checkpoint (LCP)**: This is a checkpoint that is specific to a single node; however, LCPs take place for all nodes in the cluster more or less concurrently. An LCP involves saving all of a node's data to disk, and so usually occurs every few minutes, depending upon the amount of data stored by the node.

    More detailed information about LCPs and their behaviour can be found in the MySQL Manual, in the sections Defining Data Nodes [http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-ndbd-definition.html], and Configuring Parameters for Local Checkpoints [http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-config-lcp-params.html].

  - **Global Checkpoint (GCP)**: A GCP occurs every few seconds, when transactions for all nodes are synchronised and the REDO log is flushed to disk.

    A related term is *GCI*, which stands for "Global Checkpoint ID". This marks the point in the REDO log where a GCP took place.

- **Node**: A component of MySQL Cluster. 3 node types are supported:

  - *Management (MGM) node*: This is an instance of `ndb_mgmd`, the cluster management server daemon.

  - *Data node* (sometimes also referred to as a "storage nodes", although this usage is now discouraged): This is an instance of `ndbd`, and stores cluster data.

  - *API node*: This is an application that accesses cluster data. *SQL node* refers to a `mysqld` process that is connected to the cluster as an API node.

  For more information about these node types, please refer to Section 1.3.3, "Review of MySQL Cluster Concepts", or to Process Management in MySQL Cluster [http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-process-management.html], in the MySQL Manual.

- **Node Failure**: MySQL Cluster is not solely dependent upon the functioning of any single node making up the cluster, which can continue to run even when one node fails.

- **Node Restart**: The process of restarting a cluster node which has stopped on its own or been

stopped deliberately. This can be done for several different reasons, including the following:

- Restarting a node which has shut down on its own (when this has occurred, it is known as *forced shutdown* or *node failure*; the other cases dicussed here involve manually shutting down the node and restarting it)
- To update the node's configuration
- As part of a software or hardware upgrade
- In order to defragment the node's `DataMemory`

- **Initial Node Restart**: The process of starting a cluster node with its filesystem removed. This is sometimes used in the course of software upgrades and in other special circumstances.

- **System Crash** (or **System Failure**): This can occur when so many cluster nodes have failed that the cluster's state can no longer be guaranteed.

- **System Restart**: The process of restarting the cluster and reinitialising its state from disk logs and checkpoints. This is required after either a planned or an unplanned shutdown of the cluster.

- **Fragment**: Contains a portion of a database table; in other words, in the `NDB` storage engine, a table is broken up into and stored as a number of subsets, usually referred to as fragments. A fragment is sometimes also called a *partition*.

- **Replica**: Under the `NDB` storage engine, each table fragment has number of replicas in order to provide redundancy.

- **Transporter**: A protocol providing data transfer across a network. The NDB API supports 4 different types of transporter connections: TCP/IP (local), TCP/IP (remote), SCI, and SHM. TCP/IP is, of course, the familiar network protocol that underlies HTTP, FTP, and so forth, on the Internet. SCI (Scalable Coherent Interface) is a high-speed protocol used in building multiprocessor systems and parallel-processing applications. SHM stands for Unix-style shared memory segments. For an informal introduction to SCI, see this essay [http://www.dolphinics.com/corporate/scitech.html] at dolphinics.com.

- `NDB`: This originally stood for “Network Database”. It now refers to the storage engine used by MySQL AB to enable its MySQL Cluster distributed database.

- **ACC**: Access Manager. Handles hash indexes of primary keys providing speedy access to the records.

- **TUP**: Tuple Manager. This handles storage of tuples (records) and contains the filtering engine used to filter out records and attributes when performing reads and/or updates.

- **TC**: Transaction Coordinator. Handles co-ordination of transactions and timeouts; serves as the interface to the NDB API for indexes and scan operations.

1.3. The NDB Cluster Transaction and Scanning API

This section defines and discusses the high-level architecture of the NDB API, and introduces the NDB classes which are of greatest use and interest to the developer. It also covers most important NDB API concepts, including a review of MySQL Cluster Concepts.

1.3.1. Core NDB API Classes

The NDB API is a MySQL Cluster application interface that implements transactions. It consists of the following fundamental classes:

- `Ndb_cluster_connection` represents a connection to a cluster.

  See Section 3.2, "The `Ndb_cluster_connection` Class".

- `Ndb` is the main class, and represents a connection to a database.

  See Section 3.1, "The `Ndb` Class".

- `NdbDictionary` provides meta-information about tables and attributes.

  See Section 3.4, "The `NdbDictionary` Class".

- `NdbTransaction` represents a transaction.

  See Section 3.9, "The `NdbTransaction` Class".

- `NdbOperation` represents an operation using a primary key.

  See Section 3.6, "The `NdbOperation` Class".

- `NdbScanOperation` represents an operation performing a full table scan.

  See Section 3.6.4, "The `NdbScanOperation` Class".

- `NdbIndexOperation` represents an operation using a unique hash index.

  See Section 3.6.3, "The `NdbIndexOperation` Class".

- `NdbIndexScanOperation` represents an operation performing a scan using an ordered index.

  See Section 3.6.4.3, "The `NdbIndexScanOperation` Class".

- `NdbRecAttr` represents an attribute value.

  See Section 3.7, "The `NdbRecAttr` Class".

In addition, the NDB API defines an `NdbError` structure, which contains the specification for an error.

It is also possible to receive events triggered when data in the database is changed. This is accomplished through the `NdbEventOperation` class.

> **Important**
>
> The `NDB` event notification API is not supported prior to MySQL 5.1. (Bug#19719 [http://bugs.mysql.com/19719])

For more information about these classes as well as some additional auxiliary classes not listed here, see Chapter 3, *NDB API CLASSES*.

1.3.2. Application Program Basics

The main structure of an application program is as follows:

1. Connect to a cluster using the `Ndb_cluster_connection` object.
2. Initiate a database connection by constructing and initialising one or more `Ndb` objects.
3. Identify the tables, columns, and indexes on which you wish to operate, using `NdbDictionary` and one or more of its subclasses.
4. Define and execute transactions using the `NdbTransaction` class.
5. Delete `Ndb` objects.
6. Terminate the connection to the cluster (terminate an instance of `Ndb_cluster_connection`).

1.3.2.1. Using Transactions

The procedure for using transactions is as follows:

1. Start a transaction (instantiate an `NdbTransaction` object).
2. Add and define operations associated with the transaction using instances of one or more of the `NdbOperation`, `NdbScanOperation`, `NdbIndexOperation`, and `NdbIndexScanOperation` classes.
3. Execute the transaction (call `NdbTransaction::execute()`).
4. The operation can be of two different types — `Commit` or `NoCommit`:

   - If the operation is of type `NoCommit`, then the application program requests that the operation portion of a transaction be executed, but without actually committing the transaction. Following the execution of a `NoCommit` operation, the program can continue to define additional transaction operations for later execution.

     `NoCommit` operations can also be rolled back by the application.

   - If the operation is of type `Commit`, then the transaction is immediately committed. The transaction must be closed after it has been committed (even if the commit fails), and no further operations can be added to or defined for this transaction.

   See Section 3.9.1.3, "The `NdbTransaction::ExecType` Type".

1.3.2.2. Synchronous Transactions

Synchronous transactions are defined and executed as follows:

1. Begin (create) the transaction, which is referenced by an `NdbTransaction` object typically created using `Ndb::startTransaction()`. At this point, the transaction is merely being defined; it is not yet sent to the NDB kernel.

2. Define operations and add them to the transaction, using one or more of the following:

   - `NdbTransaction::getNdbOperation()`
   - `NdbTransaction::getNdbScanOperation()`
   - `NdbTransaction::getNdbIndexOperation()`
   - `NdbTransaction::getNdbIndexScanOperation()`

   along with the appropriate methods of the respective`NdbOperation` class (or possibly one or more of its subclasses). Note that, at this point, the transaction has still not yet been sent to the NDB kernel.

3. Execute the transaction, using the `NdbTransaction::execute()` method.

4. Close the transaction by calling `Ndb::closeTransaction()`.

For an example of this process, see Section 6.1, "Using Synchronous Transactions".

To execute several synchronous transactions in parallel, you can either use multiple `Ndb` objects in several threads, or start multiple application programs.

1.3.2.3. Operations

An `NdbTransaction` consists of a list of operations, each of which is represented by an instance of `NdbOperation`, `NdbScanOperation`, `NdbIndexOperation`, or `NdbIndexScanOperation` (that is, of `NdbOperation` or one of its child classes).

Some general information about cluster access operation types can be found in Understanding the Impact of Cluster Interconnects [http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-performance-figures.html], in the MySQL Manual.

1.3.2.3.1. Single-row operations

After the operation is created using NdbTransaction::getNdbOperation() or NdbTransaction::getNdbIndexOperation(), it is defined in the following three steps:

1. Specify the standard operation type using `NdbOperation::readTuple()`.
2. Specify search conditions using `NdbOperation::equal()`.
3. Specify attribute actions using `NdbOperation::getValue()`.

Here are two brief examples illustrating this process. For the sake of brevity, we omit error handling.

This first example uses an `NdbOperation`:

```
// 1. Retrieve table object
myTable= myDict->getTable("MYTABLENAME");

// 2. Create an NdbOperation on this table
myOperation= myTransaction->getNdbOperation(myTable);

// 3. Define the operation's type and lock mode
myOperation->readTuple(NdbOperation::LM_Read);

// 4. Specify search conditions
```

```
myOperation->equal("ATTR1", i);

// 5. Perform attribute retrieval
myRecAttr= myOperation->getValue("ATTR2", NULL);
```

For additional examples of this sort, see Section 6.1, “Using Synchronous Transactions”.

The second example uses an `NdbIndexOperation`:

```
// 1. Retrieve index object
myIndex= myDict->getIndex("MYINDEX", "MYTABLENAME");

// 2. Create
myOperation= myTransaction->getNdbIndexOperation(myIndex);

// 3. Define type of operation and lock mode
myOperation->readTuple(NdbOperation::LM_Read);

// 4. Specify Search Conditions
myOperation->equal("ATTR1", i);

// 5. Attribute Actions
myRecAttr = myOperation->getValue("ATTR2", NULL);
```

Another example of this second type can be found in Section 6.5, “Using Secondary Indexes in Scans”.

We now discuss in somewhat greater detail each step involved in the creation and use of synchronous transactions.

1. **Define single row operation type.** The following operation types are supported:

   - `NdbOperation::insertTuple()`: Inserts a non-existing tuple.
   - `NdbOperation::writeTuple()`: Updates a tuple if one exists, otherwise inserts a new tuple.
   - `NdbOperation::updateTuple()`: Updates an existing tuple.
   - `NdbOperation::deleteTuple()`: Deletes an existing tuple.
   - `NdbOperation::readTuple()`: Reads an existing tuple using the specified lock mode.

   All of these operations operate on the unique tuple key. When `NdbIndexOperation` is used, then each of these operations operates on a defined unique hash index.

   > **Note**
   >
   > If you want to define multiple operations within the same transaction, then you need to call `NdbTransaction::getNdbOperation()` or `NdbTransaction::getNdbIndexOperation()` for each operation.

2. **Specify Search Conditions.** The search condition is used to select tuples. Search conditions are set using `NdbOperation::equal()`.

3. **Specify Attribute Actions.** Next, it is necessary to determine which attributes should be read or updated. It is important to remember that:

   - Deletes can neither read nor set values, but only delete them.
   - Reads can only read values.
   - Updates can only set values. Normally the attribute is identified by name, but it is also possible

to use the attribute's identity to determine the attribute.

`NdbOperation::getValue()` returns an `NdbRecAttr` object containing the value as read. To obtain the actual value, one of two methods can be used; the application can either

- Use its own memory (passed through a pointer *`aValue`*) to `NdbOperation::getValue()`, or
- receive the attribute value in an `NdbRecAttr` object allocated by the NDB API.

The `NdbRecAttr` object is released when `Ndb::closeTransaction()` is called. For this reason, the application cannot reference this object following any subsequent call to `Ndb::closeTransaction()`. Attempting to read data from an `NdbRecAttr` object before calling `NdbTransaction::execute()` yields an undefined result.

1.3.2.3.2. Scan Operations

Scans are roughly the equivalent of SQL cursors, providing a means to perform high-speed row processing. A scan can be performed on either a table (using an `NdbScanOperation`) or an ordered index (by means of an `NdbIndexScanOperation`).

Scan operations have the following characteristics:

- They can perform read operations which may be shared, exclusive, or dirty.
- They can potentially work with multiple rows.
- They can be used to update or delete multiple rows.
- They can operate on several nodes in parallel.

After the operation is created using `NdbTransaction::getNdbScanOperation()` or `NdbTransaction::getNdbIndexScanOperation()`, it is carried out as follows:

1. Define the standard operation type, using `NdbScanOperation::readTuples()`.
2. Specify search conditions, using `NdbScanFilter`, `NdbIndexScanOperation::setBound()`, or both.
3. Specify attribute actions using `NdbOperation::getValue()`.
4. Execute the transaction using `NdbTransaction::execute()`.
5. Traverse the result set by means of successive calls to `NdbScanOperation::nextResult()`.

Here are two brief examples illustrating this process. Once again, in order to keep things relatively short and simple, we forego any error handling.

This first example performs a table scan using an `NdbScanOperation`:

```
// 1. Retrieve a table object
myTable= myDict->getTable("MYTABLENAME");

// 2. Create a scan operation (NdbScanOperation) on this table
```

```
myOperation= myTransaction->getNdbScanOperation(myTable);

// 3. Define the operation's type and lock mode
myOperation->readTuples(NdbOperation::LM_Read);

// 4. Specify search conditions
NdbScanFilter sf(myOperation);
sf.begin(NdbScanFilter::OR);
sf.eq(0, i);   // Return rows with column 0 equal to i or
sf.eq(1, i+1); // column 1 equal to (i+1)
sf.end();

// 5. Retrieve attributes
myRecAttr= myOperation->getValue("ATTR2", NULL);
```

The second example uses an NdbIndexScanOperation to perform an index scan:

```
// 1. Retrieve index object
myIndex= myDict->getIndex("MYORDEREDINDEX", "MYTABLENAME");

// 2. Create an operation (NdbIndexScanOperation object)
myOperation= myTransaction->getNdbIndexScanOperation(myIndex);

// 3. Define type of operation and lock mode
myOperation->readTuples(NdbOperation::LM_Read);

// 4. Specify search conditions
// All rows with ATTR1 between i and (i+1)
myOperation->setBound("ATTR1", NdbIndexScanOperation::BoundGE, i);
myOperation->setBound("ATTR1", NdbIndexScanOperation::BoundLE, i+1);

// 5. Retrieve attributes
myRecAttr = MyOperation->getValue("ATTR2", NULL);
```

Some additional discussion of each step required to perform a scan follows:

1. **Define Scan Operation Type.** It is important to remember that only a single operation is supported for each scan operation (`NdbScanOperation::readTuples()` or `NdbIndexScanOperation::readTuples()`).

   > **Note**
   >
   > If you want to define multiple scan operations within the same transaction, then you need to call `NdbTransaction::getNdbScanOperation()` or `NdbTransaction::getNdbIndexScanOperation()` separately for *each* operation.

2. **Specify Search Conditions.** The search condition is used to select tuples. If no search condition is specified, the scan will return all rows in the table. The search condition can be an `NdbScanFilter` (which can be used on both `NdbScanOperation` and `NdbIndexScanOperation`) or bounds (which can be used only on index scans - see `NdbIndexScanOperation::setBound()`). An index scan can use both `NdbScanFilter` and bounds.

   > **Note**
   >
   > When NdbScanFilter is used, each row is examined, whether or not it is actually returned. However, when using bounds, only rows within the bounds will be examined.

3. **Specify Attribute Actions.** Next, it is necessary to define which attributes should be read. As with transaction attributes, scan attributes are defined by name, but it is also possible to use the attributes' identities to define attributes as well. As discussed elsewhere in this document (see Section 1.3.2.2, “Synchronous Transactions”), the value read is returned by the `NdbOperation::getValue()` method as an `NdbRecAttr` object.

1.3.2.3.3. Using Scans to Update or Delete Rows

Scanning can also be used to update or delete rows. This is performed by

1. Scanning with exclusive locks using `NdbOperation::LM_Exclusive`.
2. (*When iterating through the result set*:) For each row, optionally calling either `NdbScanOperation::updateCurrentTuple()` or `NdbScanOperation::deleteCurrentTuple()`.
3. (*If performing `NdbScanOperation::updateCurrentTuple()`*:) Setting new values for records simply by using `NdbOperation::setValue()`. `NdbOperation::equal()` should not be called in such cases, as the primary key is retrieved from the scan.

> **Important**
>
> The update or delete is not actually performed until the next call to `NdbTransaction::execute()` is made, just as with single row operations. `NdbTransaction::execute()` also must be called before any locks are released; for more information, see Section 1.3.2.3.4, "Lock Handling with Scans".

Features Specific to Index Scans. When performing an index scan, it is possible to scan only a subset of a table using `NdbIndexScanOperation::setBound()`. In addition, result sets can be sorted in either ascending or descending order, using `NdbIndexScanOperation::readTuples()`. Note that rows are returned unordered by default unless *`sorted`* is set to `true`. It is also important to note that, when using `NdbIndexScanOperation::BoundEQ()` on a partition key, only fragments containing rows will actually be scanned. Finally, when performing a sorted scan, any value passed as the `NdbIndexScanOperation::readTuples()` method's *`parallel`* argument will be ignored and maximum parallelism will be used instead. In other words, all fragments which it is possible to scan are scanned simultaneously and in parallel in such cases.

1.3.2.3.4. Lock Handling with Scans

Performing scans on either a table or an index has the potential to return a great many records; however, Ndb locks only a predetermined number of rows per fragment at a time. The number of rows locked per fragment is controlled by the batch parameter passed to `NdbScanOperation::readTuples()`.

In order to allow the application to handle how locks are released, `NdbScanOperation::nextResult()` has a Boolean parameter *`fetchAllowed`*. If `NdbScanOperation::nextResult()` is called with *`fetchAllowed`* equal to `false`, then no locks may be released as result of the function call. Otherwise the locks for the current batch may be released.

This next example shows a scan delete that handles locks in an efficient manner. For the sake of brevity, we omit error-handling.

```
int check;

// Outer loop for each batch of rows
while((check = MyScanOperation->nextResult(true)) == 0)
{
  do
  {
    // Inner loop for each row within the batch
    MyScanOperation->deleteCurrentTuple();
  }
  while((check = MyScanOperation->nextResult(false)) == 0);

  // When there are no more rows in the batch, execute all defined deletes
  MyTransaction->execute(NoCommit);
}
```

For a more complete example of a scan, see Section 6.4, “Basic Scanning Example”.

1.3.2.3.5. Error Handling

Errors can occur either when operations making up a transaction are being defined, or when the transaction is actually being executed. Catching and handling either sort of error requires testing the value returned by `NdbTransaction::execute()`, and then, if an error is indicated (that is, if this value is equal to `-1`), using the following two methods in order to identify the error's type and location:

- `NdbTransaction::getNdbErrorOperation()` returns a reference to the operation causing the most recent error.
- `NdbTransaction::getNdbErrorLine()` yields the method number of the erroneous method in the operation, starting with `1`.

This short example illustrates how to detect an error and to use these two methods to identify it:

```
theTransaction = theNdb->startTransaction();
theOperation = theTransaction->getNdbOperation("TEST_TABLE");
if(theOperation == NULL)
  goto error;

theOperation->readTuple(NdbOperation::LM_Read);
theOperation->setValue("ATTR_1", at1);
theOperation->setValue("ATTR_2", at1);  //  Error occurs here
theOperation->setValue("ATTR_3", at1);
theOperation->setValue("ATTR_4", at1);

if(theTransaction->execute(Commit) == -1)
{
  errorLine = theTransaction->getNdbErrorLine();
  errorOperation = theTransaction->getNdbErrorOperation();
}
```

Here, `errorLine` is `3`, as the error occurred in the third method called on the `NdbOperation` object (in this case, `theOperation`). If the result of `NdbTransaction::getNdbErrorLine()` is `0`, then the error occurred when the operations were executed. In this example, `errorOperation` is a pointer to the object `theOperation`. The `NdbTransaction::getNdbError()` method returns an `NdbError` object providing information about the error.

> **Note**
>
> Transactions are *not* automatically closed when an error occurs. You must call `Ndb::closeTransaction()` or `NdbTransaction::close()` to close the transaction.
>
> See Section 3.1.1.9, “`Ndb::closeTransaction()`”, and Section 3.9.2.7, “`NdbTransaction::close()`”.

One recommended way to handle a transaction failure (that is, when an error is reported) is as shown here:

1. Roll back the transaction by calling `NdbTransaction::execute()` with a special `ExecType` value for the *`type`* parameter.

   See Section 3.9.2.5, “`NdbTransaction::execute()`” and Section 3.9.1.3, “The `NdbTransaction::ExecType` Type”, for more information about how this is done.

2. Close the transaction by calling `NdbTransaction::closeTransaction()`.

3. If the error was temporary, attempt to restart the transaction.

Several errors can occur when a transaction contains multiple operations which are simultaneously executed. In this case the application must go through all operations and query each of their `NdbError` objects to find out what really happened.

> **Important**
>
> Errors can occur even when a commit is reported as successful. In order to handle such situations, the NDB API provides an additional `NdbTransaction::commitStatus()` method to check the transaction's commit status.
>
> See Section 3.9.2.10, “`NdbTransaction::commitStatus()`”.

1.3.3. Review of MySQL Cluster Concepts

This seciton covers the NDB Kernel, and discusses MySQL Cluster transaction handling and transaction coordinators. It also describes NDB record structures and concurrency issues.

The *NDB Kernel* is the collection of data nodes belonging to a MySQL Cluster. The application programmer can for most purposes view the set of all storage nodes as a single entity. Each data node is made up of three main components:

- **TC**: The transaction coordinator.
- **ACC**: The index storage component.
- **TUP**: The data storage component.

When an application executes a transaction, it connects to one transaction coordinator on one data node. Usually, the programmer does not need to specify which TC should be used, but in some cases where performance is important, the programmer can provide “hints” to use a certain TC. (If the node with the desired transaction coordinator is down, then another TC will automatically take its place.)

Each data node has an ACC and a TUP which store the indexes and data portions of the database table fragment. Even though a single TC is responsible for the transaction, several ACCs and TUPs on other data nodes might be involved in that transaction's execution.

1.3.3.1. Selecting a Transaction Coordinator

The default method is to select the transaction coordinator (TC) determined to be the "nearest" data node, using a heuristic for proximity based on the type of transporter connection. In order of nearest to most distant, these are:

1. SCI
2. SHM
3. TCP/IP (localhost)
4. TCP/IP (remote host)

If there are several connections available with the same proximity, one is selected for each transaction in a round-robin fashion. Optionally, you may set the method for TC selection to round-robin mode, where each new set of transactions is placed on the next data node. The pool of connections from which this selection is made consists of all available connections.

As noted in Section 1.3.3, "Review of MySQL Cluster Concepts", the application programmer can provide hints to the NDB API as to which transaction coordinator should be uses. This is done by providing a table and a partition key (usually the primary key). If the primary key as the partition key, then the transaction is placed on the node where the primary replica of that record resides. Note that this is only a hint; the system can be reconfigured at any time, in which case the NDB API chooses a transaction coordinator without using the hint. For more information, see Section 3.4.2.2.16, "`Column::getPartitionKey()`", and Section 3.1.1.8, "`Ndb::startTransaction()`". The application programmer can specify the partition key from SQL by using this construct:

```
CREATE TABLE ... ENGINE=NDB PARTITION BY KEY (attribute_list);
```

For additional information, see Partitioning [http://dev.mysql.com/doc/refman/5.1/en/partitioning.html], and in particular `KEY` Partitioning [http://dev.mysql.com/doc/refman/5.1/en/partitioning-key.html], in the MySQL Manual.

1.3.3.2. NDB Record Structure

The `NDB Cluster` storage engine used by MySQL Cluster is a relational database engine storing records in tables as with other relational database systems. Table rows represent records as tuples of relational data. When a new table is created, its attribute schema is specified for the table as a whole, and thus each table row has the same structure. Again, this is typical of relational databases, and `NDB` is no different in this regard.

Primary Keys. Each record has from 1 up to 32 attributes which belong to the primary key of the table.

Transactions. Transactions are committed first to main memory, and then to disk, after a global checkpoint (GCP) is issued. Since all data are (in most NDB Cluster configurations) synchronously replicated and stored on multiple data nodes, the system can handle processor failures without loss of data. However, in the case of a system-wide failure, all transactions (committed or not) occurring since the most recent GCP are lost.

Concurrency Control. `NDB Cluster` uses *pessimistic concurrency control* based on locking. If a requested lock (implicit and depending on database operation) cannot be attained within a specified time, then a timeout error results.

Concurrent transactions as requested by parallel application programs and thread-based applications can sometimes deadlock when they try to access the same information simultaneously. Thus, applications need to be written in a manner such that timeout errors occurring due to such deadlocks are handled gracefully. This generally means that the transaction encountering a timeout should be rolled back and restarted.

Hints and Performance. Placing the transaction coordinator in close proximity to the actual data used in the transaction can in many cases improve performance significantly. This is particularly true for systems using TCP/IP. For example, a Solaris system using a single 500 MHz processor has a cost model for TCP/IP communication which can be represented by the formula

```
[30 microseconds] + ([100 nanoseconds] * [number of bytes])
```

This means that if we can ensure that we use "popular" links we increase buffering and thus drastically reduce the costs of communication. The same system using SCI has a different cost model:

```
[5 microseconds] + ([10 nanoseconds] * [number of bytes])
```

This means that the efficiency of an SCI system is much less dependent on selection of transaction coordinators. Typically, TCP/IP systems spend 30 to 60% of their working time on communication, whereas for SCI systems this figure is in the range of 5 to 10%. Thus, employing SCI for data transport means that less effort from the NDB API programmer is required and greater scalability can be achieved, even for applications using data from many different parts of the database.

A simple example would be an application that uses many simple updates where a transaction needs to update one record. This record has a 32-bit primary key which also serves as the partitioning key. Then the `keyData` is used as the address of the integer of the primary key and `keyLen` is `4`.

1.3.4. The Adaptive Send Algorithm

Discusses the mechanics of transaction handling and transmission in MySQL Cluster and the NDB API, and the objects used to implement these.

When transactions are sent using `NdbTransaction::execute()`, they are not immediately transferred to the NDB Kernel. Instead, transactions are kept in a special send list (buffer) in the `Ndb` object to which they belong. The adaptive send algorithm decides when transactions should actually be transferred to the NDB kernel.

The NDB API is designed as a multi-threaded interface, and so it is often desirable to transfer database operations from more than one thread at a time. The NDB API keeps track of which `Ndb` objects are active in transferring information to the NDB kernel and the expected number of threads to interact with the NDB kernel. Note that a given instance of `Ndb` should be used in at most one thread; different threads should *not* share the same `Ndb` object.

There are four conditions leading to the transfer of database operations from `Ndb` object buffers to the NDB kernel:

1. The NDB Transporter (TCP/IP, SCI, or shared memory) decides that a buffer is full and sends it off. The buffer size is implementation-dependent and may change between MySQL Cluster releases. When TCP/IP is the transporter, the buffer size is usually around 64 KB. Since each `Ndb` object provides a single buffer per data node, the notion of a “full” buffer is local to each data node.

2. The accumulation of statistical data on transferred information may force sending of buffers to all storage nodes (that is, when all the buffers become full).

3. Every 10 ms, a special transmission thread checks whether or not any send activity has occurred. If not, then the thread will force transmission to all nodes. This means that 20 ms is the maximum amount of time that database operations are kept waiting before being dispatched. A 10-millisecond limit is likely in future releases of MySQL Cluster; checks more frequent than this require additional support from the operating system.

4. For methods that are affected by the adaptive send algorithm (such as `NdbTransaction::execute()`), there is a *`force`* parameter that overrides its default behaviour in this regard and forces immediate transmission to all nodes. See the individual NDB API class listings for more information.

> **Note**
>
> The conditions listed above are subject to change in future releases of MySQL Cluster.

Chapter 2. GETTING STARTED WITH THE NDB API

This chapter discusses preparations for writing an NDB API application.

2.1. Compiling and Linking NDB API Programs

This section provides information on compiling and linking NDB API applications, including requirements and compiler and linker options.

2.1.1. General Requirements

To use the NDB API with MySQL, you must have the NDB client library and its header files installed alongside the regular MySQL client libraries and headers. These are automatically installed when you build MySQL using the `--with-ndbcluster configure` option or when using a MySQL binary package that supports the `NDBCluster` storage engine.

> **Note**
>
> MySQL 4.1 does not install the required NDB-specific header files. You should use MySQL 5.0 or later when writing NDB API applications, and this Guide is targeted for use with MySQL 5.1.
>
> The library and header files were not included in MySQL 5.1 binary distributions prior to MySQL 5.1.12; beginning with 5.1.12, you can find them in `/usr/include/storage/ndb`. This issue did not occur when compiling MySQL 5.1 from source.

2.1.2. Compiler Options

Header Files. In order to compile source files that use the NDB API, you must ensure that the necessary header files can be found. Header files specific to the NDB API are installed in the following subdirectories of the MySQL `include` directory:

- `include/mysql/storage/ndb/ndbapi`
- `include/mysql/storage/ndb/mgmapi`

Compiler Flags. The MySQL-specific compiler flags needed can be determined using the `mysql_config` utility that is part of the MySQL installation:

```
$ mysql_config --cflags
-I/usr/local/mysql/include/mysql -Wreturn-type -Wtrigraphs -W -Wformat
-Wsign-compare -Wunused  -mcpu=pentium4 -march=pentium4
```

This sets the include path for the MySQL header files but not for those specific to the NDB API. The `--include` option to `mysql_config` returns the generic include path switch:

```
shell> mysql_config --include
-I/usr/local/mysql/include/mysql
```

It is necessary to add the subdirectory paths explicitly, so that adding all the needed compile flags to the `CXXFLAGS` shell variable should look something like this:

```
CFLAGS="$CFLAGS "`mysql_config --cflags`
```

```
CFLAGS="$CFLAGS "`mysql_config --include`storage/ndb
CFLAGS="$CFLAGS "`mysql_config --include`storage/ndb/ndbapi
CFLAGS="$CFLAGS "`mysql_config --include`storage/ndb/mgmapi
```

> **Tip**
>
> If you do not intend to use the Cluster management functions, the last line in the previous example can be omitted. However, if you are interested in the management functions only, and do not want or need to access Cluster data except from MySQL, then you can omit the line referencing the `ndbapi` directory.

2.1.3. Linker Options

`NDB` API applications must be linked against both the MySQL and `NDB` client libraries. The `NDB` client library also requires some functions from the `mystrings` library, so this must be linked in as well.

The necessary linker flags for the MySQL client library are returned by `mysql_config --libs`. For multithreaded applications you should use the `--libs_r` instead:

```
$ mysql_config --libs_r
-L/usr/local/mysql-5.1/lib/mysql -lmysqlclient_r -lz -lpthread -lcrypt
-lnsl -lm -lpthread -L/usr/lib -lssl -lcrypto
```

To link an NDB API application, it is necessary to add `-lndbclient` and `-lmystrings` to these options. Adding all the required linker flags to the `LDFLAGS` variable should look something like this:

```
LDFLAGS="$LDFLAGS "`mysql_config --libs_r`
LDFLAGS="$LDFLAGS -lndbclient -lmystrings"
```

2.1.4. Using Autotools

It is often faster and simpler to use GNU autotools than to write your own makefiles. In this section, we provide an autoconf macro `WITH_MYSQL` that can be used to add a `--with-mysql` option to a configure file, and that automatically sets the correct compiler and linker flags for given MySQL installation.

All of the examples in this chapter include a common `mysql.m4` file defining `WITH_MYSQL`. A typical complete example consists of the actual source file and the following helper files:

- `acinclude`
- `configure.in`
- `Makefile.m4`

`automake` also requires that you provide `README`, `NEWS`, `AUTHORS`, and `ChangeLog` files; however, these can be left empty.

To create all necessary build files, run the following:

```
aclocal
autoconf
automake -a -c
configure --with-mysql=/mysql/prefix/path
```

Normally, this needs to be done only once, after which `make` will accommodate any file changes.

Example 1-1: `acinclude.m4`.

```
m4_include([../mysql.m4])
```

Example 1-2: `configure.in`.

```
AC_INIT(example, 1.0)
AM_INIT_AUTOMAKE(example, 1.0)
WITH_MYSQL()
AC_OUTPUT(Makefile)
```

Example 1-3: `Makefile.am`.

```
bin_PROGRAMS = example
example_SOURCES = example.cc
```

Example 1-4: `WITH_MYSQL` source for inclusion in `acinclude.m4`.

```
dnl
dnl configure.in helper macros
dnl

AC_DEFUN([WITH_MYSQL], [
  AC_MSG_CHECKING(for mysql_config executable)

  AC_ARG_WITH(mysql, [  --with-mysql=PATH path to mysql_config binary or mysql prefix dir], [
  if test -x $withval -a -f $withval
    then
      MYSQL_CONFIG=$withval
    elif test -x $withval/bin/mysql_config -a -f $withval/bin/mysql_config
    then
     MYSQL_CONFIG=$withval/bin/mysql_config
    fi
  ], [
  if test -x /usr/local/mysql/bin/mysql_config -a -f /usr/local/mysql/bin/mysql_config
    then
      MYSQL_CONFIG=/usr/local/mysql/bin/mysql_config
    elif test -x /usr/bin/mysql_config -a -f /usr/bin/mysql_config
    then
      MYSQL_CONFIG=/usr/bin/mysql_config
    fi
  ])

  if test "x$MYSQL_CONFIG" = "x"
  then
    AC_MSG_RESULT(not found)
    exit 3
  else
    AC_PROG_CC
    AC_PROG_CXX

    # add regular MySQL C flags
    ADDFLAGS=`$MYSQL_CONFIG --cflags`

    # add NDB API specific C flags
    IBASE=`$MYSQL_CONFIG --include`
    ADDFLAGS="$ADDFLAGS $IBASE/storage/ndb"
    ADDFLAGS="$ADDFLAGS $IBASE/storage/ndb/ndbapi"
    ADDFLAGS="$ADDFLAGS $IBASE/storage/ndb/mgmapi"

    CFLAGS="$CFLAGS $ADDFLAGS"
    CXXFLAGS="$CXXFLAGS $ADDFLAGS"

    LDFLAGS="$LDFLAGS "`$MYSQL_CONFIG --libs_r`" -lndbclient -lmystrings -lmysys"
    LDFLAGS="$LDFLAGS "`$MYSQL_CONFIG --libs_r`" -lndbclient -lmystrings"

    AC_MSG_RESULT($MYSQL_CONFIG)
  fi
])
```

2.2. Connecting to the Cluster

This section covers connecting an NDB API application to a MySQL cluster.

2.2.1. Include Files

NDB API applications require one or more of the following include files:

- Applications accessing Cluster data via the NDB API must include the file NdbApi.hpp.
- Applications making use of both the NDB API and the regular MySQL client API also need to include mysql.h.
- Applications that use cluster management functions need the include file mgmapi.h.

2.2.2. API Initialisation and Cleanup

Before using the NDB API, it must first be initialised by calling the ndb_init() function. Once an NDB API application is complete, call ndb_end(0) to perform a cleanup.

2.2.3. Establishing the Connection

To establish a connection to the server, it is necessary to create an instance of Ndb_cluster_connection, whose constructor takes as its argument a cluster connectstring; if no connectstring is given, localhost is assumed.

The cluster connection is not actually initiated until the Ndb_cluster_connection::connect() method is called. When invoked without any arguments, the connection attempt is retried each 1 second indefinitely until successful, and no reporting is done. See Section 3.2, “The Ndb_cluster_connection Class”, for details.

By default an API node will connect to the “nearest” data node — usually a data node running on the same machine, due to the fact that shared memory transport can be used instead of the slower TCP/IP. This may lead to poor load distribution in some cases, so it is possible to enforce a round-robin node connection scheme by calling the set_optimized_node_selection() method with 0 as its argument prior to calling connect(). (See Section 3.2.1.4, “Ndb_cluster_connection::set_optimized_node_selection()”.)

The connect() method initiates a connection to a cluster management node only — it does not wait for any connections to data nodes to be made. This can be accomplished by using wait_until_ready() after calling connect(). The wait_until_ready() method waits up to a given number of seconds for a connection to a data node to be established.

In the following example, initialisation and connection are handled in the two functions example_init() and example_end(), which will be included in subsequent examples via the file example_connection.h.

Example 2-1: Connection example.

```
#include <stdio.h>
#include <stdlib.h>
#include <NdbApi.hpp>
#include <mysql.h>
#include <mgmapi.h>

Ndb_cluster_connection* connect_to_cluster();
void disconnect_from_cluster(Ndb_cluster_connection *c);

Ndb_cluster_connection* connect_to_cluster()
{
  Ndb_cluster_connection* c;

  if(ndb_init())
    exit(EXIT_FAILURE);
```

```
  c= new Ndb_cluster_connection();

  if(c->connect(4, 5, 1))
  {
    fprintf(stderr, "Unable to connect to cluster within 30 seconds.\n\n");
    exit(EXIT_FAILURE);
  }

  if(c->wait_until_ready(30, 0) < 0)
  {
    fprintf(stderr, "Cluster was not ready within 30 seconds.\n\n");
    exit(EXIT_FAILURE);
  }
}

void disconnect_from_cluster(Ndb_cluster_connection *c)
{
  delete c;

  ndb_end(2);
}

int main(int argc, char* argv[])
{
  Ndb_cluster_connection *ndb_connection= connect_to_cluster();

  printf("Connection Established.\n\n");

  disconnect_from_cluster(ndb_connection);

  return EXIT_SUCCESS;
}
```

2.3. Mapping MySQL Database Object Names and Types to NDB

This section discusses NDB naming and other conventions with regard to database objects.

Databases and Schemas. Databases and schemas are not represented by objects as such in the NDB API. Instead, they are modelled as attributes of Table and Index objects. The value of the database attribute of one of these objects is always the same as the name of the MySQL database to which the table or index belongs. The value of the schema attribute of a Table or Index object is always 'def' (for “default”).

Tables. MySQL table names are directly mapped to NDB table names without modification. Table names starting with 'NDB$' are reserved for internal use>, as is the SYSTAB_0 table in the sys database.

Indexes. There are two different type of NDB indexes:

- *Hash indexes* are unique, but not ordered.
- *B-tree indexes* are ordered, but allow duplicate values.

Names of unique indexes and primary keys are handled as follows:

- For a MySQL UNIQUE index, both a B-tree and a hash index are created. The B-tree index uses the MySQL name for the index; the name for the hash index is generated by appending '$unique' to the index name.
- For a MySQL primary key only a B-tree index is created. This index is given the name PRIMARY. There is no extra hash; however, the uniqueness of the primary key is guaranteed by making the MySQL key the internal primary key of the NDB table.

Column Names and Values. NDB column names are the same as their MySQL names.

Datatypes. MySQL datatypes are stored in NDB columns as follows:

- The MySQL TINYINT, SMALLINT, INT, and BIGINT datatypes map to NDB types having the same names and storage requirements as their MySQL counterparts.
- The MySQL FLOAT and DOUBLE datatypes are mapped to NDB types having the same names and storage requirements.
- The storage space required for a MySQL CHAR column is determined by the maximum number of characters and the column's character set. For most (but not all) character sets, each character takes one byte of storage. When using UTF-8, each character requires three bytes. You can find the number of bytes needed per character in a given character set by checking the Maxlen column in the output of SHOW CHARACTER SET.
- In MySQL 5.1, the storage requirements for a VARCHAR or VARBINARY column depend on whether the column is stored in memory or on disk:
  - For in-memory columns, the NDBCLUSTER storage engine supports variable-width columns with 4-byte alignment. This means that (for example) a the string 'abcde' stored in a VARCHAR(50) column using the latin1 character set requires 12 bytes — in this case, 2 bytes times 5 characters is 10, rounded up to the next even multiple of 4 yields 12. (This represents a change in behaviour from Cluster in MySQL 5.0 and 4.1, where a column having the same definition required 52 bytes storage per row regardless of the length of the string being stored in the row.)
  - In Disk Data columns, VARCHAR and VARBINARY are stored as fixed-width columns. This means that each of these types requires the same amount of storage as a CHAR of the same size.
- Each row in a Cluster BLOB or TEXT column is made up of two separate parts. One of these is of fixed size (256 bytes), and is actually stored in the original table. The other consists of any data in excess of 256 bytes, which stored in a hidden table. The rows in this second table are always 2000 bytes long. This means that record of *size* bytes in a TEXT or BLOB column requires
  - 256 bytes, if *size* <= 256
  - 256 + 2000 * ((*size* – 256) \ 2000) + 1) bytes otherwise

Chapter 3. NDB API CLASSES

This chapter provides a detailed listing of all public NDB API classes.

Each class listing includes:

- Description of the purpose of the class in the API.
- A diagram of the class showing its public members and types.

  The sections of this covering the NdbDictionary and NdbOperation classes also include entity-relationship diagrams showing the hierarchy of inner classes, subclasses, and public types descending from them.
- Listings of all public members, including descriptions of all method parameters and types.

3.1. The Ndb Class

This class represents the NDB kernel; it is the primary class of the NDB API.

Description. Any non-trivial NDB API program makes use of at least one instance of Ndb. By using several Ndb objects, it is possible to implement a multi-threaded application. You should remember that one Ndb object cannot be shared between threads; however, it is possible for a single thread to use multiple Ndb objects. A single application process can support a maximum of 128 Ndb objects.

> **Note**
>
> The Ndb object is multi-thread safe in that each Ndb object can be handled by one thread at a time. If an Ndb object is handed over to another thread, then the application must ensure that a memory barrier is used to ensure that the new thread sees all updates performed by the previous thread.
>
> Semaphores and mutexes are examples of easy ways to provide memory barriers without having to bother about the memory barrier concept.

It is also possible to use multiple Ndb objects to perform operations on different clusters in a single application. See the Note onApplication-Level Partitioning for conditions and restrictions applying to such usage.

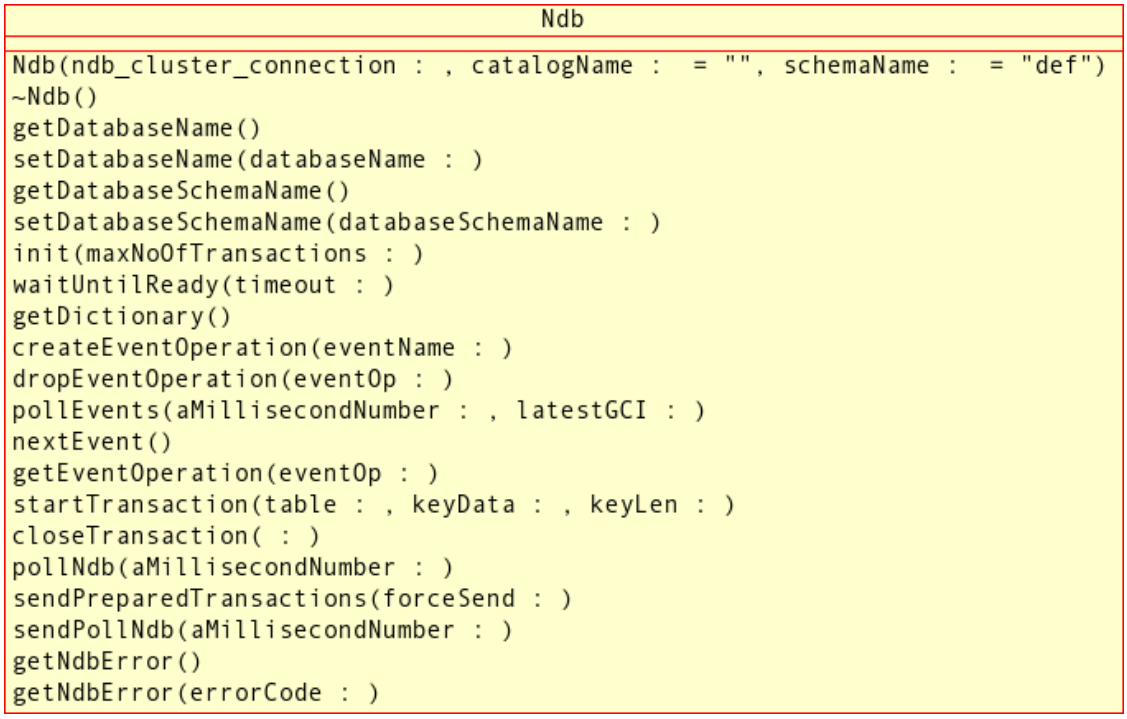
Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
Ndb()	Class constructor; represents a connection to a MySQL Cluster.
~Ndb()	Class destructor; terminates a Cluster connection when it is no longer to be used
init()	Initialises an Ndb object and makes it ready for use.
getDictionary()	Gets a dictionary, which is used for working with database schema information.
getDatabaseName()	Gets the name of the current database.
setDatabaseName()	Sets the name of the current database.
getDatabaseSchemaName()	Gets the name of the current database schema.

Method	Purpose / Use
setDatabaseSchemaName()	Sets the name of the current database schema.
startTransaction()	Begins a transaction. (See Section 3.9, "The NdbTransaction Class".)
closeTransaction()	Closes a transaction.
createEventOperation()	Creates a subscription to a database event. (See Section 3.5, "The NdbEventOperation Class".)
dropEventOperation()	Drops a subscription to a database event.
pollEvents()	Waits for an event to occur.
getNdbError()	Retrieves an error. (See Section 4.1, "The NdbError Structure".)

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.1.1, "Ndb Class Methods".

Class Diagram. This diagram shows all the available methods of the Ndb class:



3.1.1. Ndb Class Methods

The sections that follow discuss the public methods of the Ndb class.

3.1.1.1. Ndb Class Constructor

Description. This creates an instance of Ndb, which represents a connection to the MySQL Cluster. All NDB API applications should begin with the creation of at least one Ndb object. This requires the creation of at least one instance of Ndb_cluster_connection, which serves as a container for a cluster connectstring.

Signature.

```
Ndb
    (
    Ndb_cluster_connection *ndb_cluster_connection,
    const char             *catalogName = "",
    const char             *schemaName = "def"
    )
```

Parameters. The `Ndb` class constructor can take up to 3 parameters, of which only the first is required:

- *`ndb_cluster_connection`* is an instance of `Ndb_cluster_connection`, which represents a cluster connectstring. (See Section 3.2, "The `Ndb_cluster_connection` Class".)
- *`catalogName`* is an optional parameter providing a namespace for the tables and indexes created in any connection from the `Ndb` object.

  The default value for this parameter is an empty string.
- The optional *`schemaName`* provides an additional namespace for the tables and indexes created in a given catalog.

  The default value for this parameter is the string "def".

Return Value. An `Ndb` object.

`~Ndb()` (Class Destructor). The destructor for the `Ndb` class should be called in order to terminate an instance of `Ndb`. It requires no arguments, nor any special handling.

3.1.1.2. `Ndb::init()`

Description. This method is used to initialise an `Ndb` object.

Signature.

```
int init
    (
      int maxNoOfTransactions = 4
    )
```

Parameters. The `init()` method takes a single parameter *`maxNoOfTransactions`* of type integer. This parameter specifies the maximum number of parallel `NdbTransaction` objects that can be handled by this instance of `Ndb`. The maximum permitted value for *`maxNoOfTransactions`* is 1024; if not specified, it defaults to 4.

> **Note**
>
> Each scan or index operation uses an extra `NdbTransaction` object. See Section 3.9, "The `NdbTransaction` Class".

Return Value. This method returns an `int`, which can be either of two values:

- **0**: indicates that the `Ndb` object was initialised successfully.
- **-1**: indicates failure.

3.1.1.3. `Ndb::getDictionary()`

Description. This method is used to obtain an object for retrieving or manipulating database schema information. This `Dictionary` object contains meta-information about all tables in the cluster.

> **Note**
>
> The dictionary returned by this method operates independently of any transaction. See Section 3.4.1, "The `Dictionary` Class", for more information.

Signature.

```
NdbDictionary::Dictionary* getDictionary
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An instance of the `Dictionary` class.

3.1.1.4. `Ndb::getDatabaseName()`

Description. This method can be used to obtain the name of the current database.

Signature.

```
const char* getDatabaseName
    (
      void
    )
```

Parameters. None.

Return Value. The name of the current database.

3.1.1.5. `Ndb::setDatabaseName()`

Description. This method is used to set the name of the current database.

Signature.

```
void setDatabaseName
    (
      const char *databaseName
    )
```

Parameters. `setDatabaseName()` takes a single, required parameter, the name of the new database to be set as the current database.

Return Value. N/A.

3.1.1.6. `Ndb::getDatabaseSchemaName()`

Description. This method can be used to obtain the current database schema name.

Signature.

```
const char* getDatabaseSchemaName
    (
      void
    )
```

Parameters. None.

Return Value. The name of the current database schema.

3.1.1.7. Ndb::setDatabaseSchemaName()

Description. This method allows you to set the name of the current database schema.

Signature.

```
void setDatabaseSchemaName
    (
      const char *databaseSchemaName
    )
```

Parameters. The name of the database schema.

Return Value. N/A.

3.1.1.8. Ndb::startTransaction()

Description. This method is used to begin a new transaction.

> **Important**
>
> When the transaction is completed it must be closed using NdbTransaction::close() or Ndb::closeTransaction(). This must be done regardless of the transaction's final outcome, even if it fails due to an error.
>
> See Section 3.1.1.9, "Ndb::closeTransaction()", and Section 3.9.2.7, "NdbTransaction::close()".

Signature.

```
NdbTransaction* startTransaction
    (
      const NdbDictionary::Table *table = 0,
      const char                 *keyData = 0,
      Uint32                     *keyLen = 0
    )
```

Parameters. This method takes three parameters, as follows:

- *table*: A pointer to a Table object. (See Section 3.4.3.7, "The Table Class".) This is used to determine on which node the Transaction Coordinator should run.
- *keyData*: A pointer to a partition key corresponding to *table*.
- *keyLen*: The length of the partition key, expressed in bytes.

Return Value. An NdbTransaction object. See Section 3.9, "The NdbTransaction Class".

3.1.1.9. Ndb::closeTransaction()

Description. This is one of two NDB API methods provided for closing a transaction (the other being NdbTransaction::close() — see Section 3.9.2.7, "NdbTransaction::close()"). You must call one of these two methods to close the transaction once it has been completed, whether or not the transaction succeeded.

Signature.

```
void closeTransaction
    (
      NdbTransaction *transaction
    )
```

Parameters. This method takes a single argument, a pointer to the `NdbTransaction` to be closed.

Return Value. N/A.

3.1.1.10. `Ndb::createEventOperation`

Description. This method creates a subscription to a database event.

Signature.

```
NdbEventOperation* createEventOperation
    (
      const char *eventName
    )
```

Parameters. This method takes a single argument, the unique *`eventName`* identifying the event to which you wish to subscribe.

Return Value. A pointer to an `NdbEventOperation` object (or `NULL`, in the event of failure). See Section 3.5, "The `NdbEventOperation` Class".

3.1.1.11. `Ndb::dropEventOperation()`

Description. This method drops a subscription to a database event represented by an `NdbEventOperation` object.

Signature.

```
int dropEventOperation
    (
      NdbEventOperation *eventOp
    )
```

Parameters. This method requires a single input parameter, a pointer to an instance of `NdbEventOperation`.

Return Value. **0** on success; any other result indicates failure.

3.1.1.12. `Ndb::pollEvents()`

Description. This method waits for an event to occur, and returns as soon as an event is detected in any existing subscription belonging to the `Ndb` object for which the method is invoked. It is used to determine whether any events are available in the subscription queue.

Signature.

```
int pollEvents
    (
      int     maxTimeToWait,
      Uint64* latestGCI = 0
    )
```

Parameters. This method takes two parameters, as shown here:

- The maximum time to wait, in milliseconds, before “giving up” and reporting that no events were available (that is, before the method automatically returns **0**).
- The index of the most recent global checkpoint. Normally, this may safely be permitted to assume its default value, which is **0**.

Return Value. `pollEvents()` returns a value of type `int`, which may be interpreted as follows:

- > **0**: There are events available in the queue.
- **0**: There are no events available.
- < **0**: Indicates failure (possible error).

3.1.1.13. `Ndb::nextEvent()`

Description. Returns the next event operation having data from a subscription queue.

Signature.

```
NdbEventOperation* nextEvent
    (
      void
    )
```

Parameters. None.

Return Value. This method returns an `NdbEventOperation` object representing the next event in a subscription queue, if there is such an event. If there is no event in the queue, it returns `NULL` instead. (See Section 3.5, “The `NdbEventOperation` Class”.)

3.1.1.14. `Ndb::getNdbError()`

Description. This method provides you with two different ways to obtain an `NdbError` object representing an error condition. For more detailed information about error handling in the `NDB` API, see Chapter 4, *NDB API ERRORS*.

Signature. The `getNdbError()` method actually has two variants. The first of these simply gets the most recent error to have occurred:

```
const NdbError& getNdbError
    (
      void
    )
```

The second variant returns the error corresponding to a given error code:

```
const NdbError& getNdbError
    (
      int errorCode
    )
```

Regardless of which version of the method is used, the `NdbError` object returned persists until the next `NDB` API method is invoked.

Parameters. To obtain the most recent error, simply call `getNdbError()` without any parameters. To obtain the error matching a specific *`errorCode`*, invoke the method passing the code (an `int`) to it as a parameter. For a listing of `NDB` API error codes and corresponding error messages, see Section 4.2, “`NDB` Error Codes, Classifications, and Messages”.

Return Value. An `NdbError` object containing information about the error, including its type and, where applicable, contextual information as to how the error arose. See Section 4.1, "The `NdbError` Structure", for details.

3.2. The `Ndb_cluster_connection` Class

This class represents a connection to a cluster of data nodes.

Description. An NDB application program should begin with the creation of a single `Ndb_cluster_connection` object, and typically makes use of a single `Ndb_cluster_connection`. The application connects to a cluster management server when this object's `connect()` method is called. By using the `wait_until_ready()` method it is possible to wait for the connection to reach one or more data nodes.

> **Note**
>
> There is no restriction against instantiating multiple `Ndb_cluster_connection` objects representing connections to different management servers in a single application, nor against using these for creating multiple instances of the `Ndb` class. Such `Ndb_cluster_connection` objects (and the `Ndb` instances based on them) are not required even to connect to the same cluster.
>
> For example, it is entirely possible to perform *application-level partitioning* of data in such a manner that data meeting one set of criteria are "handed off" to one cluster via an `Ndb` object that makes use of an `Ndb_cluster_connection` object representing a connection to that cluster, while data not meeting those criteria (or perhaps a different set of criteria) can be sent to a different cluster through a different instance of Ndb that makes use of an `Ndb_cluster_connection` "pointing" to the second cluster.
>
> It is possible to extend this scenario to develop a single application that accesses an arbitrary number of clusters. However, in doing so, the following conditions and requirements must be kept in mind:
>
> - A cluster management server (`ndb_mgmd`) can connect to one and only one cluster without being restarted and reconfigured, as it must read the data telling it which data nodes make up the cluster from a configuration file (`config.ini`).
> - An `Ndb_cluster_connection` object "belongs" to a single management server whose hostname or IP address is used in instantiating this object (passed as the *`connectstring`* argument to its constructor); once the object is created, it cannot be used to initiate a connection to a different management server.
>
>   (See Section 3.2.1.1, "`Ndb_cluster_connection` Class Constructor".)
> - An `Ndb` object making use of this connection (`Ndb_cluster_connection`) cannot be re-used to connect to a different cluster management server (and thus to a different collection of data nodes making up a cluster). Any given instance of `Ndb` is bound to a specific `Ndb_cluster_connection` when created, and that `Ndb_cluster_connection` is in turn bound to a single and unique management server when it is instantiated.
>
>   (See Section 3.1.1.1, "`Ndb` Class Constructor".)
> - The bindings described above persist for the lifetimes of the `Ndb` and `Ndb_cluster_connection` objects in question.
>
> Therefore, it is imperative in designing and implementing any application that accesses mul-

tiple clusters in a single session, that a separate set of `Ndb_cluster_connection` and `Ndb` objects be instantiated for connecting to each cluster management server, and that no confusion arises as to which of these is used to access which MySQL Cluster.

It is also important to keep in mind that no direct "sharing" of data or data nodes between different clusters is possible. A data node can belong to one and only one cluster, and any movement of data between clusters must be accomplished on the application level.

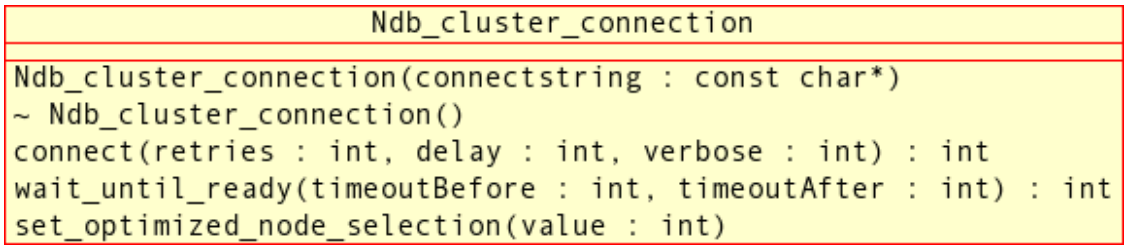
For examples demonstrating how connections to two different clusters can be made and used in a single application, see Section 6.2, "Using Synchronous Transactions and Multiple Clusters", and Section 6.7, "Event Handling with Multiple Clusters".

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Ndb_cluster_connection()`	Constructor; Creates a connection to a cluster of data nodes.
`connect()`	Connects to a cluster management server.
`wait_until_ready()`	Waits until a connection with one or more data nodes is successful.
`set_optimzed_node_selection()`	Used to control node-selection behaviour.

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.2.1, "`Ndb_cluster_connection` Class Methods".

Class Diagram. This diagram shows all the available methods of the `Ndb_cluster_connection` class:



3.2.1. `Ndb_cluster_connection` Class Methods

3.2.1.1. `Ndb_cluster_connection` Class Constructor

Description. This method creates a connection to a MySQL cluster, that is, to a cluster of data nodes. The object returned by this method is required in order to instantiate an `Ndb` object. (See Section 3.1, "The `Ndb` Class".) Thus, every `NDB` API application requires the use of an `Ndb_cluster_connection`.

Signature.

```
Ndb_cluster_connection
    (
      const char* connectstring = 0
    )
```

Parameters. This method requires a single parameter — a *`connectstring`* pointing to the location of the management server.

Return Value. An instance of `Ndb_cluster_connection`.

3.2.1.2. `Ndb_cluster_connection::connect()`

Description. This method connects to a cluster management server.

Signature.

```
int connect
    (
      int retries = 0,
      int delay   = 1,
      int verbose = 0
    )
```

Parameters. This method takes three parameters, all of which are optional:

- *`retries`* specifies the number of times to retry the connection in the event of failure. The default value (`0`) means that no additional attempts to connect will be made in the event of failure; a negative value for *`retries`* results in the connection attempt being repeated indefinitely.
- The *`delay`* represents the number of seconds between reconnect attempts; the default is `1` second.
- *`verbose`* indicates whether the method should output a report of its progress, with `1` causing this reporting to be enabled; the default is `0` (reporting disabled).

Return Value. This method returns an `int`, which can have one of the following 3 values:

- **0**: The connection attempt was successful.
- **1**: Indicates a recoverable error.
- **-1**: Indicates an unrecoverable error.

3.2.1.3. `Ndb_cluster_connection::wait_until_ready()`

Description. This method waits until the requested connection with one or more data nodes is successful.

Signature.

```
int wait_until_ready
    (
      int timeoutBefore,
      int timeoutAfter
    )
```

Parameters. This method takes two parameters:

- *`timeoutBefore`* determines the number of seconds to wait until the first “live” node is detected. If this amount of time is exceeded with no live nodes detected, then the method immediately returns a negative value.
- *`timeoutAfter`* determines the number of seconds to wait after the first “live” node is detected for all nodes to become active. If this amount of time is exceeded without all nodes becoming active,

then the method immediately returns a value greater than zero.

If this method returns `0`, then all nodes are “live”.

Return Value. `wait_until_ready()` returns an `int`, whose value is interpreted as follows:

- `= 0`: All nodes are “live”.
- `> 0`: At least one node is “live” (however, it is not known whether *all* nodes are “live”).
- `< 0`: An error occurred.

3.2.1.4. `Ndb_cluster_connection::set_optimized_node_selection()`

Description. This method can be used to override the `connect()` method's default behaviour as regards which node should be connected to first.

Signature.

```
void set_optimized_node_selection
    (
      int value
    )
```

Parameters. An integer *`value`*.

Return Value. *None*.

3.3. The `NdbBlob` Class

This class represents a handle to a `BLOB` column and provides read and write access to `BLOB` column values. This object has a number of different states and provides several modes of access to `BLOB` data; these are also described in this section.

Description. An instance of `NdbBlob` is created using the `NdbOperation::getBlobHandle()` method during the operation preparation phase. (See Section 3.6, “The `NdbOperation` Class”.) This object acts as a handle on a `BLOB` column.

`BLOB` Data Storage. `BLOB` data is stored in 2 locations:

- The header and inline bytes are stored in the blob attribute.
- The blob's data segments are stored in a separate table named `NDB$BLOB_`*`tid`*`_cid`, where *`tid`* is the table ID, and *`cid`* is the blob column ID.

The inline and data segment sizes can be set using the appropriate `NdbDictionary::Column()` methods when the table is created. See Section 3.4.2, “The `Column` Class”, for more information about these methods.

Data Access Types. `NdbBlob` supports 3 types of data access:

- In the preparation phase, the `NdbBlob` methods `getValue()` and `setValue()` are used to prepare a read or write of a `BLOB` value of known size.

- Also in the preparation phase, `setActiveHook()` is used to define a routine which is invoked as soon as the handle becomes active.
- In the active phase, `readData()` and `writeData()` are used to read and write `BLOB` values having arbitrary sizes.

These data access types can be applied in combination, provided that they are used in the order given above.

`BLOB` Operations. `BLOB` operations take effect when the next transaction is executed. In some cases, `NdbBlob` is forced to perform implicit execution. To avoid this, you should always operate on complete blob data segments, and use `NdbTransaction::executePendingBlobOps()` to flush reads and writes. There is no penalty for doing this if nothing is pending. It is not necessary to do so following execution (obviously) or after next scan result is obtained. `NdbBlob` also supports reading post- or pre-blob data from events. The handle can be read after the next event on the main table has been retrieved. The data becomes available immediately. (See Section 3.5, "The `NdbEventOperation` Class".)

`BLOB`s and `NdbOperation`s. `NdbOperation` methods acting on `NdbBlob` objects have the following characteristics:

- `NdbOperation::insertTuple()` must use `NdbBlob::setValue()` if the `BLOB` attribute is non-nullable.
- `NdbOperation::readTuple()` used with any lock mode can read but not write blob values.

  When the `LM_CommittedRead` lock mode is used with `readTuple()`, the lock mode is automatically upgraded to `LM_Read` whenever blob attributes are accessed.
- `NdbOperation::updateTuple()` can either overwrite an existing value using `NdbBlob::setValue()`, or update it during the active phase.
- `NdbOperation::writeTuple()` always overwrites blob values, and must use `NdbBlob::setValue()` if the `BLOB` attribute is non-nullable.
- `NdbOperation::deleteTuple()` creates implicit, non-accessible `BLOB` handles.
- A scan with any lock mode can use its blob handles to read blob values but not write them.

  A scan using the `LM_Exclusive` lock mode can update row and blob values using `updateCurrentTuple()`; the operation returned must explicitly create its own blob handle.

  A scan using the `LM_Exclusive` lock mode can delete row values (and therefore blob values) using `deleteCurrentTuple()`; this create implicit non-accessible blob handles.
- An operation which is returned by `lockCurrentTuple()` cannot update blob values.

See Section 3.6, "The `NdbOperation` Class".

Known Issues. The following are known issues or limitations encountered when working with `NdbBlob` objects:

- Too many pending `BLOB` operations can overflow the I/O buffers.
- The table and its `BLOB` data segment tables are not created atomically.

Public Types. The public types defined by `NdbBlob` are shown here:

Type	Purpose / Use
`ActiveHook`	Callback for `NdbBlob::setActiveHook()`
`State`	Represents the states that may be assumed by the `NdbBlob`.

For a discussion of each of these types, along with its possible values, see Section 3.3.1, "`NdbBlob` Types".

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

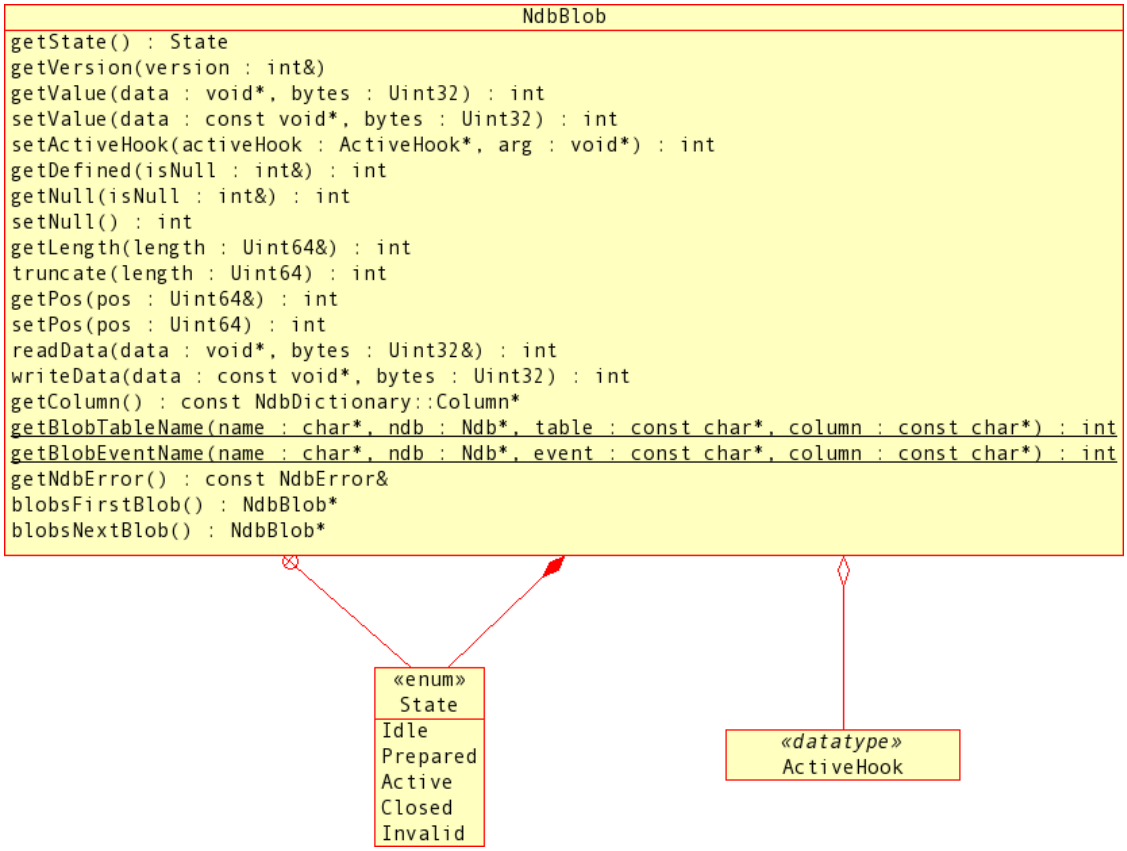
Method	Purpose / Use
`getState()`	Gets the state of an `NdbBlob` object
`getValue()`	Prepares to read a blob value
`setValue()`	Prepares to insert or update a blob value
`setActiveHook()`	Defines a callback for blob handle activation
`getVersion()`	Checks whether a blob is statement-based or event-based
`getNull()`	Checks whether a blob value is `NULL`
`setNull()`	Sets a blob to `NULL`
`getLength()`	Gets the length of a blob, in bytes
`truncate()`	Truncates a bloc to a given length
`getPos()`	Gets the current position for reading/writing
`setPos()`	Sets the position at which to begin reading/writing
`readData()`	Reads data from a blob
`writeData()`	Writes blob data
`getColumn()`	Gets a blob column.
`getNdbError()`	Gets an error (an `NdbError` object)
`blobsFirstBlob()`	Gets the first blob in a list.
`blobsNextBlob()`	Gets the next blob in a list
`getBlobEventName()`	Gets a blob event name
`getBlobTableName()`	Gets a blob data segment's table name.

> **Note**
>
> `blobsFirstBlob()` and `blobsNextBlob()` are static methods.

> **Tip**
>
> Most `NdbBlob` methods (nearly all of those whose return type is `int`) return `0` on success and `-1` in the event of failure.

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.3.2, "`NdbBlob` Class Methods".

Class Diagram. This diagram shows all the available methods and types of the `NdbBlob` class:



3.3.1. `NdbBlob` Types

This section details the public types belonging to the `NdbBlob` class.

3.3.1.1. The `NdbBlob::ActiveHook` Type

ActiveHook is a datatype defined for use as a callback for the `setActiveHook()` method. (See Section 3.3.2.4, "`NdbBlob::setActiveHook()`".)

Definition. `ActiveHook` is a custom datatype defined as shown here:

```
typedef int ActiveHook
    (
      NdbBlob* me,
      void*    arg
    )
```

Description. This is a callback for `NdbBlob::setActiveHook()`, and is invoked immediately once the prepared operation has been executed (but not committed). Any calls to `getValue()` or `setValue()` are performed first. The `BLOB` handle is active so `readData()` or `writeData()` can be used to manipulate the `BLOB` value. A user-defined argument is passed along with the `NdbBlob`. `ActiveHook()` returns a nonzero value in the event of an error.

3.3.1.2. The `NdbBlob::State` Type

This is an enumerated datatype which represents the possible states of an `NdbBlob` instance.

Description. An `NdbBlob` may assume any one of these states

Enumeration Values.

Value	Description
`Prepared`	this is the state of the `NdbBlob` prior to operation execution.
`Active`	This is the `BLOB` handle's state following execution or the fetching of the next result, but before the transaction is committed.
`Closed`	This state occurs after the transaction has been committed.
`Invalid`	This follows a rollback or the close of a transaction.

3.3.2. `NdbBlob` Class Methods

This section discusses the public methods available in the `NdbBlob` class.

> **Important**
>
> This class has no public constructor. You can obtain a blob handle using `NdbEventOperation::getBlobHandle()`. (See Section 3.5, "The `NdbEventOperation` Class".)

3.3.2.1. `NdbBlob::getState()`

Description. This method gets the current state of the `NdbBlob` object for which it is invoked. Possible states are described in Section 3.3.1.2, "The `NdbBlob::State` Type".

Signature.

```
State getState
    (
      void
    )
```

Parameters. None.

Return Value. A `State` value. For possible values, see Section 3.3.1.2, "The `NdbBlob::State` Type".

3.3.2.2. `NdbBlob::getValue()`

Description. Use this method to prepare to read a blob value; the value is available following invocation. Use `getNull()` to check for a `NULL` value; use `getLength()` to get the actual length of the blob, and to check for truncation. `getValue()` sets the current read/write position to the point following the end of the data which was read.

Signature.

```
int getValue
    (
      void*  data,
      Uint32 bytes
    )
```

Parameters. This method takes two parameters — a pointer to the *`data`* to be read, and the number of *`bytes`* to be read.

Return Value. `0` on success, `-1` on failure.

3.3.2.3. NdbBlob::setValue()

Description. This method is used to prepare for inserting or updating a blob value. Any existing blob data that is longer than the new data is truncated. The data buffer must remain valid until the operation has been executed. `setValue()` sets the current read/write position to the point following the end of the data. You can set *`data`* to a null pointer (`0`) in order to create a `NULL` value.

Signature.

```
int setValue
    (
      const void*  data,
      Uint32       bytes
    )
```

Parameters. This method takes two parameters:

- The *`data`* that is to be inserted or used to overwrite the blob value.
- The number of *`bytes`* — that is, the length — of the *`data`*.

Return Value. `0` on success, `-1` on failure.

3.3.2.4. NdbBlob::setActiveHook()

Description. This method defines a callback for blob handle activation. The queue of prepared operations will be executed in no-commit mode up to this point; then, the callback is invoked. For additional information, see Section 3.3.1.1, "The NdbBlob::ActiveHook Type".

Signature.

```
int setActiveHook
    (
      ActiveHook*  activeHook,
      void*        arg
    )
```

Parameters. This method requires two parameters:

- A pointer to an `ActiveHook` value; this is a callback as explained in Section 3.3.1.1, "The NdbBlob::ActiveHook Type".
- A pointer to `void`, for any data to be passed to the callback.

Return Value. `0` on success, `-1` on failure.

3.3.2.5. NdbBlob::getVersion()

Description. This method is used to distinguish whether a blob operation is statement-based or event-based.

Signature.

```
void getVersion
    (
      int& version
    )
```

Parameters. This method takes a single parameter, an integer reference to the blob version (operation type).

Return Value. One of the following three values:

- `-1`: This is a “normal” (statement-based) blob.
- `0`: This is an event-operation based blob, following a change in its data.
- `1`: This is an event-operation based blob, prior to any change in its data.

> **Note**
>
> `getVersion()` is always successful, assuming that it is invoked as a method of a valid `NdbBlob` instance.

3.3.2.6. NdbBlob::getNull()

Description. This method checks whether the blob's value is `NULL`.

Signature.

```
int getNull
    (
      int& isNull
    )
```

Parameters. A reference to an integer *`isNull`*. Following invocation, this parameter has one of the following values, interpreted as shown here:

- `-1`: The blob is undefined. If this is a non-event blob, this result causes a state error.
- `0`: The blob has a non-null value.
- `1`: The blob's value is `NULL`.

Return Value. *None*.

3.3.2.7. NdbBlob::setNull()

Description. This method sets the value of a blob to `NULL`.

Signature.

```
int setNull
    (
      void
    )
```

Parameters. None.

Return Value. `0` on success; `-1` on failure.

3.3.2.8. NdbBlob::getLength()

Description. This method gets the blob's current length in bytes.

Signature.

```
int getLength
    (
      Uint64& length
    )
```

Parameters. A reference to the length.

Return Value. The blob's length in bytes. For a `NULL` blob, this method returns `0`. to distinguish between a blob whose length is `0` blob and one which is `NULL`, use the `getNull()` method.

3.3.2.9. NdbBlob::truncate()

Description. This method is used to truncate a blob to a given length.

Signature.

```
int truncate
    (
      Uint64 length = 0
    )
```

Parameters. `truncate()` takes a single parameter which specifies the new *`length`* to which the blob is to be truncated. This method has no effect if *`length`* is greater than the blob's current length (which you can check using `getLength()`).

Return Value. `0` on success, `-1` on failure.

3.3.2.10. NdbBlob::getPos()

Description. This method gets the current read/write position in a blob.

Signature.

```
int getPos
    (
      Uint64& pos
    )
```

Parameters. One parameter, a reference to the position.

Return Value. Returns `0` on success, or `-1` on failure. (Following a successful invocation, *`pos`* will hold the current read/write position within the blob, as a number of bytes from the beginning.)

3.3.2.11. NdbBlob::setPos()

Description. This method sets the position within the blob at which to read or write data.

Signature.

```
int setPos
    (
      Uint64 pos
    )
```

Parameters. The setPos() method takes a single parameter *`pos`* (an unsigned 64-bit integer), which is the position for reading or writing data. The value of *`pos`* must be between `0` and the blob's current

length.

> **Important**
>
> “Sparse” blobs are not supported in the NDB API; there can be no unused data positions within a blob.

Return Value. `0` on success, `-1` on failure.

3.3.2.12. NdbBlob::readData()

Description. This method is used to read data from a blob.

Signature.

```
int readData
    (
      void*     data,
      Uint32&   bytes
    )
```

Parameters. `readData()` accepts a pointer to the data to be read, and a reference to the number of bytes read.

Return Value. `0` on success, `-1` on failure. Following a successful invocation, *`data`* points to the data that was read, and *`bytes`* holds the number of bytes read.

3.3.2.13. NdbBlob::writeData()

Description. This method is used to write data to an `NdbBlob`. After a successful invocation, the read/write position will be at the first byte following the data that was written to the blob.

> **Note**
>
> A write past the current end of the blob data extends the blob automatically.

Signature.

```
int writeData
    (
      const void*  data,
      Uint32       bytes
    )
```

Parameters. This method takes two parameters, a pointer to the *`data`* to be written, and the number of *`bytes`* to write.

Return Value. `0` on success, `-1` on failure.

3.3.2.14. NdbBlob::getColumn()

Description. Use this method to get the `BLOB` column to which the `NdbBlob` belongs.

Signature.

```
const Column* getColumn
    (
      void
    )
```

Parameters. None.

Return Value. A Column object. (See Section 3.4.2, “The Column Class”.)

3.3.2.15. NdbBlob::getNdbError()

Description. Use this method to obtain an error object. The error may be blob-specific or may be copied from a failed implicit operation. The error code is copied back to the operation unless the operation already has a non-zero error code.

Signature.

```
const NdbError& getNdbError
    (
      void
    ) const
```

Parameters. None.

Return Value. An NdbError object. See Section 4.1, “The NdbError Structure”.

3.3.2.16. NdbBlob::blobsFirstBlob()

Description. This method initialises a list of blobs belonging to the current operation and returns the first blob in the list.

Signature.

```
NdbBlob* blobsFirstBlob
    (
      void
    )
```

Parameters. None.

Return Value. A pointer to the desired blob.

3.3.2.17. NdbBlob::blobsNextBlob()

Description. Use the method to obtain the next in a list of blobs that was initialised using blobsFirstBlob(). See Section 3.3.2.16, “NdbBlob::blobsFirstBlob()”.

Signature.

```
NdbBlob* blobsNextBlob
    (
      void
    )
```

Parameters. None.

Return Value. A pointer to the desired blob.

3.3.2.18. NdbBlob::getBlobEventName()

Description. This method gets a blob event name. The blob event is created if the main event monitors the blob column. The name includes the main event name.

Signature.

```
static int getBlobEventName
    (
```

```
      char*       name,
      Ndb*        ndb,
      const char* event,
      const char* column
    )
```

Parameters. This method takes 4 parameters:

- *name*: The name of the blob event.
- *ndb*: The relevant Ndb object.
- *event*: The name of the main event.
- *column*: The blob column.

Return Value. 0 on success, -1 on failure.

3.3.2.19. NdbBlob::getBlobTabletName()

Description. This method gets the blob data segment table name.

> **Note**
>
> This method is generally of use only for testing and debugging purposes.

Signature.

```
static int getBlobTableName
    (
      char*       name,
      Ndb*        ndb,
      const char* table,
      const char* column
    )
```

Parameters. This method takes 4 parameters:

- *name*: The name of the blob data segment table.
- *ndb*: The relevant Ndb object.
- *table*: The name of the main table.
- *column*: The blob column.

Return Value. 0 on success, -1 on failure.

3.4. The NdbDictionary Class

This class provides meta-information about database objects, such as tables, columns, and indexes.

While the preferred method of database object creation and deletion is through the MySQL Server, NdbDictionary also permits the developer to perform these tasks via the NDB API.

Description. This is a data dictionary class that supports enquiries about tables, columns, and indexes.

It also provides ways to define these database objects and to remove them. Both sorts of functionality are supplied via inner classes that model these objects. These include the following:

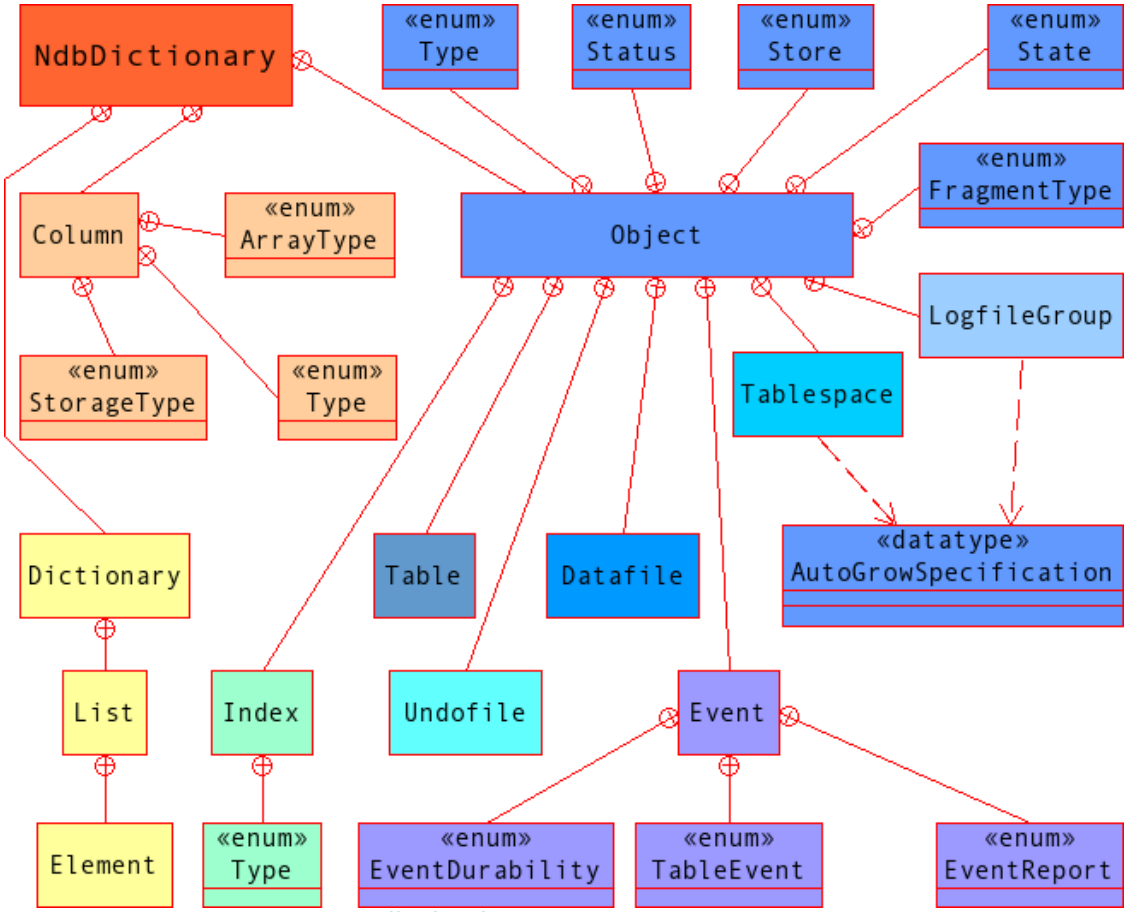
- `NdbDictionary::Object::Table` for working with tables
- `NdbDictionary::Column` for creating table columns
- `NdbDictionary::Object::Index` for working with secondary indexes
- `NdbDictionary::Dictionary` for creating database objects and making schema enquiries
- `NdbDictionary::Object::Event` for working with events in the cluster.

Additional `NdbDictionary::Object` subclasses model the tablespaces, logfile groups, datafiles, and undofiles required for working with Cluster Disk Data tables introduced in MySQL 5.1.

> **Warning**
>
> Tables and indexes created using `NdbDictionary` cannot be viewed from the MySQL Server.
>
> Dropping indexes via the `NDB` API that were created originally from a MySQL Cluster causes inconsistencies. It is possible that a table from which one or more indexes have been dropped using the `NDB` API will no longer be usable by MySQL following such operations. In this event, the table must be dropped, and then re-created using MySQL to make it accessible to MySQL once more.

Public Methods. `NdbDictionary` itself has no public methods. All work is accomplished by accessing its subclasses and their public members.

`NdbDictionary` Subclass Hierarchy. This diagram shows the hierarchy made up of the `NdbDictionary` class, its subclasses, and their enumerated datatypes:



The next several sections discuss `NdbDictionary`'s subclasses and their public members in detail. The organisation of these sections reflects that of the `NdbDictionary` class hierarchy.

> **Note**
>
> For the numeric equivalents to enumerations of NdbDictionary subclasses, see the file `/storage/ndb/include/ndbapi/NdbDictionary.hpp` in the MySQL 5.1 source tree.

3.4.1. The `Dictionary` Class

This section describes the `Dictionary` class and its subclasses `List` and `Element`.

Description. This is used for defining and retrieving data object metadata. It also includes methods for creating and dropping database objects.

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Dictionary()`	Class constructor method
`~Dictionary()`	Destructor method
`getTable()`	Gets the table having the given name
`getIndex()`	Gets the index having the given name
`getEvent()`	Gets the event having the given name

Method	Purpose / Use
getTablespace()	Gets the tablespace having the given name
getLogfileGroup()	Gets the logfile group having the given name
getDatafile()	Gets the datafile having the given name
getUndofile()	Gets the undofile having the given name
getNdbError()	Retrieves the latest error
createTable()	Creates a table
createIndex()	Creates an index
createEvent()	Creates an event
createTablespace()	Creates a tablespace
createLogfileGroup()	Creates a logfile group
createDatafile()	Creates a datafile
createUndofile()	Creates an undofile
dropTable()	Drops a table
dropIndex()	
dropEvent()	Drops an index
dropTablespace()	Drops a tablespace
dropLogfileGroup()	Drops a logfile group
dropDatafile()	Drops a datafile
dropUndofile()	Drops an undofile
listObjects()	Fetches a list of the objects in the dictionary
listIndexes()	Fetches a list of the indexes defined on a given table
removeCachedTable()	Removes a table from the local cache
removeCachedIndex()	Removes an index from the local cache

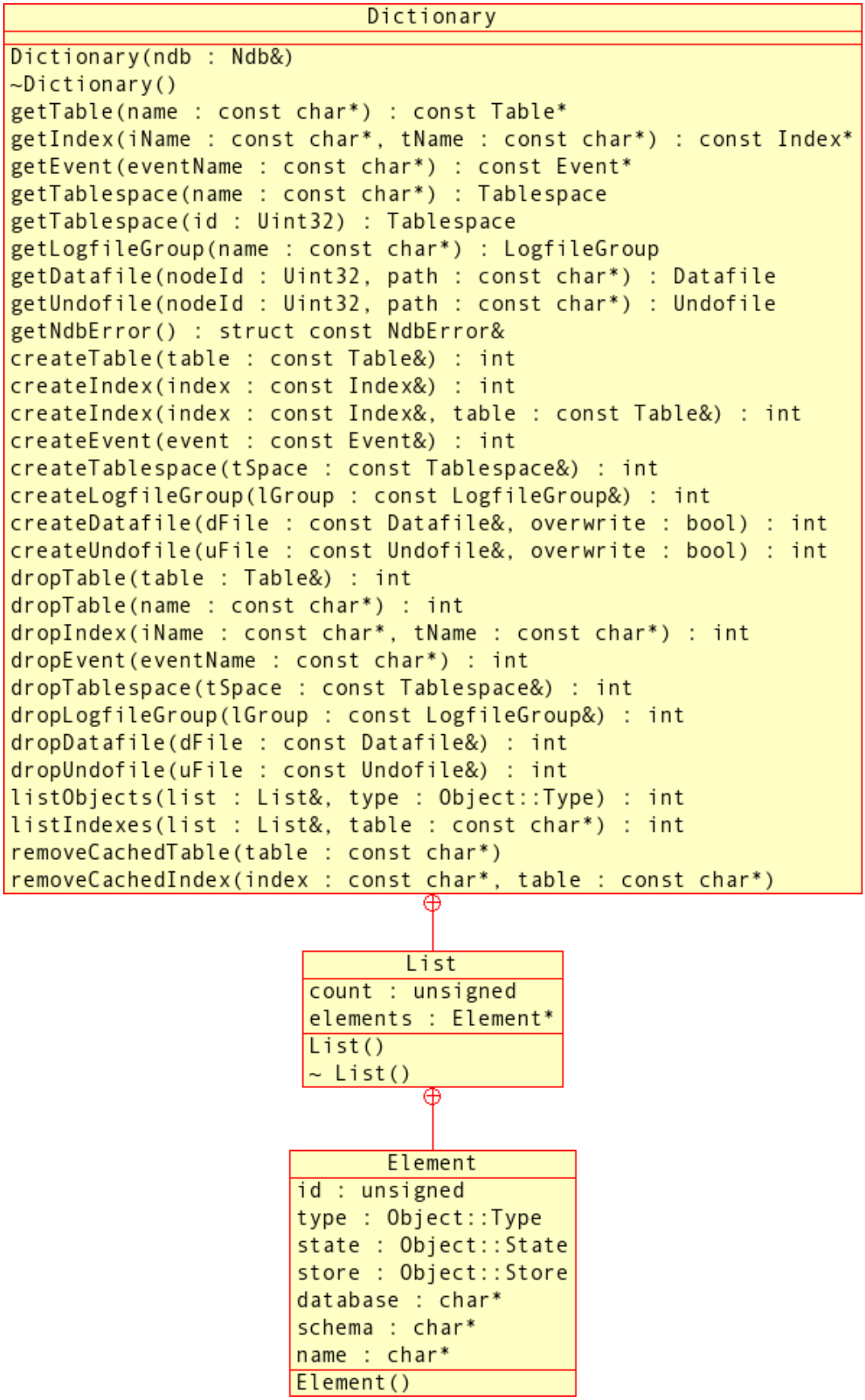
For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.4.1.1, "Dictionary Class Methods".

> **Important**
>
> Objects created using the Dictionary::create*() methods are not visible from the MySQL Server. For this reason, it is usually preferable to avoid using them.

> **Note**
>
> The Dictionary class does not have any methods for working directly with columns. You must use Column class methods for this purpose — see Section 3.4.2, "The Column Class", for details.

Public Types. See Section 3.4.1.2, "The List Class", and Section 3.4.1.2.1, "The Element Structure".

Dictionary Class and Subclass Diagram. This diagram shows all the public members of the Dictionary class and its subclasses:


Dictionary
Dictionary(ndb : Ndb&)
~Dictionary()
getTable(name : const char*) : const Table*
getIndex(iName : const char*, tName : const char*) : const Index*
getEvent(eventName : const char*) : const Event*
getTablespace(name : const char*) : Tablespace
getTablespace(id : Uint32) : Tablespace
getLogfileGroup(name : const char*) : LogfileGroup
getDatafile(nodeId : Uint32, path : const char*) : Datafile
getUndofile(nodeId : Uint32, path : const char*) : Undofile
getNdbError() : struct const NdbError&
createTable(table : const Table&) : int
createIndex(index : const Index&) : int
createIndex(index : const Index&, table : const Table&) : int
createEvent(event : const Event&) : int
createTablespace(tSpace : const Tablespace&) : int
createLogfileGroup(lGroup : const LogfileGroup&) : int
createDatafile(dFile : const Datafile&, overwrite : bool) : int
createUndofile(uFile : const Undofile&, overwrite : bool) : int
dropTable(table : Table&) : int
dropTable(name : const char*) : int
dropIndex(iName : const char*, tName : const char*) : int
dropEvent(eventName : const char*) : int
dropTablespace(tSpace : const Tablespace&) : int
dropLogfileGroup(lGroup : const LogfileGroup&) : int
dropDatafile(dFile : const Datafile&) : int
dropUndofile(uFile : const Undofile&) : int
listObjects(list : List&, type : Object::Type) : int
listIndexes(list : List&, table : const char*) : int
removeCachedTable(table : const char*)
removeCachedIndex(index : const char*, table : const char*)
List
count : unsigned
elements : Element*
List()
~ List()
Element
id : unsigned
type : Object::Type
state : Object::State
store : Object::Store
database : char*
schema : char*
name : char*
Element()

3.4.1.1. `Dictionary` Class Methods

This section details all of the public methods of the `Dictionary` class.

3.4.1.1.1. `Dictionary` Class Constructor

Description. This method creates a new instance of the `Dictionary` class.

> **Note**
>
> Both the constructor and destructor for this class are protected methods, rather than public.

Signature.

```
protected Dictionary
    (
      Ndb& ndb
    )
```

Parameters. An `Ndb` object. See Section 3.1, "The `Ndb` Class".

Return Value. A `Dictionary` object.

Destructor. The destructor takes no parameters and returns nothing.

```
protected ~Dictionary
    (
      void
    )
```

3.4.1.1.2. `Dictionary::getTable()`

Description. This method can be used to access the table with a known name. See Section 3.4.3.7, "The `Table` Class".

Signature.

```
const Table* getTable
    (
      const char* name
    ) const
```

Parameters. The *`name`* of the table.

Return Value. A pointer to the table, or `NULL` if there is no table with the *`name`* supplied.

3.4.1.1.3. `Dictionary::getIndex()`

Description. This method retrieves a pointer to an index, given the name of the index and the name of the table to which the table belongs.

Signature.

```
const Index* getIndex
    (
      const char* iName,
      const char* tName
    ) const
```

Parameters. Two parameters are required:

- The name of the index (*iName*)
- The name of the table to which the index belongs (*tName*)

Both are string values, represented by character pointers.

Return Value. A pointer to an Index. See Section 3.4.3.5, "The Index Class", for information about this object.

3.4.1.1.4. Dictionary::getEvent()

Description. This method is used to obtain an Event object, given the event's name.

Signature.

```
const Event* getEvent
    (
      const char* eventName
    )
```

Parameters. The *eventName*, a string (character pointer).

Return Value. A pointer to an Event object. See Section 3.4.3.4, "The Event Class", for more information.

3.4.1.1.5. Dictionary::getTablespace()

Description. Given either the name or ID of a tablespace, this method returns the corresponding Tablespace object.

Signatures. Using the tablespace name:

```
Tablespace getTablespace
    (
      const char* name
    )
```

Using the tablespace ID:

```
Tablespace getTablespace
    (
      Uint32 id
    )
```

Parameters. Either one of the following:

- The *name* of the tablespace, a string (as a character pointer)
- The unsigned 32-bit integer *id* of the tablespace

Return Value. A Tablespace object, as discussed in Section 3.4.3.8, "The Tablespace Class".

3.4.1.1.6. Dictionary::getLogfileGroup()

Description. This method gets a LogfileGroup object, given the name of the logfile group.

Signature.

```
LogfileGroup getLogfileGroup
    (
```

```
      const char* name
    )
```

Parameters. The *`name`* of the logfile group.

Return Value. An instance of `LogfileGroup`; see Section 3.4.3.6, "The `LogfileGroup` Class", for more information.

3.4.1.1.7. `Dictionary::getDatafile()`

Description. This method is used to retrieve a `Datafile` object, given the node ID of the data node where a datafile is located and the path to the datafile on that node's filesystem.

Signature.

```
Datafile getDatafile
    (
      Uint32      nodeId,
      const char* path
    )
```

Parameters. This method must be invoked using two arguments, as shown here:

- The 32-bit unsigned integer *`nodeId`* of the data node where the datafile is located
- The *`path`* to the datafile on the node's filesystem (string as character pointer)

Return Value. A `Datafile` object — see Section 3.4.3.3, "The `Datafile` Class", for details.

3.4.1.1.8. `Dictionary::getUndofile()`

Description. This method gets an `Undofile` object, given the ID of the node where an undofile is located and the filesystem path to the file.

Signature.

```
Undofile getUndofile
    (
      Uint32      nodeId,
      const char* path
    )
```

Parameters. This method requires the following two arguments:

- The *`nodeId`* of the data node where the undofile is located; this value is passed as a 32-bit unsigned integer
- The *`path`* to the undofile on the node's filesystem (string as character pointer)

Return Value. An instance of `Undofile`. For more information, see Section 3.4.3.9, "The `Undofile` Class".

3.4.1.1.9. `Dictionary::getNdbError()`

Description. This method retrieves the most recent `NDB` API error.

Signature.

```
const struct NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A reference to an `NdbError` object. See Section 4.1, "The `NdbError` Structure".

3.4.1.1.10. `Dictionary::createTable()`

Description. Creates a table given an instance of `Table`.

Signature.

```
int createTable
    (
      const Table& table
    )
```

Parameters. An instance of `Table`. See Section 3.4.3.7, "The `Table` Class", for more information.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.11. `Dictionary::createIndex()`

Description. This method creates an index given an instance of `Index` and possibly an optional instance of `Table`.

Signature.

```
int createIndex
    (
      const Index& index
    )
```

```
int createIndex
    (
      const Index& index,
      const Table& table
    )
```

Parameters. *Required*: A reference to an `Index` object. *Optional*: A reference to a `Table` object.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.12. `Dictionary::createEvent()`

Description. Creates an event, given a reference to an `Event` object.

Signature.

```
int createEvent
    (
      const Event& event
    )
```

Parameters. A reference *`event`* to an `Event` object.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.13. `Dictionary::createTablespace()`

Description. This method creates a new tablespace, given a `Tablespace` object.

Signature.

```
int createTablespace
    (
      const Tablespace& tSpace
    )
```

Parameters. This method requires a single argument — a reference to an instance of `Tablespace`.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.14. `Dictionary::createLogfileGroup()`

Description. This method creates a new logfile group, given an instance of `LogfileGroup`.

Signature.

```
int createLogfileGroup
    (
      const LogfileGroup& lGroup
    )
```

Parameters. A single argument, a reference to a `LogfileGroup` object, is required.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.15. `Dictionary::createDatafile()`

Description. This method creates a new datafile, given a `Datafile` object.

Signature.

```
int createDatafile
    (
      const Datafile& dFile
    )
```

Parameters. A single argument — a reference to an instance of `Datafile` — is required.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.16. `Dictionary::createUndofile()`

Description. This method creates a new undofile, given an `Undofile` object.

Signature.

```
int createUndofile
    (
      const Undofile& uFile
    )
```

Parameters. This method requires one argument: a reference to an instance of `Undofile`.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.17. `Dictionary::dropTable()`

Description. Drops a table given an instance of `Table`.

Signature.

```
int dropTable
    (
      const Table& table
    )
```

Parameters. An instance of `Table`. See Section 3.4.3.7, "The `Table` Class", for more information.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.18. `Dictionary::dropIndex()`

Description. This method drops an index given an instance of `Index` and possibly an optional instance of `Table`.

Signature.

```
int dropIndex
    (
      const Index& index
    )
```

```
int dropIndex
    (
      const Index& index,
      const Table& table
    )
```

Parameters. *Required*: A reference to an `Index` object. *Optional*: A reference to a `Table` object.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.19. `Dictionary::dropEvent()`

Description. This method drops an event, given a reference to an `Event` object.

Signature.

```
int dropEvent
    (
      const Event& event
    )
```

Parameters. A reference *`event`* to an `Event` object.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.20. `Dictionary::dropTablespace()`

Description. This method drops a tablespace, given a `Tablespace` object.

Signature.

```
int dropTablespace
```

```
    (
      const Tablespace& tSpace
    )
```

Parameters. This method requires a single argument — a reference to an instance of `Tablespace`.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.21. `Dictionary::dropLogfileGroup()`

Description. This method drops a logfile group, given an instance of `LogfileGroup`.

Signature.

```
int dropLogfileGroup
    (
      const LogfileGroup& lGroup
    )
```

Parameters. A single argument, a reference to a `LogfileGroup` object, is required.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.22. `Dictionary::dropDatafile()`

Description. This method drops a datafile, given a `Datafile` object.

Signature.

```
int dropDatafile
    (
      const Datafile& dFile
    )
```

Parameters. A single argument — a reference to an instance of `Datafile` — is required.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.23. `Dictionary::dropUndofile()`

Description. This method drops an undofile, given an `Undofile` object.

Signature.

```
int dropUndofile
    (
      const Undofile& uFile
    )
```

Parameters. This method requires one argument: a reference to an instance of `Undofile`.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.24. `Dictionary::listObjects()`

Description. This method is used to obtain a list of objects in the dictionary. It is possible to get all of the objects in the dictionary, or to restrict the list to objects of a single type.

Signatures.

```
int listObjects
    (
      List&       list,
      Object::Type type = Object::TypeUndefined
    ) const
```

or

```
int listObjects
    (
      List&       list,
      Object::Type type = Object::TypeUndefined
    )
```

Parameters. A reference to a `List` object is required — this is the list that contains the dictionary's objects after `listObjects()` is called. (See Section 3.4.1.2, “The `List` Class”.) An optional second argument *`type`* may be used to restrict the list to only those objects of the given type — that is, of the specified `Object::Type`. (See Section 3.4.3.1.5, “The `Object::Type` Type”.)

Return Value. `0` on success, `-1` on failure.

3.4.1.1.25. `Dictionary::listIndexes()`

Description. This method is used to obtain a `List` of all the indexes on a table, given the table's name. (See Section 3.4.1.2, “The `List` Class”.)

Signature.

```
int listIndexes
    (
      List&      list,
      const char* table
) const
```

or

```
int listIndexes
    (
      List&      list,
      const char* table
    )
```

Parameters. `listIndexes()` takes two arguments:

- A reference to the `List` that contains the indexes following the call to the method
- The name of the *`table`* whose indexes are to be listed

Both of these arguments are required.

Return Value. `0` on success, `-1` on failure.

3.4.1.1.26. `Dictionary::removeCachedTable()`

Description. This method removes the specified table from the local cache.

Signature.

```
void removeCachedTable
    (
      const char* table
    )
```

Parameters. The name of the *`table`* to be removed from the cache.

Return Value. *None*.

3.4.1.1.27. `Dictionary::removeCachedIndex()`

Description. This method removes the specified index from the local cache.

Signature.

```
void removeCachedIndex
    (
      const char* index,
      const char* table
    )
```

Parameters. The `removeCachedIndex()` requires two arguments:

- The name of the *`index`* to be removed from the cache
- The name of the *`table`* in which the index is found

Return Value. *None*.

3.4.1.2. The `List` Class

This section covers the `List` class, a subclass of `NdbDictionary::Dictionary`.

Description. The `List` class is a `Dictionary` subclass that is used for representing lists populated by the methods `Dictionary::listObjects()` and `Dictionary::listIndexes()`. (See Section 3.4.1.1.24, "`Dictionary::listObjects()`", and Section 3.4.1.1.25, "`Dictionary::listIndexes()`".)

Class Methods. This class has only two methods, a constructor and a destructor. Neither method takes any arguments, and the return type of each is `void`.

Constructor. Calling the `List` constructor creates a new `List` whose `count` and `elements` attributes are both set equal to `0`.

Destructor. The destructor `~List()` is simply defined in such a way as to remove all elements and their properties. You can find its definition in the file `/storage/ndb/include/ndbapi/NdbDictionary.hpp`.

Attributes. A `List` has two attributes:

- `count`, an unsigned integer, which stores the number of elements in the list.
- `elements`, a pointer to an array of `Element` data structures contained in the list. See Section 3.4.1.2.1, "The `Element` Structure".

Types. The `List` class defines an `Element` structure, which is described in the following section.

> **Note**
>
> For a graphical representation of this class and its parent-child relationships, see Section 3.4.1, "The `Dictionary` Class".

3.4.1.2.1. The `Element` Structure

This section discusses the structure `NdbDictionary::Dictionary::List::Element`.

Description. The `Element` structure models an element of a list; it is used to store an object in a `List` populated by the methods `Dictionary::listObjects()` and `Dictionary::listIndexes()`.

Attributes. An `Element` has the attributes shown in the following table:

Attribute	Type	Initial Value
`id`	`unsigned int`	`0`
`type`	`Object::Type`	`Object::TypeUndefined`
`state`	`Object::State`	`Object::StateUndefined`
`store`	`Object::Store`	`Object::StoreUndefined`
`database`	`char*`	`0`
`schema`	`char*`	`0`
`name`	`char*`	`0`

> **Note**
>
> For a graphical representation of this class and its parent-child relationships, see Section 3.4.1, "The `Dictionary` Class".

3.4.2. The `Column` Class

This class represents a column in an NDB Cluster table.

Description. Each instance of the `Column` is characterised by its type, which is determined by a number of type specifiers:

- Built-in type
- Array length or maximum length
- Precision and scale (*currently not in use*)
- Character set (applicable only to columns using string datatypes)
- Inline and part sizes (applicable only to `BLOB` columns)

These types in general correspond to MySQL datatypes and their variants. The data formats are same as in MySQL. The `NDB` API provides no support for constructing such formats; however, they are checked by the `NDB` kernel.

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`getName()`	Gets the name of the column
`getNullable()`	Checks whether the column can be set to `NULL`
`getPrimaryKey()`	Check whether the column is part of the table's primary key

Method	Purpose / Use
getColumnNo()	Gets the column number
equal()	Compares Column objects
getType()	Gets the column's type (Type value)
getLength()	Gets the column's length
getCharset()	Get the character set used by a string (text) column (not applicable to columns not storing character data)
getInlineSize()	Gets the inline size of a BLOB column (not applicable to other column types)
getPartSize()	Gets the part size of a BLOB column (not applicable to other column types)
getStripeSize()	Gets a BLOB column's stripe size (not applicable to other column types)
getSize()	Gets the size of an element
getPartitionKey()	Checks whether the column is part of the table's partitioning key
getArrayType()	Gets the column's array type
getStorageType()	Gets the storage type used by this column
getPrecision()	Gets the column's precision (used for decimal types only)
getScale()	Gets the column's scale (used for decimal types only)
Column()	Class constructor; there is also a copy constructor
~Column()	Class destructor
setName()	Sets the column's name
setNullable()	Toggles the column's nullability
setPrimaryKey()	Determines whether the column is part of the primary key
setType()	Sets the column's Type
setLength()	Sets the column's length
setCharset()	Sets the character set used by a column containing character data (not applicable to non-textual columns)
setInlineSize()	Sets the inline size for a BLOB column (not applicable to non-BLOB columns)
setPartSize()	Sets the part size for a BLOB column (not applicable to non-BLOB columns)
setStripeSize()	Sets the stripe size for a BLOB column (not applicable to non-BLOB columns)
setPartitionKey()	Determines whether the column is part of the table's partitioning key
setArrayType()	Sets the column's ArrayType
setStorageType()	Sets the storage type to be used by this column
getPrecision()	Gets the column's precision (used for decimal types only)
getScale()	Gets the column's scale (used for decimal types only)
getPrecision()	Sets the column's precision (used for decimal types only)
getScale()	Sets the column's scale (used for decimal types only)

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.4.2.2,

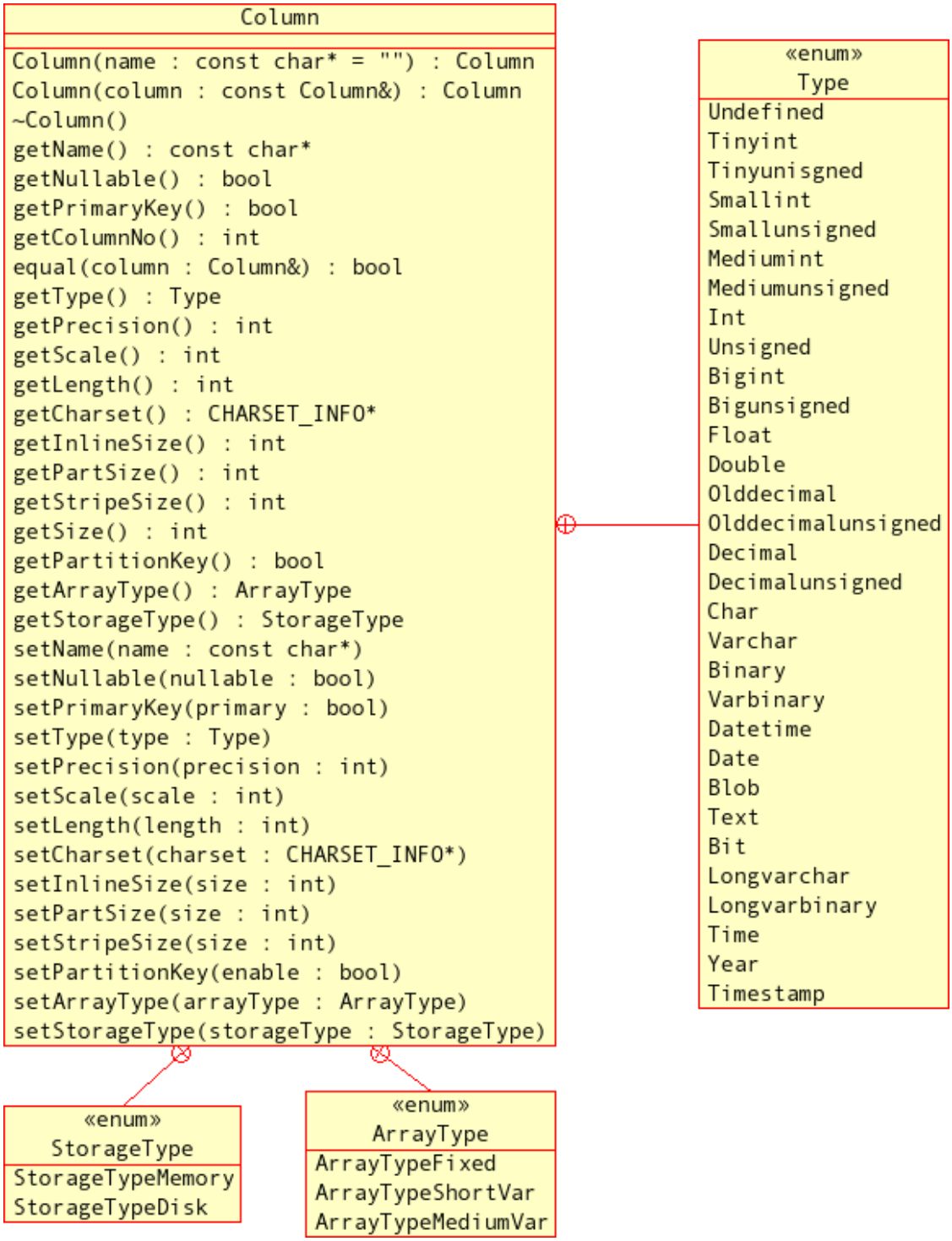
"Column Class Methods".

Public Types. These are the public types of the Column class:

Type	Purpose / Use
ArrayType	Specifies the column's internal storage format
StorageType	Determines whether the column is stored in memory or on disk
Type	The column's datatype. NDB columns have the datatypes as found in MySQL

For a discussion of each of these types, along with its possible values, see Section 3.4.2.1, "Column Types".

Class Diagram. This diagram shows all the available methods and enumerated types of the Column class:



3.4.2.1. `Column` Types

This section details the public types belonging to the `Column` class.

3.4.2.1.1. The `Column::ArrayType` Type

This type describes the `Column`'s internal attribute format.

Description. The attribute storage format can be either fixed or variable.

Enumeration Values.

Value	Description
ArrayTypeFixed	stored as a fixed number of bytes
ArrayTypeShortVar	stored as a variable number of bytes; uses 1 byte overhead
ArrayTypeMediumVar	stored as a variable number of bytes; uses 2 bytes overhead

The fixed storage format is faster but also generally requires more space than the variable format. The default is ArrayTypeShortVar for Var* types and ArrayTypeFixed for others. The default is usually sufficient.

3.4.2.1.2. The Column::StorageType Type

This type describes the storage type used by a Column object.

Description. The storage type used for a given column can be either in memory or on disk. Columns stored on disk mean that less RAM is required overall but such columns cannot be indexed, and are potentially much slower to access. The default is StorageTypeMemory.

Enumeration Values.

Value	Description
StorageTypeMemory	Store the column in memory
StorageTypeDisk	Store the column on disk

3.4.2.1.3. Column::Type

Type is used to describe the Column object's datatype.

Description. Datatypes for Column objects are analogous to the datatypes used by MySQL. The types Tinyint, Tinyintunsigned, Smallint, Smallunsigned, Mediumint, Mediumunsigned, Int, Unsigned, Bigint, Bigunsigned, Float, and Double (that is, types Tinyint through Double in the order listed in the Enumeration Values table) can be used in arrays.

Enumeration Values.

Value	Description
Undefined	Undefined
Tinyint	1-byte signed integer
Tinyunsigned	1-byte unsigned integer
Smallint	2-byte signed integer
Smallunsigned	2-byte unsigned integer
Mediumint	3-byte signed integer
Mediumunsigned	3-byte unsigned integer
Int	4-byte signed integer
Unsigned	4-byte unsigned integer
Bigint	8-byte signed integer
Bigunsigned	8-byte signed integer

Value	Description
Float	4-byte float
Double	8-byte float
Olddecimal	Signed decimal as used prior to MySQL 5.0
Olddecimalunsigned	Unsigned decimal as used prior to MySQL 5.0
Decimal	Signed decimal as used by MySQL 5.0 and later
Decimalunsigned	Unsigned decimal as used by MySQL 5.0 and later
Char	A fixed-length array of 1-byte characters; maximum length is 255 characters
Varchar	A variable-length array of 1-byte characters; maximum length is 255 characters
Binary	A fixed-length array of 1-byte binary characters; maximum length is 255 characters
Varbinary	A variable-length array of 1-byte binary characters; maximum length is 255 characters
Datetime	An 8-byte date and time value, with a precision of 1 second
Date	A 4-byte date value, with a precision of 1 day
Blob	A binary large object; see Section 3.3, "The NdbBlob Class"
Text	A text blob
Bit	A bit value; the length specifies the number of bits
Longvarchar	A 2-byte Varchar
Longvarbinary	A 2-byte Varbinary
Time	Time without date
Year	1-byte year value in the range 1901-2155 (same as MySQL)
Timestamp	Unix time

> **Caution**
>
> Do not confuse Column::Type with Object::Type or Table::Type.

3.4.2.2. Column Class Methods

This section documents the public methods of the Column class.

> **Note**
>
> The assignment (=) operator is overloaded for this class, so that it always performs a deep copy.

> **Warning**
>
> As with other database objects, Column object creation and attribute changes to existing columns done using the NDB API are not visible from MySQL. For example, if you change a column's datatype using Column::setType(), MySQL will regard the type of column as being unchanged. The only exception to this rule with regard to columns is that you can change the name of an existing column using Column::setName().

3.4.2.2.1. Column Constructor

Description. You can create a new `Column` or copy an existing one using the class constructor.

> **Warning**
>
> A `Column` created using the `NDB` API will *not* be visible to a MySQL server.

Signature. You can create either a new instance of the `Column` class, or by copying an existing `Column` object. Both of these are shown here.

- Constructor for a new `Column`:

```
Column
    (
      const char* name = ""
    )
```

- Copy constructor:

```
Column
    (
      const Column& column
    )
```

Parameters. When creating a new instance of `Column`, the constructor takes a single argument, which is the name of the new column to be created. The copy constructor also takes one parameter — in this case, a reference to the `Column` instance to be copied.

Return Value. A `Column` object.

Destructor. The `Column` class destructor takes no arguments and *None*.

Examples.

```
[To be supplied...]
```

3.4.2.2.2. `Column::getName()`

Description. This method returns the name of the column for which it is called.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the column.

3.4.2.2.3. `Column::getNullable()`

Description. This method is used to determine whether the column can be set to `NULL`.

Signature.

```
bool getNullable
    (
      void
```

```
    ) const
```

Parameters. *None*.

Return Value. A Boolean value: `true` if the column can be set to `NULL`, otherwise `false`.

3.4.2.2.4. `Column::getPrimaryKey()`

Description. This method is used to determine whether the column is part of the table's primary key.

Signature.

```
bool getPrimaryKey
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A Boolean value: `true` if the column is part of the primary key of the table to which this column belongs, otherwise `false`.

3.4.2.2.5. `Column::getColumnNo()`

Description. This method gets the number of a column — that is, its horizontal position within the table.

Signature.

```
int getColumnNo
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The column number as an integer.

3.4.2.2.6. `Column::equal()`

Description. This method is used to compare one `Column` with another to determine whether the two `Column` objects are the same.

Signature.

```
bool equal
    (
      const Column& column
    ) const
```

Parameters. `equal()` takes a single parameter, a reference to an instance of `Column`.

Return Value. `true` if the columns being compared are equal, otherwise `false`.

3.4.2.2.7. `Column::getType()`

Description. This method gets the column's datatype.

Signature.

```
Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The `Type` (datatype) of the column. For a list of possible values, see Section 3.4.2.1.3, "`Column::Type`".

3.4.2.2.8. `Column::getPrecision()`

Description. This method gets the precision of a column.

> **Note**
>
> This method is applicable to decimal columns only.

Signature.

```
int getPrecision
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The column's precision, as an integer. The precision is defined as the number of significant digits; for more information, see the discussion of the `DECIMAL` datatype in Numeric Types [http://dev.mysql.com/doc/refman/5.1/en/numeric-types.html], in the MySQL Manual.

3.4.2.2.9. `Column::getScale()`

Description. This method gets the scale used for a decimal column value.

> **Note**
>
> This method is applicable to decimal columns only.

Signature.

```
int getScale
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The decimal column's scale, as an integer. The scale of a decimal column represents the number of digits that can be stored following the decimal point. It is possible for this value to be `0`. For more information, see the discussion of the `DECIMAL` datatype in Numeric Types [http://dev.mysql.com/doc/refman/5.1/en/numeric-types.html], in the MySQL Manual.

3.4.2.2.10. `Column::getLength()`

Description. This method gets the length of a column. This is either the array length for the column or — for a variable length array — the maximum length.

Signature.

```
int getLength
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The (maximum) array length of the column, as an integer.

3.4.2.2.11. Column::getCharset()

Description. This gets the character set used by a text column.

> **Note**
>
> This method is applicable only to columns whose Type value is Char, Varchar, or Text.

Signature.

```
CHARSET_INFO* getCharset
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to a CHARSET_INFO structure specifying both character set and collation. This is the same as a MySQL MY_CHARSET_INFO data structure; for more information, see mysql_get_character_set_info() [http://dev.mysql.com/doc/refman/5.1/en/mysql-get-character-set-info.html],in the MySQL Manual.

3.4.2.2.12. Column::getInlineSize()

Description. This method retrieves the inline size of a blob column — that is, the number of initial bytes to store in the table's blob attribute. This part is normally in main memory and can be indexed.

> **Note**
>
> This method is applicable only to blob columns.

Signature.

```
int getInlineSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The blob column's inline size, as an integer.

3.4.2.2.13. Column::getPartSize()

Description. This method is used to get the part size of a blob column — that is, the number of bytes that are stored in each tuple of the blob table.

> **Note**
>
> This method is applicable to blob columns only.

Signature.

```
int getPartSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The column's part size, as an integer. In the case of a `Tinyblob` column, this value is `0` (that is, only inline bytes are stored).

3.4.2.2.14. `Column::getStripeSize()`

Description. This method gets the stripe size of a blob column — that is, the number of consecutive parts to store in each node group.

Signature.

```
int getStripeSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The column's stripe size, as an integer.

3.4.2.2.15. `Column::getSize()`

Description. This function is used to obtain the size of a column.

Signature.

```
int getSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The column's size in bytes (an integer value).

3.4.2.2.16. `Column::getPartitionKey()`

Description. This method is used to check whether the column is part of the table's partitioning key.

> **Note**
>
> A *partitioning key* is a set of attributes used to distribute the tuples onto the `NDB` nodes. This key a hashing function specific to the `NDBCluster` storage engine.
>
> An example where this would be useful is an inventory tracking application involving multiple warehouses and regions, where it might be good to use the warehouse ID and district id as the partition key. This would place all data for a specific district and warehouse in the same database node. Locally to each fragment the full primary key will still be used with the hashing algorithm in such a case.
>
> For more information about partitioning, partitioning schemes, and partitioning keys in MySQL 5.1, see Partitioning [http://dev.mysql.com/doc/refman/5.1/en/partitioning.html], in

the MySQL Manual.

Important

The only type of user-defined partitioning that is supported for use with the NDBCluster storage engine in MySQL 5.1 is key partitioning.

Signature.

```
bool getPartitionKey
    (
      void
    ) const
```

Parameters. *None*.

Return Value. `true` if the column is part of the partitioning key for the table, otherwise `false`.

3.4.2.2.17. Column::getArrayType()

Description. This method gets the column's array type.

Signature.

```
ArrayType getArrayType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `ArrayType`; see Section 3.4.2.1.1, "The `Column::ArrayType` Type" for possible values.

3.4.2.2.18. Column::getStorageType()

Description. This method obtains a column's storage type.

Signature.

```
StorageType getStorageType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `StorageType` value; for more information about this type, see Section 3.4.2.1.2, "The `Column::StorageType` Type".

3.4.2.2.19. Column::setName()

Description. This method is used to set the name of a column.

Important

`setName()` is the only `Column` method whose result is visible from a MySQL Server. MySQL cannot see any other changes made to existing columns using the NDB API.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. This method takes a single argument — the new name for the column.

Return Value. This method *None*.

3.4.2.2.20. Column::setNullable()

Description. This method allows you to toggle the nullability of a column.

> **Important**
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setNullable
    (
      bool nullable
    )
```

Parameters. A Boolean value. Using `true` makes it possible to insert `NULL`s into the column; if *`nullable`* is `false`, then this method performs the equivalent of changing the column to `NOT NULL` in MySQL.

Return Value. *No return value*.

3.4.2.2.21. Column::setPrimaryKey()

Description. This method is used to make a column part of the table's primary key, or to remove it from the primary key.

> **Important**
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setPrimaryKey
    (
      bool primary
    )
```

Parameters. This method takes a single Boolean value. If it is `true`, then the column becomes part of the table's primary key; if `false`, then the column is removed from the primary key.

Return Value. *No return value*.

3.4.2.2.22. Column::setType()

Description. This method sets the `Type` (datatype) of a column.

> **Important**
>
> `setType()` resets *all* column attributes to their (type dependent) default values; it should be the first method that you call when changing the attributes of a given column.

Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setType
    (
      Type type
    )
```

Parameters. This method takes a single parameter — the new `Column::Type` for the column. The default is `Unsigned`. For a listing of all permitted values, see Section 3.4.2.1.3, "`Column::Type`".

Return Value. *No return value*.

3.4.2.2.23. `Column::setPrecision()`

Description. This method can be used to set the precision of a decimal column.

Important

This method is applicable to decimal columns only.

Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setPrecision
    (
      int precision
    )
```

Parameters. This method takes a single parameter — precision is an integer, the value of the column's new precision. For additional information about decimal precision and scale, see Section 3.4.2.2.8, "`Column::getPrecision()`", and Section 3.4.2.2.9, "`Column::getScale()`".

Return Value. *No return value*.

3.4.2.2.24. `Column::setScale()`

Description. This method can be used to set the scale of a decimal column.

Important

This method is applicable to decimal columns only.

Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setScale
    (
      int scale
    )
```

Parameters. This method takes a single parameter — the integer *`scale`* is the new scale for the decimal column. For additional information about decimal precision and scale, see Section 3.4.2.2.8, "`Column::getPrecision()`", and Section 3.4.2.2.9, "`Column::getScale()`".

Return Value. *No return value*.

3.4.2.2.25. Column::setLength()

Description. This method sets the length of a column. For a variable-length array, this is the maximum length; otherwise it is the array length.

> **Important**
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setLength
    (
      int length
    )
```

Parameters. This method takes a single argument — the integer value *length* is the new length for the column.

Return Value. *No return value*.

3.4.2.2.26. Column::setCharset()

Description. This method can be used to set the character set and collation of a Char, Varchar, or Text column.

> **Important**
>
> This method is applicable to Char, Varchar, and Text columns only.
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setCharset
    (
      CHARSET_INFO* cs
    )
```

Parameters. This method takes one parameter. *cs* is a pointer to a CHARSET_INFO structure. For additional information, see Section 3.4.2.2.11, "Column::getCharset()".

Return Value. *No return value*.

3.4.2.2.27. Column::setInlineSize

Description. This method gets the inline size of a BLOB column — that is, the number of initial bytes to store in the table's blob attribute. This part is normally kept in main memory, and can be indexed and interpreted.

> **Important**
>
> This method is applicable to BLOB columns only.
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setInlineSize
    (
      int size
```

```
    )
```

Parameters. The integer *`size`* is the new inline size for the `BLOB` column.

Return Value. *No return value*.

3.4.2.2.28. `Column::setPartSize()`

Description. This method sets the part size of a `BLOB` column — that is, the number of bytes to store in each tuple of the `BLOB` table.

> **Important**
>
> This method is applicable to `BLOB` columns only.
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setPartSize
    (
      int size
    )
```

Parameters. The integer *`size`* is the number of bytes to store in the `BLOB` table. Using zero for this value allows only inline bytes to be stored, in effect making the column's type `TINYBLOB`.

Return Value. *No return value*.

3.4.2.2.29. `Column::setStripeSize()`

Description. This method sets the stripe size of a `BLOB` column — that is, the number of consecutive parts to store in each node group.

> **Important**
>
> This method is applicable to `BLOB` columns only.
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setStripeSize
    (
      int size
    )
```

Parameters. This method takes a single argument. The integer *`size`* is the new stripe size for the column.

Return Value. *No return value*.

3.4.2.2.30. `Column::setPartitionKey()`

Description. This method makes it possible to add a column to the partitioning key of the table to which it belongs, or to remove the column from the table's partitioning key.

> **Important**
>
> Changes made to columns using this method are not visible to MySQL.

For additional information, see Section 3.4.2.2.16, “`Column::getPartitionKey()`”.

Signature.

```
void setPartitionKey
    (
      bool enable
    )
```

Parameters. The single parameter *`enable`* is a Boolean value. Passing `true` to this method makes the column part of the table's partitioning key; if *`enable`* is `false`, then the column is removed from the partitioning key.

Return Value. *No return value*.

3.4.3. The `Object` Class

This class provides meta-information about database objects such as tables and indexes. `Object` subclasses model these and other database objects.

Public Methods. The following table lists the public methods of the `Object` class and the purpose or use of each method:

Method	Purpose / Use
`getObjectStatus()`	Gets an object's status
`getObjectVersion()`	Gets the version of an object
`getObjectId()`	Gets an object's ID

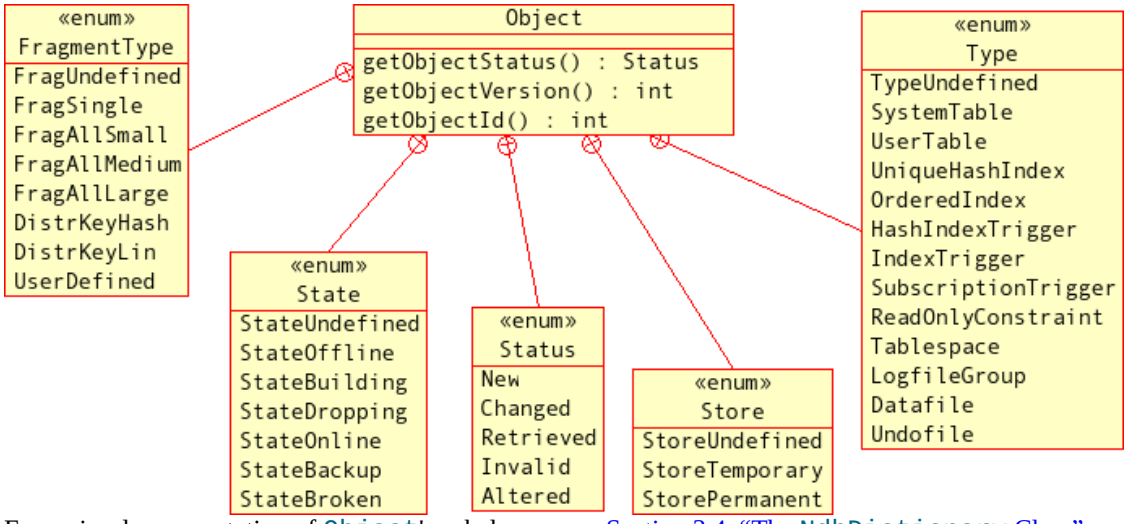
For a detailed discussion of each of these methods, see Section 3.4.3.2, “`Object` Class Methods”.

Public Types. These are the public types of the `Object` class:

Type	Purpose / Use
`FragmentType`	Fragmentation type used by the object (a table or index)
`State`	The object's state (whether it is usable)
`Status`	The object's state (whether it is available)
`Store`	Whether the object has been temporarily or permanently stored
`Type`	The object's type (what sort of table, index, or other database object the `Object` represents)

For a discussion of each of these types, along with its possible values, see Section 3.4.3.1, “`Object` Class Enumerated Types”.

This diagram shows all public members of the `Object` class:



For a visual representation of `Object`'s subclasses, see Section 3.4, "The `NdbDictionary` Class".

3.4.3.1. `Object` Class Enumerated Types

This section details the public enumerated types belonging to the `Object` class.

3.4.3.1.1. The `Object::FragmentType` Type

This type describes the `Object`'s fragmentation type.

Description. This parameter specifies how data in the table or index is distributed among the cluster's storage nodes, that is, the number of fragments per node. The larger the table, the larger the number of fragments that should be used. Note that all replicas count as a single fragment. For a table, the default is `FragAllMedium`. For a unique hash index, the default is taken from the underlying table and cannot currently be changed.

Enumeration Values.

Value	Description
`FragUndefined`	The fragmentation type is undefined or the default
`FragAllMedium`	Two fragments per node
`FragAllLarge`	Four fragments per node

3.4.3.1.2. The `Object::State` Type

This type describes the state of the `Object`.

Description. This parameter provides us with the object's state. By *state*, we mean whether or not the object is defined and is in a usable condition.

Enumeration Values.

Value	Description
`StateUndefined`	Undefined
`StateOffline`	Offline, not useable
`StateBuilding`	Building (e.g. restore?), not useable(?)
`StateDropping`	Going offline or being dropped; not usable

Value	Description
StateOnline	Online, usable
StateBackup	Online, being backed up, usable
StateBroken	Broken; should be dropped and re-created

3.4.3.1.3. The Object::Status Type

This type describes the Object's status.

Description. Reading an object's Status tells whether or not it is available in the NDB kernel.

Enumeration Values.

Value	Description
New	The object exists only in memory, and has not yet been created in the NDB kernel
Changed	The object has been modified in memory, and must be committed in the NDB Kernel for changes to take effect
Retrieved	The object exists, and has been read into main memory from the NDB Kernel
Invalid	The object has been invalidated, and should no longer be used
Altered	The table has been altered in the NDB kernel, but is still available for use

3.4.3.1.4. The Object::Store Type

This type describes the Object's persistence.

Description. Reading this value tells us is the object is temporary or permanent.

Enumeration Values.

Value	Description
StoreUndefined	The object is undefined
StoreTemporary	Temporary storage; the object or data will be deleted on system re-start
StorePermanent	The object or data is permanent; it has been logged to disk

3.4.3.1.5. The Object::Type Type

This type describes the Object's type.

Description. The Type of the object can be one of several different sorts of index, trigger, tablespace, and so on.

Enumeration Values.

Value	Description
TypeUndefined	Undefined

Value	Description
SystemTable	System table
UserTable	User table (may be temporary)
UniqueHashIndex	Unique (but unordered) hash index
OrderedIndex	Ordered (but not unique) index
HashIndexTrigger	Index maintenance (*internal*)
IndexTrigger	Index maintenance (*internal*)
SubscriptionTrigger	Backup or replication (*internal*)
ReadOnlyConstraint	Trigger (*internal*)
Tablespace	Tablespace
LogfileGroup	Logfile group
Datafile	Datafile
Undofile	Undofile

3.4.3.2. `Object` Class Methods

The sections that follow describe each of the public methods of the `Object` class.

> **Important**
>
> All 3 of these methods are pure virtual methods, and are reimplemented in the `Table`, `Index`, and `Event` subclasses where needed. See Section 3.4.3.7, "The `Table` Class", Section 3.4.3.5, "The `Index` Class", and Section 3.4.3.4, "The `Event` Class".

3.4.3.2.1. `Object::getObjectStatus()`

Description. This method retrieves the status of the object for which it is invoked.

Signature.

```
virtual Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The current `Status` of the `Object`. For possible values, see Section 3.4.3.1.3, "The `Object::Status` Type".

3.4.3.2.2. `Object::getObjectVersion()`

Description. The method gets the current version of the object.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object's version number, an integer.

3.4.3.2.3. `Object::getObjectId()`

Description. This method retrieves the object's ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID, an integer.

3.4.3.3. The `Datafile` Class

This section covers the `Datafile` class.

Description. The `Datafile` class models a Cluster Disk Data datafile, which is used to store Disk Data table data.

> **Note**
>
> In MySQL 5.1, only unindexed column data can be stored on disk. Indexes and indexes columns continue to be stored in memory as with previous versions of MySQL Cluster.
>
> Versions of MySQL prior to 5.1 do not support Disk Data storage and so do not support datafiles; thus the `Datafile` class is unavailable for `NDB` API applications written against these MySQL versions.

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

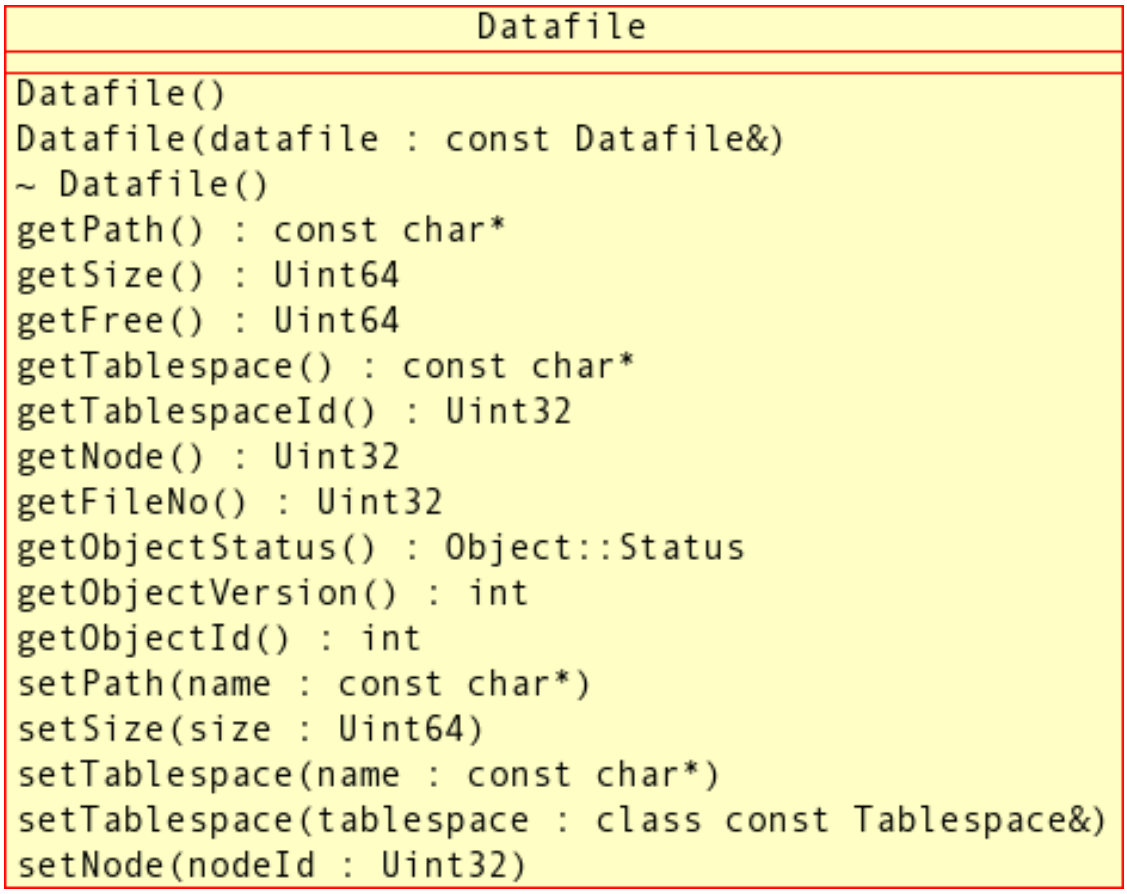
Method	Purpose / Use
`Datafile()`	Class constructor
`~Datafile()`	Destructor
`getPath()`	Gets the filesystem path to the datafile
`getSize()`	Gets the size of the datafile
`getFree()`	Gets the amount of free space in the datafile
`getNode()`	Gets the ID of the node where the datafile is located
`getTablespace()`	Gets the name of the tablespace to which the datafile belongs
`getTablespaceId()`	Gets the ID of the tablespace to which the datafile belongs
`getFileNo()`	Gets the number of the datafile in the table space
`getObjectStatus()`	Gets the datafile's object status
`getObjectVersion()`	Gets the datafile's object version
`getObjectId()`	Gets the datafile's object ID
`setPath()`	Sets the name and location of the datafile on the filesystem
`setSize()`	Sets the datafile's size
`setTablespace()`	Sets the tablespace to which the datafile belongs

Method	Purpose / Use
setNode()	Sets the Cluster node where the datafile is to be located

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.4.3.3.1, "Datafile Class Methods".

Public Types. The Datafile class defines no public types.

Class Diagram. This diagram shows all the available methods of the Datafile class:



3.4.3.3.1. Datafile Class Methods

This section provides descriptions of the public methods of the Datafile class.

3.4.3.3.1.1. Datafile Class Constructor

Description. This method creates a new instance of Datafile, or a copy of an existing one.

Signature. To create a new instance:

```
Datafile
    (
      void
    )
```

To create a copy of an existing Datafile instance:

```
Datafile
```

```
    (
      const Datafile& datafile
    )
```

Parameters. New instance: *None*. Copy constructor: a reference to the `Datafile` instance to be copied.

Return Value. A `Datafile` object.

3.4.3.3.1.2. `Datafile::getPath()`

Description. This method returns the filesystem path to the datafile.

Signature.

```
const char* getPath
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The path to the datafile on the data node's filesystem, a string (character pointer).

3.4.3.3.1.3. `Datafile::getSize()`

Description. This method gets the size of the datafile in bytes.

Signature.

```
Uint64 getSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The size of the data file, in bytes, as an unsigned 64-bit integer.

3.4.3.3.1.4. `Datafile::getFree()`

Description. This method gets the free space available in the datafile.

Signature.

```
Uint64 getFree
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of bytes free in the datafile, as an unsigned 64-bit integer.

3.4.3.3.1.5. `Datafile::getTablespace()`

Description. This method can be used to obtain the name of the tablespace to which the datafile belongs.

Note

You can also access the associated tablespace's ID directly. See Section 3.4.3.3.1.6, “`Datafile::getTablespaceId()`”.

Signature.

```
const char* getTablespace
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the associated tablespace (as a character pointer).

3.4.3.3.1.6. `Datafile::getTablespaceId()`

Description. This method gets the ID of the tablespace to which the datafile belongs.

> **Note**
>
> You can also access the name of the associated tablespace directly. See Section 3.4.3.3.1.5, “`Datafile::getTablespace()`”.

Signature.

```
Uint32 getTablespaceId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The is method returns the tablespace ID as an unsigned 32-bit integer.

3.4.3.3.1.7. `Datafile::getNode()`

Description. This method retrieves the ID of the Cluster node on which the datafile resides.

Signature.

```
Uint32 getNode
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The node ID as an unsigned 32-bit integer.

3.4.3.3.1.8. `Datafile::getFileNo()`

Description. This method gets the number of the file within the associated tablespace.

Signature.

```
Uint32 getFileNo
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The file number, as an unsigned 32-bit integer.

3.4.3.3.1.9. `Datafile::getObjectStatus()`

Description. This method is used to obtain the datafile's object status.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The datafile's `Status`. See Section 3.4.3.1.3, "The `Object::Status` Type".

3.4.3.3.1.10. `Datafile::getObjectVersion()`

Description. This method retrieves the datafile's object version.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The datafile's object version, as an integer.

3.4.3.3.1.11. `Datafile::getObjectId()`

Description. This method is used to obtain the object ID of the datafile.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The datafile's object ID, as an integer.

3.4.3.3.1.12. `Datafile::setPath()`

Description. This method sets the path to the datafile on the data node's filesystem.

Signature.

```
const char* getPath
    (
      void
    ) const
```

Parameters. The path to the file, a string (as a character pointer).

Return Value. *None*.

3.4.3.3.1.13. `Datafile::setSize()`

Description. This method sets the size of the datafile.

Signature.

```
void setSize
    (
      Uint64 size
    )
```

Parameters. This method takes a single parameter — the desired *`size`* in bytes for the datafile, as an unsigned 64-bit integer.

Return Value. *None*.

3.4.3.3.1.14. `Datafile::setTablespace()`

Description. This method is used to associate the datafile with a tablespace.

Signatures. `setTablespace()` can be invoked with either the name of the tablespace, as shown here:

```
void setTablespace
    (
      const char* name
    )
```

Or with a reference to a `Tablespace` object.

```
void setTablespace
    (
      const class Tablespace& tablespace
    )
```

Parameters. This method takes a single parameter, which can be either one of the following:

- The *`name`* of the tablespace (as a character pointer).
- A reference *`tablespace`* to the corresponding `Tablespace` object.

Return Value. *None*.

3.4.3.3.1.15. `Datafile::setNode()`

Description. Designates the node to which this datafile belongs.

Signature.

```
void setNode
    (
      Uint32 nodeId
    )
```

Parameters. The *`nodeId`* of the node on which the datafile is to be located (an unsigned 32-bit integer value).

Return Value. *None*.

3.4.3.4. The `Event` Class

This section discusses the Event class, its methods and defined types.

Description. This class represents a database event in a MySQL Cluster.

Public Methods. The following table lists the public methods of the Event class and the purpose or use of each method:

Method	Purpose / Use
Event()	Class constructor
~Event()	Destructor
getName()	Gets the event's name
getTable()	Gets the Table object on which the event is defined
getTableName()	Gets the name of the table on which the event is defined
getTableEvent()	Checks whether an event is to be detected
getDurability()	Gets the event's durability
getReport()	Gets the event's reporting options
getNoOfEventColumns()	Gets the number of columns for which an event is defined
getEventColumn()	Gets a column for which an event is defined
getObjectStatus()	Gets the event's object status
getObjectVersion()	Gets the event's object version
getObjectId()	Gets the event's object ID
setName()	Sets the event's name
setTable()	Sets the Table object on which the event is defined
addTableEvent()	Adds the type of event that should be detected
setDurability()	Sets the event's durability
setReport()	The the event's reporting options
addEventColumn()	Adds a column on which events should be detected
addEventColumns()	Adds multiple columns on which events should be detected
mergeEvents()	Stes the event's merge flag

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.4.3.4.2, “Event Class Methods”.

Public Types. These are the public types of the Event class:

Type	Purpose / Use
TableEvent	Represents the type of a table event
EventDurability	Specifies an event's scope, accessibility, and lifetime
EventReport	Specifies the reporting option for a table event

For a discussion of each of these types, along with its possible values, see Section 3.4.3.4.1, “Event Types”.

Class Diagram. This diagram shows all the available methods and enumerated types of the Event class:

3.4.3.4.1. `Event` Types

This section details the public types belonging to the `Event` class.

3.4.3.4.1.1. The `Event::TableEvent` Type

This section describes `TableEvent`, a type defined by the `Event` class.

Description. `TableEvent` is used to classify the types of events that may be associated with tables in the `NDB` API.

Enumeration Values.

Value	Description
TE_INSERT	Insert event on a table
TE_DELETE	Delete event on a table
TE_UPDATE	Update event on a table
TE_DROP	Occurs when a table is dropped
TE_ALTER	Occurs when a table definition is changed
TE_CREATE	Occurs when a table is created
TE_GCP_COMPLETE	Occurs on Cluster failures
TE_CLUSTER_FAILURE	Occurs on the completion of a global checkpoint
TE_STOP	Occurs when an event operation is stopped
TE_NODE_FAILURE	Occurs when a Cluster node fails
TE_SUBSCRIBE	Occurs when a cluster node subscribes to an event
TE_UNSUBSCRIBE	Occurs when a cluster node unsubscribes from an event
TE_ALL	Occurs when any event occurs on a table (not relevant when a specific event is received)

3.4.3.4.1.2. The Event::EventDurability Type

This section discusses EventDurability, a type defined by the Event class.

Description. The values of this type are used to describe an event's lifetime or persistence as well as its scope.

Enumeration Values.

Value	Description
ED_UNDEFINED	The event is undefined or of an unsupported type
ED_SESSION	This event persists only for the duration of the current session, and is available only to the current application. It is deleted after the application disconnects or following a cluster restart
ED_TEMPORARY	Any application may use the event, but it is deleted following a cluster restart
ED_PERMANENT	Any application may use the event, and it persists until deleted by an application — even following a cluster restart

3.4.3.4.1.3. The Event::EventReport Type

This section discusses EventReport, a type defined by the Event class.

Description. The values of this type are used to specify reporting options for table events.

Enumeration Values.

Value	Description
ER_UPDATED	Reporting of update events
ER_ALL	Reporting of all events, except for those not resulting in any up-

Value	Description
	dates to the inline parts of BLOB columns
ER_SUBSCRIBE	Reporting of subscription events

3.4.3.4.2. Event Class Methods

3.4.3.4.2.1. Event Constructor

Description. The Event constructor creates a new instance with a given name, and optionally associated with a table.

Signatures. Name only:

```
Event
    (
      const char* name
    )
```

Name and associated table:

```
Event
    (
      const char*                  name,
      const NdbDictionary::Table& table
    )
```

Parameters. At a minimum, a *name* (as a constant character pointer) for the event is required. Optionally, an event may also be associated with a table; this argument, when present, is a reference to a Table object (see Section 3.4.3.7, "The Table Class").

Return Value. A new instance of Event.

Destructor. A destructor for this class is supplied as a virtual method which takes no arguments and whose return type is void.

3.4.3.4.2.2. Event::getName()

Description. This method obtains the name of the event.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the event, as a character pointer.

3.4.3.4.2.3. Event::getTable()

Description. This method is used to find the table with which an event is associated. It returns a reference to the corresponding Table object. You may also obtain the name of the table directly using getTableName(). (For details, see Section 3.4.3.4.2.4, "Event::getTableName()".)

Signature.

```
const NdbDictionary::Table* getTable
    (
```

```
      void
    ) const
```

Parameters. *None*.

Return Value. The table with which the event is associated — if there is one — as a pointer to a `Table` object; otherwise, this method returns `NULL`. (See Section 3.4.3.7, "The `Table` Class".)

3.4.3.4.2.4. `Event::getTableName()`

Description. This method obtains the name of the table with which an event is associated, and can serve as a convenient alternative to `getTable()`. (See Section 3.4.3.4.2.3, "`Event::getTable()`".)

Signature.

```
const char* getTableName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the table associated with this event, as a character pointer.

3.4.3.4.2.5. `Event::getTableEvent()`

Description. This method is used to check whether a given table event will be detected.

Signature.

```
bool getTableEvent
    (
      const TableEvent te
    ) const
```

Parameters. This method takes a single parameter - the table event's type, that is, a `TableEvent` value. See Section 3.4.3.4.1.1, "The `Event::TableEvent` Type", for the list of possible values.

Return Value. This method returns `true` if events of `TableEvent` type *`te`* will be detected. Otherwise, the return value is `false`.

3.4.3.4.2.6. `Event::getDurability()`

Description. This method gets the event's lifetime and scope (that is, its `EventDurability`).

Signature.

```
EventDurability getDurability
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `EventDurability` value. See Section 3.4.3.4.1.2, "The `Event::EventDurability` Type", for possible values.

3.4.3.4.2.7. `Event::getReport()`

Description. This method is used to obtain the reporting option in force for this event.

Signature.

```
EventReport getReport
    (
      void
    ) const
```

Parameters. *None*.

Return Value. One of the reporting options specified in Section 3.4.3.4.1.3, “The `Event::EventReport` Type”.

3.4.3.4.2.8. `Event::getNoOfEventColumns()`

Description. This method obtains the number of columns on which an event is defined.

Signature.

```
int getNoOfEventColumns
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of columns (as an integer), or `-1` in the case of an error.

3.4.3.4.2.9. `Event::getEventColumn()`

Description. This method is used to obtain a specific column from among those on which an event is defined.

Signature.

```
const Column* getEventColumn
    (
      unsigned no
    ) const
```

Parameters. The number (*no*) of the column, as obtained using `getNoOfColumns()` (see Section 3.4.3.4.2.8, “`Event::getNoOfEventColumns()`”).

Return Value. A pointer to the `Column` corresponding to *no*.

3.4.3.4.2.10. `Event::getObjectStatus()`

Description. This method gets the object status of the event.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object status of the event — for possible values, see Section 3.4.3.1.3, “The `Object::Status` Type”.

3.4.3.4.2.11. Event::getObjectVersion()

Description. This method gets the event's object version.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object version of the event, as an integer.

3.4.3.4.2.12. Event::getObjectId()

Description. This method retrieves an event's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID of the event, as an integer.

3.4.3.4.2.13. Event::setName()

Description. This method is used to set the name of an event. The name must be unique among all events visible from the current application (see Section 3.4.3.4.2.6, "Event::getDurability()").

> **Note**
>
> You can also set the event's name when first creating it. See Section 3.4.3.4.2.1, "Event Constructor".

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. The *name* to be given to the event (as a constant character pointer).

Return Value. *None*.

3.4.3.4.2.14. Event::setTable()

Description. This method defines a table on which events are to be detected.

> **Note**
>
> By default, event detection takes place on all columns in the table. Use addEvent-Column() to override this behaviour. For details, see Section 3.4.3.4.2.18, "Event::addEventColumn()".

Signature.

```
void setTable
    (
      const NdbDictionary::Table& table
    )
```

Parameters. This method requires a single parameter, a reference to the table (see Section 3.4.3.7, “The `Table` Class”) on which events are to be detected.

Return Value. *None*.

3.4.3.4.2.15. `Event::addTableEvent()`

Description. This method is used to add types of events that should be detected.

Signature.

```
void addTableEvent
    (
      const TableEvent te
    )
```

Parameters. This method requires a `TableEvent` value. See Section 3.4.3.4.1.1, “The `Event::TableEvent` Type” for possible values.

Return Value. *None*.

3.4.3.4.2.16. `Event::setDurability()`

Description. This method sets an event's durability — that is, its lifetime and scope.

Signature.

```
void setDurability(EventDurability ed)
```

Parameters. This method requires a single `EventDurability` value as a parameter. See Section 3.4.3.4.1.2, “The `Event::EventDurability` Type”, for possible values.

Return Value. *None*.

3.4.3.4.2.17. `Event::setReport()`

Description. This method is used to set a reporting option for an event. Possible option values may be found in Section 3.4.3.4.1.3, “The `Event::EventReport` Type”.

Signature.

```
void setReport
    (
      EventReport er
    )
```

Parameters. An `EventReport` option value.

Return Value. *None*.

3.4.3.4.2.18. `Event::addEventColumn()`

Description. This method is used to add a column on which events should be detected. The column

may be indicated either by its ID or its name.

> **Important**
>
> You must invoke `Dictionary::createEvent()` before any errors will be detected. See Section 3.4.1.1.12, "`Dictionary::createEvent()`".

> **Note**
>
> If you know several columns by name, you can enable event detection on all of them at one time by using `addEventColumns()`. See Section 3.4.3.4.2.19, "`Event::addEventColumns()`".

Signature. Identifying the event using its column ID:

```
void addEventColumn
    (
      unsigned attrId
    )
```

Identifying the column by name:

```
void addEventColumn
    (
      const char* columnName
    )
```

Parameters. This method takes a single argument, which may be either one of:

- The column ID (`attrId`), which should be an integer greater than or equal to `0`, and less than the value returned by `getNoOfEventColumns()`.
- The column's *`name`* (as a constant character pointer).

Return Value. *None*.

3.4.3.4.2.19. `Event::addEventColumns()`

Description. This method is used to enable event detection on several columns at the same time. You must use the names of the columns.

> **Important**
>
> As with `addEventColumn()`, you must invoke `Dictionary::createEvent()` before any errors will be detected. See Section 3.4.1.1.12, "`Dictionary::createEvent()`".

Signature.

```
void addEventColumns
    (
      int          n,
      const char** columnNames
    )
```

Parameters. This method requires two arguments:

- The number of columns *`n`* (an integer).
- The names of the columns *`columnNames`* — this must be passed as a pointer to a character pointer.

Return Value. *None*.

3.4.3.4.2.20. `Event::mergeEvents()`

Description. This method is used to set the *merge events flag*, which is `false` by default. Setting it to `true` implies that events are merged as follows:

- For a given `NdbEventOperation` associated with this event, events on the same primary key within the same global checkpoint index (GCI) are merged into a single event.
- A blob table event is created for each blob attribute, and blob events are handled as part of main table events.
- Blob post/pre data from blob part events can be read via `NdbBlob` methods as a single value.

> **Note**
>
> Currently this flag is not inherited by `NdbEventOperation`, and must be set on `NdbEventOperation` explicitly. See Section 3.5, "The `NdbEventOperation` Class".

Signature.

```
void mergeEvents
    (
      bool flag
    )
```

Parameters. A Boolean *`flag`* value.

Return Value. *None*.

3.4.3.5. The `Index` Class

This section provides a reference to the `Index` class and its public members.

Description. This class represents an index on an `NDB Cluster` table column. It is a descendant of the `NdbDictionary` class, via the `Object` class. For information on these, see Section 3.4, "The `NdbDictionary` Class", and Section 3.4.3, "The `Object` Class".

Public Methods. The following table lists the public methods of `Index` and the purpose or use of each method:

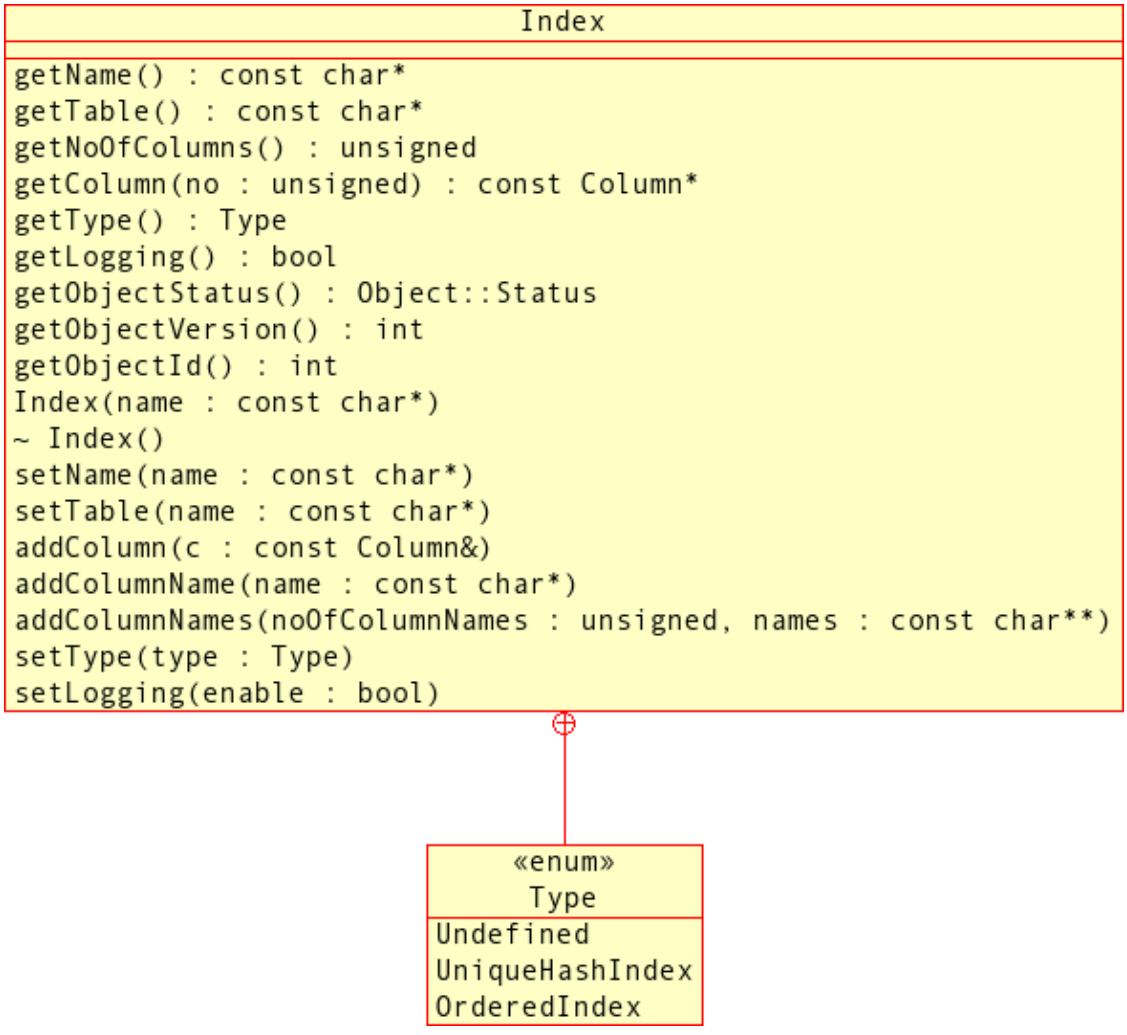
Method	Purpose / Use
`Index()`	Class constructor
`~Index()`	Destructor
`getName()`	Gets the name of the index
`getTable()`	Gets the name of the table being indexed
`getNoOfColumns()`	Gets the number of columns belonging to the index
`getColumn()`	Gets a column making up (part of) the index
`getType()`	Gets the index type
`getLogging()`	Checks whether the index is logged to disk
`getObjectStatus()`	Gets the index object status
`getObjectVersion()`	Gets the index object status

Method	Purpose / Use
`getObjectId()`	Gets the index object ID
`setName()`	Sets the name of the index
`setTable()`	Sets the name of the table to be indexed
`addColumn()`	Adds a `Column` object to the index
`addColumnName()`	Adds a column by name to the index
`addColumnNames()`	Adds multiple columns by name to the index
`setType()`	Set the index type
`setLogging()`	Enable/disable logging of the index to disk

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.4.3.5.2, "`Index` Class Methods".

Public Types. Index has one public type, the `Type` type. For a discussion of Type, see Section 3.4.3.5.1, "The `Index::Type` Type".

Class Diagram. This diagram shows all the available methods and enumerated types of the `Index` class:



3.4.3.5.1. The `Index::Type` Type

Description. This is an enumerated type which describes the sort of column index represented by a given instance of `Index`.

> **Caution**
>
> Do not confuse this enumerated type with `Object::Type`, which is discussed in Section 3.4.3.1.5, "The `Object::Type` Type", or with `Table::Type` or `Column::Type`.

Enumeration Values.

Value	Description
`Undefined`	Undefined object type (initial/default value)
`UniqueHashIndex`	Unique unordered hash index (only index type currently supported)
`OrderedIndex`	Non-unique, ordered index

3.4.3.5.2. `Index` Class Methods

This section contain descriptions of all public methods of the `Index` class. This class has relatively few methods (compared to, say, `Table`), which are fairly straightforward to use.

> **Important**
>
> If you create or change indexes using the `NDB` API, these modifications cannot be seen by MySQL. The only exception to this is renaming the index using `Index::setName()`.

3.4.3.5.2.1. `Index` Class Constructor

Description. This is used to create an new instance of `Index`.

> **Important**
>
> Indexes created using the `NDB` API cannot be seen by the MySQL Server.

Signature.

```
Index
    (
      const char* name = ""
    )
```

Parameters. The name of the new index. It is possible to create an index without a name, and then assign a name to it later using `setName()`. See Section 3.4.3.5.2.11, "`Index::setName()`".

Return Value. A new instance of `Index`.

Destructor. The destructor (`~Index()`) is supplied as a virtual method.

3.4.3.5.2.2. `Index::getName()`

Description. This method is used to obtain the name of an index.

Signature.

```
const char* getName
```

```
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the index, as a constant character pointer.

3.4.3.5.2.3. `Index::getTable()`

Description. This method can be used to obtain the name of the table to which the index belongs.

Signature.

```
const char* getTable
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the table, as a constant character pointer.

3.4.3.5.2.4. `Index::getNoOfColumns()`

Description. This method is used to obtain the number of columns making up the index.

Signature.

```
unsigned getNoOfColumns
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An unsigned integer representing the number of columns in the index.

3.4.3.5.2.5. `Index::getColumn()`

Description. This method retrieves the column at the specified position within the index.

Signature.

```
const Column* getColumn
    (
      unsigned no
    ) const
```

Parameters. The ordinal position number *no* of the column, as an unsigned integer. Use the `getNoOfColumns()` method to determine how many columns make up the index — see Section 3.4.3.5.2.4, “`Index::getNoOfColumns()`”, for details.

Return Value. The column having position *no* in the index, as a pointer to an instance of `Column`. See Section 3.4.2, “The `Column` Class”.

3.4.3.5.2.6. `Index::getType()`

Description. This method can be used to find the type of index.

Signature.

```
Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An index type. See Section 3.4.3.5.1, “The `Index::Type` Type”, for possible values.

3.4.3.5.2.7. `Index::getLogging()`

Description. Use this method to determine whether logging to disk has been enabled for the index.

> **Note**
>
> Indexes which are not logged are rebuilt when the cluster is started or restarted.
>
> Ordered indexes currently do not support logging to disk; they are rebuilt each time the cluster is started. (This includes restarts.)

Signature.

```
bool getLogging
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A Boolean value:

- `true`: The index is being logged to disk.
- `false`: The index is not being logged.

3.4.3.5.2.8. `Index::getObjectStatus()`

Description. This method gets the object status of the index.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `Status` value — see Section 3.4.3.1.3, “The `Object::Status` Type”, for more information.

3.4.3.5.2.9. `Index::getObjectVersion()`

Description. This method gets the object version of the index.

Signature.

```
virtual int getObjectVersion
    (
      void
```

```
    ) const
```

Parameters. *None*.

Return Value. The object version for the index, as an integer.

3.4.3.5.2.10. `Index::getObjectId()`

Description. This method is used to obtain the object ID of the index.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID, as an integer.

3.4.3.5.2.11. `Index::setName()`

Description. This method sets the name of the index.

> **Note**
>
> This is the only `Index::set*()` method whose result is visible to a MySQL Server.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. The desired *`name`* for the index, as a constant character pointer.

Return Value. *None*.

3.4.3.5.2.12. `Index::setTable()`

Description. This method sets the table that is to be indexed. The table is referenced by name.

Signature.

```
void setTable
    (
      const char* name
    )
```

Parameters. The *`name`* of the table to be indexed, as a constant character pointer.

Return Value. *None*.

3.4.3.5.2.13. `Index::addColumn()`

Description. This method may be used to add a column to an index.

> **Note**

The order of the columns matches the order in which they are added to the index. However, this matters only with ordered indexes.

Signature.

```
void addColumn
    (
      const Column& c
    )
```

Parameters. A reference *c* to the column which is to be added to the index.

Return Value. *None*.

3.4.3.5.2.14. Index::addColumnName()

Description. This method works in the same way as addColumn(), except that it takes the name of the column as a parameter. See Section 3.4.3.5.2.5, "Index::getColumn()".

Signature.

```
void addColumnName
    (
      const char* name
    )
```

Parameters. The *name* of the column to be added to the index, as a constant character pointer.

Return Value. *None*.

3.4.3.5.2.15. Index::addColumnNames()

Description. This method is used to add several column names to an index definition at one time.

Note

As with the addColumn() and addColumnName() methods, the indexes are numbered in the order in which they were added. (However, this matters only for ordered indexes.)

Signature.

```
void addColumnNames
    (
      unsigned     noOfNames,
      const char** names
    )
```

Parameters. This method takes two parameters:

- The number of columns/names *noOfNames* to be added to the index.
- The *names* to be added (as a pointer to a pointer).

Return Value. *None*.

3.4.3.5.2.16. Index::setType()

Description. This method is used to set the index type.

Signature.

```
void setType
    (
      Type type
    )
```

Parameters. The *type* of index. For possible values, see Section 3.4.3.5.1, "The Index::Type Type".

Return Value. *None*.

3.4.3.5.2.17. Index::setLogging

Description. This method is used to enable or disable logging of the index to disk.

Signature.

```
void setLogging
    (
      bool enable
    )
```

Parameters. setLogging() takes a single Boolean parameter *enable*. If *enable* is true, then logging is enabled for the index; if false, then logging of this index is disabled.

Return Value. *None*.

3.4.3.6. The LogfileGroup Class

This section discusses the LogfileGroup class, which represents a MySQL Cluster Disk Data logfile group.

Description. This class represents a MySQL Cluster Disk Data logfile group, which is used for storing Disk Data undofiles. For general information about logfile groups and undofiles, see the MySQL Cluster Disk Data Storage [http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-disk-data.html], in the MySQL Manual.

> **Note**
>
> In MySQL 5.1, only unindexed column data can be stored on disk. Indexes and indexes columns continue to be stored in memory as with previous versions of MySQL Cluster.
>
> Versions of MySQL prior to 5.1 do not support Disk Data storage and so do not support log-file groups; thus the LogfileGroup class is unavailable for NDB API applications written against these MySQL versions.

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
LogfileGroup()	Class constructor
~LogfileGroup()	Virtual destructor
getName()	Retrieves the logfile group's name
getUndoBufferSize()	Gets the size of the logfile group's UNDO buffer
getAutoGrowSpecification()	Gets the logfile group's AutoGrowSpecification values

Method	Purpose / Use
getUndoFreeWords()	Retrieves the amount of free space in the UNDO buffer
getObjectStatus()	Gets the logfile group's object status value
getObjectVersion()	Retrieves the logfile group's object version
getObjectId()	Get the object ID of the logfile group
setName()	Sets the name of the logfile group
setUndoBufferSize()	Sets the size of the logfile group's UNDO buffer.
setAutoGrowSpecification()	Sets AutoGrowSpecification values for the logfile group

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.4.3.6.1, “LogfileGroup Class Methods”.

Public Types. The LogfileGroup class does not itself define any public types. However, two of its methods make use of the AutoGrowSpecification data structure as a parameter or return value. For more information, see Section 3.4.4, “The AutoGrowSpecification Structure”.

Class Diagram. This diagram shows all the available public methods of the LogfileGroup class:

```
LogfileGroup
---
LogfileGroup()
LogfileGroup(lGroup : const LogfileGroup&)
~ LogfileGroup()
getName() : const char*
getUndoBufferSize() : Uint32
getAutoGrowSpecification() : const AutoGrowSpecification&
getUndoFreeWords() : Uint64
getObjectStatus() : Object::Status
getObjectVersion() : int
getObjectId() : int
setName(name : const char*)
setUndoBufferSize(size : Uint32)
setAutoGrowSpecification(autoGrowSpec : const AutoGrowSpecification&)
```

3.4.3.6.1. LogfileGroup Class Methods

This section provides descriptions for the public methods of the LogfileGroup class.

3.4.3.6.1.1. LogfileGroup Constructor

Description. The LogfileGroup class has two public constructors, one of which takes no arguments and creates a completely new instance. The other is a copy constructor.

> **Note**
>
> The Dictionary class also supplies methods for creating and destroying LogfileGroup objects. See Section 3.4.1, “The Dictionary Class”.

Signatures. New instance:

```
LogfileGroup
    (
      void
    )
```

Copy constructor:

```
LogfileGroup
    (
      const LogfileGroup& logfileGroup
    )
```

Parameters. When creating a new instance, the constructor takes no parameters. When copying an existing instance, the constructor is passed a reference to the `LogfileGroup` instance to be copied.

Return Value. A `LogfileGroup` object.

Destructor.

```
virtual ~LogfileGroup
    (
      void
    )
```

Examples.

```
[To be supplied...]
```

3.4.3.6.1.2. `LogfileGroup::getName()`

Description. This method gets the name of the logfile group.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The logfile group's name, a string (as a character pointer).

3.4.3.6.1.3. `LogfileGroup::getUndoBufferSize()`

Description. This method retrieves the size of the logfile group's `UNDO` buffer.

Signature.

```
Uint32 getUndoBufferSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The size of the `UNDO` buffer, in bytes.

3.4.3.6.1.4. `LogfileGroup::getAutoGrowSpecification()`

Description. This method retrieves the `AutoGrowSpecification` associated with the logfile group.

Signature.

```
const AutoGrowSpecification& getAutoGrowSpecification
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `AutoGrowSpecification` data structure. See Section 3.4.4, "The `AutoGrowSpecification` Structure", for details.

3.4.3.6.1.5. `LogfileGroup::getUndoFreeWords()`

Description. This method retrieves the number of bytes unused in the logfile group's `UNDO` buffer.

Signature.

```
Uint64 getUndoFreeWords
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of bytes free, as a 64-bit integer.

3.4.3.6.1.6. `LogfileGroup::getObjectStatus()`

Description. This method is used to obtain the object status of the `LogfileGroup`.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The logfile group's `Status` — see Section 3.4.3.1.3, "The `Object::Status` Type" for possible values.

3.4.3.6.1.7. `LogfileGroup::getObjectVersion()`

Description. This method gets the logfile group's object version.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object version of the logfile group, as an integer.

3.4.3.6.1.8. `LogfileGroup::getObjectId()`

Description. This method is used to retrieve the object ID of the logfile group.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The logfile group's object ID (an integer value).

3.4.3.6.1.9. `LogfileGroup::setName()`

Description. This method is used to set a name for the logfile group.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. The *`name`* to be given to the logfile group (character pointer).

Return Value. *None*.

3.4.3.6.1.10. `LogfileGroup::setUndoBufferSize()`

Description. This method can be used to set the size of the logfile group's `UNDO` buffer.

Signature.

```
void setUndoBufferSize
    (
      Uint32 size
    )
```

Parameters. The *`size`* in bytes for the `UNDO` buffer (using a 32-bit unsigned integer value).

Return Value. *None*.

3.4.3.6.1.11. `LogfileGroup::setAutoGrowSpecification()`

Description. This method sets to the `AutoGrowSpecification` data for the logfile group.

Signature.

```
void setAutoGrowSpecification
    (
      const AutoGrowSpecification& autoGrowSpec
    )
```

Parameters. The data is passed as a single parameter, an `AutoGrowSpecification` data structure — see Section 3.4.4, “The `AutoGrowSpecification` Structure”.

Return Value. *None*.

3.4.3.7. The `Table` Class

This section describes the `Table` class, which models a database table in the `NDB` API.

Description. The `Table` class represents a table in a MySQL Cluster database. This class extends the `Object` class, which in turn is an inner class of the `NdbDictionary` class.

Important

It is possible using the NDB API to create tables independently of the MySQL server. However, it is usually not advisable to do so, since tables created in this fashion cannot be seen by the MySQL server. Similarly, it is possible using Table methods to modify existing tables, but these changes (except for renaming tables) are not visible to MySQL.

Calculating Table Sizes. When calculating the data storage one should add the size of all attributes (each attribute consuming a minimum of 4 bytes) and well as 12 bytes overhead. Variable size attributes have a size of 12 bytes plus the actual data storage parts, with an additional overhead based on the size of the variable part. For example, consider a table with 5 attributes: one 64-bit attribute, one 32-bit attribute, two 16-bit attributes, and one array of 64 8-bit attributes. The amount of memory consumed per record by this table is the sum of the following:

- 8 bytes for the 64-bit attribute
- 4 bytes for the 32-bit attribute
- 8 bytes for the two 16-bit attributes, each of these taking up 4 bytes due to right-alignment
- 64 bytes for the array (64 * 1 byte per array element)
- 12 bytes overhead

This totals 96 bytes per record. In addition, you should assume an overhead of about 2% for the allocation of page headers and wasted space. Thus, 1 million records should consume 96 MB, and the additional page header and other overhead comes to approximately 2 MB. Rounding up yields 100 MB.

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
Table()	Class constructor
~Table()	Destructor
getName()	Gets the table's name
getTableId()	Gets the table's ID
getColumn()	Gets a column (by name) from the table
getLogging()	Checks whether logging to disk is enabled for this table
getFragmentType()	Gets the table's FragmentType
getKValue()	Gets the table's KValue
getMinLoadFactor()	Gets the table's minimum load factor
getMaxLoadFactor()	Gets the table's maximum load factor
getNoOfColumns()	Gets the number of columns in the table
getNoOfPrimaryKeys()	Gets the number of primary keys in the table
getPrimaryKey()	Gets the name of the table's primary key
equal()	Compares the table with another table
getFrmData()	Gets the data from the table .FRM file
getFrmLength()	Gets the length of the table's .FRM file
getFragmentData()	Gets table fragment data (ID, state, and node group)
getFragmentDataLen()	Gets the length of the table fragment data

Method	Purpose / Use
getRangeListData()	Gets a RANGE or LIST array
getRangeListDataLen()	Gets the length of the table RANGE or LIST array
getTablespaceData()	Gets the ID and version of the tablespace containing the table
getTablespaceDataLen()	Gets the length of the table's tablespace data
getLinearFlag()	Gets the current setting for the table's linear hashing flag
getFragmentCount()	Gets the number of fragments for this table
getTablespace()	Gets the tablespace containing this table
getTablespaceNames()	
getObjectType()	Gets the table's object type (Object::Type as opposed to Table::Type)
getObjectStatus()	Gets the table's object status
getObjectVersion()	Gets the table's object version
getObjectId()	Gets the table's object ID
getMaxRows()	Gets the maximum number of rows that this table may contain
getDefaultNoPartitionsFlag()	Checks whether the default number of partitions is being used
getRowGCIIndicator()	
getRowChecksumIndicator()	
setName()	Sets the table's name
addColumn()	Adds a column to the table
setLogging()	Toggle logging of the table to disk
setLinearFlag()	Sets the table's linear hashing flag
setFragmentCount()	Sets the number of fragments for this table
setFragmentationType()	Sets the table's FragmentType
setKValue()	Set the KValue
setMinLoadFactor()	Set the table's minimum load factor (MinLoadFactor)
setMaxLoadFactor()	Set the table's maximum load factor (MaxLoadFactor)
setTablespace()	Set the tablespace to use for this table
setStatusInvalid()	
setMaxRows()	Sets the maximum number of rows in the table
setNoDefaultPartitionsFlag()	Toggles whether the default number of partitions should be used for the table
setFrm()	Sets the .FRM file to be used for this table
setFragmentData()	Sets the fragment ID, node group ID, and fragment state
setTablespaceNames()	Sets the tablespace names for fragments
setTablespaceData()	Sets the tablespace ID and version
setRangeListData()	Sets LIST and RANGE partition data
setObjectType()	Sets the table's object type
setRowGCIIndicator()	

Method	Purpose / Use
setRowChecksumIndicator()	

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.4.3.7.1, “Table Class Methods”.

Public Types. The Table class defines no public types.

Class Diagram. This diagram shows all the available methods of the Table class:

Table
getName() : const char*
getTableId() : int
getColumn(name : const char*) : const Column*
getColumn(attributeId : const int) : Column*
getLogging() : bool
getFragmentType() : FragmentType
getKValue() : int
getMinLoadFactor() : int
getMaxLoadFactor() : int
getNoOfColumns() : int
getNoOfPrimaryKeys() : int
getPrimaryKey(no : int) : const char*
equal(table : const Table&) : bool
getFrmData() : const void*
getFrmLength() : Uint32
getFragmentData() : const void*
getFragmentDataLen() : Uint32
getRangeListData() : const void*
getRangeListDataLen() : Uint32
getTablespaceData() : const void*
getTablespaceDataLen() : Uint32
Table(name : const char*)
Table(table : const Table&)
~ Table()
operator =(table : const Table&) : Table&
setName(name : const char*)
addColumn(column : const Column&)
setLogging(enable : bool)
setLinearFlag(flag : Uint32)
getLinearFlag() : bool
setFragmentCount(count : Uint32)
getFragmentCount() : Uint32
setFragmentType(fragmentType : FragmentType)
setKValue(kValue : int)
setMinLoadFactor(min : int)
setMaxLoadFactor(max : int)
setTablespace(name : const char*)
setTablespace(tablespace : class const Tablespace&)
getTablespace() : const char*
getTablespace(id : Uint32*, version : Uint32*) : bool
getObjectType() : Object::Type
getObjectStatus() : Object::Status
setStatusInvalid()
getObjectVersion() : int
setMaxRows(maxRows : Uint64)
getMaxRows() : Uint64
setDefaultNoPartitionsFlag(indicator : Uint32)
getDefaultNoPartitionsFlag() : Uint32
getObjectId() : int
setFrm(data : const void*, len : Uint32)
setFragmentData(data : const void*, len : Uint32)
setTablespaceNames(data : const void*, len : Uint32)
getTablespaceNames() : const void*
getTablespaceNamesLen() : Uint32
setTablespaceData(data : const void*, len : Uint32)
setRangeListData(data : const void*, len : Uint32)
setObjectType(type : Object::Type)
setRowGCIIndicator(value : bool)
getRowGCIIndicator() : bool
setRowChecksumIndicator(value : bool)
getRowChecksumIndicator() : bool

3.4.3.7.1. Table Class Methods

This section discusses the public methods of the Table class.

> **Note**
>
> The assignment (=) operator is overloaded for this class, so that it always performs a deep copy.

> **Warning**
>
> As with other database objects, Table object creation and attribute changes to existing tables done using the NDB API are not visible from MySQL. For example, if you add a new column to a table using Table::addColumn(), MySQL will not see the new column. The only exception to this rule with regard to tables is that you can change the name of an existing table using Table::setName().

3.4.3.7.1.1. Table Constructor

Description. Creates a Table instance. There are two version of the Table constructor, one for creating a new instance, and a copy constructor.

> **Important**
>
> Tables created in the NDB API using this method are not accessible from MySQL.

Signature. New instance:

```
Table
    (
      const char* name = ""
    )
```

Copy constructor:

```
Table
    (
      const Table& table
    )
```

Parameters. For a new instance, the name of the table to be created. For a copy, a reference to the table to be copied.

Return Value. A Table object.

Destructor.

```
virtual ~Table()
```

3.4.3.7.1.2. Table:getName()

Description. Gets the name of a table.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the table (a string).

3.4.3.7.1.3. Table::getId()

Description. This method gets a table's ID.

Signature.

```
int getTableId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer.

3.4.3.7.1.4. Table::getColumn()

Description. This method is used to obtain a column definition, given either the index or the name of the column.

Signature. Using the column ID:

```
Column* getColumn
    (
      const int AttributeId
    )
```

Using the column name:

```
Column* getColumn
    (
      const char* name
    )
```

Parameters. Either of: the column's index in the table (as would be returned by the column's `getColumnNo()` method), or the name of the column.

Return Value. A pointer to the column with the specified index or name. If there is no such column, then this method returns `NULL`.

3.4.3.7.1.5. Table::getLogging()

Description. This class is used to check whether a table is logged to disk — that is, whether it is permanent or temporary.

Signature.

```
bool getLogging
    (
      void
    ) const
```

Parameters. *None*.

Return Value. Returns a Boolean value. If this method returns `true`, then full checkpointing and logging are done on the table. If `false`, then the table is a temporary table and is not logged to disk; in the event of a system restart the table still exists and retains its definition, but it will be empty. The default logging value is `true`.

3.4.3.7.1.6. Table::getFragmentationType()

Description. This method gets the table's fragmentation type.

Signature.

```
FragmentType getFragmentType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `FragmentType` value, as defined in Section 3.4.3.1.1, "The `Object::FragmentType` Type".

3.4.3.7.1.7. `Table::getKValue()`

Description. This method gets the KValue, a hashing parameter which is currently restricted to the value `6`. In a future release, it may become feasible to set this parameter to other values.

Signature.

```
int getKValue
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer (currently always `6`).

3.4.3.7.1.8. `Table::getMinLoadFactor()`

Description. This method gets the value of the load factor when reduction of the hash table begins. This should always be less than the value returned by `getMaxLoadFactor()`.

Signature.

```
int getMinLoadFactor
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer (actually, a percentage expressed as an integer — see Section 3.4.3.7.1.9, "`Table::getMaxLoadFactor()`").

3.4.3.7.1.9. `Table::getMaxLoadFactor()`

Description. This method returns the load factor (a hashing parameter) when splitting of the containers in the local hash tables begins.

Signature.

```
int getMaxLoadFactor
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer whose maximum value is 100. When the maximum value is returned, this means that memory usage is optimised. Smaller values indicate that less data is stored in each container, which means that keys are found more quickly; however, this also consumes more memory.

3.4.3.7.1.10. `Table::getNoOfColumns()`

Description. This method is used to obtain the number of columns in a table.

Signature.

```
int getNoOfColumns
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer — the number of columns in the table.

3.4.3.7.1.11. `Table::getNoOfPrimaryKeys()`

Description. This method finds the number of primary key columns in the table.

Signature.

```
int getNoOfPrimaryKeys
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer.

3.4.3.7.1.12. `Table::getPrimaryKey()`

Description. This method is used to obtain the name of the table's primary key.

Signature.

```
const char* getPrimaryKey
    (
      int no
    ) const
```

Parameters. *None*.

Return Value. The name of the primary key, a string (character pointer).

3.4.3.7.1.13. `Table::equal()`

Description. This method is used to compare one instance of `Table` with another.

Signature.

```
bool equal
    (
      const Table& table
    ) const
```

Parameters. A reference to the `Table` object with which the current instance is to be compared.

Return Value. `true` if the two tables are the same, otherwise `false`.

3.4.3.7.1.14. `Table::getFrmData()`

Description. The the data from the `.FRM` file associated with the table.

Signature.

```
const void* getFrmData
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to the `.FRM` data.

3.4.3.7.1.15. `Table::getFrmLength()`

Description. Gets the length of the table's `.FRM` file data, in bytes.

Signature.

```
Uint32 getFrmLength
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The length of the `.FRM` file data (unsigned 32-bit integer).

3.4.3.7.1.16. `Table::getFragmentData()`

Description. This method gets the table's fragment data (ID, state, and node group).

Signature.

```
const void* getFragmentData
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to the data to be read.

3.4.3.7.1.17. `Table::getFragmentDataLen()`

Description. Gets the length of the table fragment data to be read, in bytes.

Signature.

```
Uint32 getFragmentDataLen
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of bytes to be read, as an unsigned 32-bit integer.

3.4.3.7.1.18. Table::getRangeListData()

Description. This method gets the range or list data associated with the table.

Signature.

```
const void* getRangeListData
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to the data.

3.4.3.7.1.19. Table::getRangeListDataLen()

Description. This method gets the size of the table's range or list array.

Signature.

```
Uint32 getRangeListDataLen
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The length of the list or range array, as an integer.

3.4.3.7.1.20. Table::getTablespaceData()

Description. This method gets the table's tablespace data (ID and version).

Signature.

```
const void* getTablespaceData
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to the data.

3.4.3.7.1.21. Table::getTablespaceDataLen()

Description. This method is used to get the length of the table's tablespace data.

Signature.

```
Uint32 getTablespaceDataLen
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The length of the data, as a 32-bit unsigned integer.

3.4.3.7.1.22. Table::getLinearFlag()

Description. This method retrieves the value of the table's linear hashing flag.

Signature.

```
bool getLinearFlag
    (
      void
    ) const
```

Parameters. *None*.

Return Value. `true` if the flag is set, and `false` if it is not.

3.4.3.7.1.23. `Table::getFragmentCount()`

Description. This method gets the number of fragments in the table.

Signature.

```
Uint32 getFragmentCount
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of table fragments, as a 32-bit unsigned integer.

3.4.3.7.1.24. `Table::getTablespace()`

Description. This method is used in two ways: to obtain the name of the tablespace to which this table is assigned; to verify that a given tablespace is the one being used by this table.

Signatures. To obtain the name of the tablespace:

```
const char* getTablespace
    (
      void
    ) const
```

To determine whether the tablespace is the one indicated by the given ID and version:

```
bool getTablespace
    (
      Uint32* id      = 0,
      Uint32* version = 0
    ) const
```

Parameters. The number and types of parameters depend on how this method is being used:

- When used to obtain the name of the tablespace in use by the table, it is called without any arguments.
- When used to determine whether the given tablespace is the one being used by this table, then getTablespace() takes two parameters:

  1. The tablespace *`id`*, given as a pointer to a 32-bit unsigned integer
  2. The tablespace *`version`*, also given as a pointer to a 32-bit unsigned integer

  The default value for both *`id`* and *`version`* is `0`.

Return Value. The return type depends on how the method is called.

- When `getTablespace()` is called without any arguments, it returns a `Tablespace` object instance. See Section 3.4.3.8, "The `Tablespace` Class", for more information.
- When called with two arguments, it returns `true` if the tablespace is the same as the one having the ID and version indicated; otherwise, it returns `false`.

3.4.3.7.1.25. `Table::getObjectType()`

Description. This method is used to obtain the table's type — that is, its `Object::Type` value

Signature.

```
Object::Type getObjectType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. Returns a `Type` value. For possible values, see Section 3.4.3.1.5, "The `Object::Type` Type".

3.4.3.7.1.26. `Table::getStatus()`

Description. This method gets the table's status — that is, its `Object::Status`.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `Status` value. For possible values, see Section 3.4.3.1.3, "The `Object::Status` Type".

3.4.3.7.1.27. `Table::getObjectVersion()`

Description. This method gets the table's object version.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The table's object version, as an integer.

3.4.3.7.1.28. `Table::getMaxRows()`

Description. This method gets the maximum number of rows that the table can hold. This is used for calculating the number of partitions.

Signature.

```
Uint64 getMaxRows
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The maximum number of table rows, as a 64-bit unsigned integer.

3.4.3.7.1.29. `Table::getDefaultNoPartitionsFlag()`

Description. This method is used to find out whether the default number of partitions is used for the table.

Signature.

```
Uint32 getDefaultNoPartitionsFlag
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 32-bit unsigned integer.

3.4.3.7.1.30. `Table::getObjectId()`

Description. This method gets the table's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID is returned as an integer.

3.4.3.7.1.31. `Table::getTablespaceNames()`

Description. This method gets a pointer to the names of the tablespaces used in the table fragments.

Signature.

```
const void* getTablespaceNames
    (
      void
    )
```

Parameters. *None*.

Return Value. A pointer to the tablespace name data.

3.4.3.7.1.32. `Table::getTablespaceNamesLen()`

Description. This method gets the length of the tablespace name data returned by `getTablespaceNames()`. (See Section 3.4.3.7.1.31, “`Table::getTablespaceNames()`”.)

Signature.

```
Uint32 getTablespaceNamesLen
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The length of the names data, in bytes, as a 32-but unsigned integer.

3.4.3.7.1.33. `Table::getRowGCIIndicator()`

Description.

Signature.

```
bool getRowGCIIndicator
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `true`/`false` value.

3.4.3.7.1.34. `Table::getRowChecksumIndicator()`

Description.

Signature.

```
bool getRowChecksumIndicator
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `true`/`false` value.

3.4.3.7.1.35. `Table::setName()`

Description. This method sets the name of the table.

> **Note**
>
> This is the only `set*()` method of `Table` whose effects are visible to MySQL.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. *`name`* is the (new) name of the table.

Return Value. *None*.

3.4.3.7.1.36. `Table::addColumn()`

Description. Adds a column to a table.

Signature.

```
void addColumn
    (
      const Column& column
    )
```

Parameters. A reference to the column which is to be added to the table.

Return Value. *None*; however, it does create a copy of the original `Column` object.

3.4.3.7.1.37. `Table::setLogging()`

Description. Toggles the table's logging state. See Section 3.4.3.7.1.5, "`Table::getLogging()`".

Signature.

```
void setLogging
    (
      bool enable
    )
```

Parameters. If *`enable`* is `true`, then logging for this table is enabled; if it is `false`, then logging is disabled.

Return Value. *None*.

3.4.3.7.1.38. `Table::setLinearFlag()`

Description.

Signature.

```
void setLinearFlag
    (
      Uint32 flag
    )
```

Parameters. The *`flag`* is a 32-bit unsigned integer.

Return Value. *None*.

3.4.3.7.1.39. `Table::setFragmentCount()`

Description. Sets the number of table fragments.

Signature.

```
void setFragmentCount
    (
      Uint32 count
    )
```

Parameters. *`count`* is the number of fragments to be used for the table.

Return Value. *None*.

3.4.3.7.1.40. `Table::setFragmentType()`

Description. This method sets the table's fragmentation type.

Signature.

```
void setFragmentType
    (
      FragmentType fragmentType
    )
```

Parameters. This method takes one argument, a `FragmentType` value. See Section 3.4.3.1.1, "The `Object::FragmentType` Type", for more information.

Return Value. *None*.

3.4.3.7.1.41. `Table::setKValue()`

Description. This sets the `KValue`, a hashing parameter.

Signature.

```
void setKValue
    (
      int kValue
    )
```

Parameters. *`kValue`* is an integer. Currently the only permitted value is `6`. In a future version this may become a variable parameter.

Return Value. *None*.

3.4.3.7.1.42. `Table::setMinLoadFactor()`

Description. This method sets the minimum load factor when reduction of the hash table begins.

Signature.

```
void setMinLoadFactor
    (
      int min
    )
```

Parameters. This method takes a single parameter *`min`*, an integer representation of a percentage (for example, `45` represents 45 percent). For more information, see Section 3.4.3.7.1.8, "`Table::getMinLoadFactor()`".

Return Value. *None*.

3.4.3.7.1.43. `Table::setMaxLoadFactor()`

Description. This method sets the maximum load factor when splitting the containers in the local hash tables.

Signature.

```
void setMaxLoadFactor
    (
      int max
    )
```

Parameters. This method takes a single parameter *`max`*, an integer representation of a percentage (for example, `45` represents 45 percent). For more information, see Section 3.4.3.7.1.9, "`Ta-`

ble::getMaxLoadFactor()".

> **Caution**
>
> This should never be greater than the minimum load factor.

Return Value. *None*.

3.4.3.7.1.44. Table::setTablespace()

Description. This method sets the tablespace for the table.

Signatures. Using the name of the tablespace:

```
void setTablespace
    (
      const char* name
    )
```

Using a Tablespace object:

```
void setTablespace
    (
      const class Tablespace& tablespace
    )
```

Parameters. This method can be called with a single argument of either of two types:

1. The *name* of the tablespace (a string).
2. A reference to an existing Tablespace instance.

See Section 3.4.3.8, "The Tablespace Class".

Return Value. *None*.

3.4.3.7.1.45. Table::setMaxRows()

Description. This method sets the maximum number of rows that can be held by the table.

Signature.

```
void setMaxRows
    (
      Uint64 maxRows
    )
```

Parameters. *maxRows* is a 64-bit unsigned integer that represents the maximum number of rows to be held in the table.

Return Value. *None*.

3.4.3.7.1.46. Table::setDefaultNoPartitionsFlag()

Description. This method sets an indicator that determines whether the default number of partitions is used for the table.

Signature.

```
void setDefaultNoPartitionsFlag
    (
      Uint32 indicator
    ) const
```

Parameters. This method takes a single argument *`indicator`*, a 32-bit unsigned integer.

Return Value. *None*.

3.4.3.7.1.47. `Table::setFrm()`

Description. This method is used to write data to this table's `.FRM` file.

Signature.

```
void setFrm
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method takes two arguments:

- A pointer to the *`data`* to be written.
- The length (*`len`*) of the data.

Return Value. *None*.

3.4.3.7.1.48. `Table::setFragmentData()`

Description. This method writes an array of fragment information containing the following information:

- Fragment ID
- Node group ID
- Fragment State

Signature.

```
void setFragmentData
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method takes two parameters:

- A pointer to the fragment *`data`* to be written
- The length (*`len`*) of this data, in bytes, as a 32-bit unsigned integer

Return Value. *None*.

3.4.3.7.1.49. `Table::setTablespaceNames()`

Description. Sets the names of the tablespaces used by the table fragments.

Signature.

```
void setTablespaceNames
    (
      const void* data
      Uint32      len
    )
```

Parameters. This method takes two parameters:

- A pointer to the tablespace names *`data`*
- The length (*`len`*) of the names data, as a 32-bit unsigned integer.

Return Value. *None*.

3.4.3.7.1.50. `Table::setTablespaceData()`

Description. This method sets the tablespace information for each fragment, and includes a tablespace ID and a tablespace version.

Signature.

```
void setTablespaceData
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method requires two parameters:

- A pointer to the *`data`* containing the tablespace ID and version
- The length (*`len`*) of this data, as a 32-bit unsigned integer.

Return Value. *None*.

3.4.3.7.1.51. `Table::setRangeListData()`

Description. This method sets an array containing information that maps range values and list values to fragments. This is essentially a sorted map consisting of fragment ID/value pairs. For range partitions there is one pair per fragment. For list partitions it could be any number of pairs, but at least as many pairs as there are fragments.

Signature.

```
void setRangeListData
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method requires two parameters:

- A pointer to the range or list *`data`* containing the ID/value pairs
- The length (*`len`*) of this data, as a 32-bit unsigned integer.

Return Value. *None*.

3.4.3.7.1.52. `Table::setObjectType()`

Description. This method sets the table's object type.

Signature.

```
void setObjectType
    (
      Object::Type type
    )
```

Parameters. The desired object *`type`*. This must be one of the `Type` values listed in Section 3.4.3.1.5, "The `Object::Type` Type".

Return Value. *None*.

3.4.3.7.1.53. `Table::setRowGCIIndicator()`

Description.

Signature.

```
void setRowGCIIndicator
    (
      bool value
    ) const
```

Parameters. A `true`/`false` *`value`*.

Return Value. *None*.

3.4.3.7.1.54. `Table::setRowChecksumIndicator()`

Description.

Signature.

```
void setRowChecksumIndicator
    (
      bool value
    ) const
```

Parameters. A `true`/`false` *`value`*.

Return Value. *None*.

3.4.3.8. The `Tablespace` Class

This section discusses the `Tablespace` class and its public members.

Description. The `Tablespace` class models a MySQL Cluster Disk Data tablespace, which contains the datafiles used to store Cluster Disk Data. For an overview of Cluster Disk Data and their characteristics, see `CREATE TABLESPACE` Syntax [http://dev.mysql.com/doc/refman/5.1/en/create-tablespace.html], in the MySQL Manual.

> **Note**
>
> In MySQL 5.1, only unindexed column data can be stored on disk. Indexes and indexes

columns continue to be stored in memory as with previous versions of MySQL Cluster.

Versions of MySQL prior to 5.1 do not support Disk Data storage and so do not support tablespaces; thus the `Tablespace` class is unavailable for `NDB` API applications written against these MySQL versions.

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Tablespace()`	Class constructor
`~Tablespace()`	Virtual destructor method
`getName()`	Gets the name of the tablespace
`getExtentSize()`	Gets the extent size used by the tablespace
`getAutoGrowSpecification()`	Used to obtain the AutoGrowSpecification structure associated with the tablespace
`getDefaultLogfileGroup()`	Gets the name of the tablespace's default logfile group
`getDefaultLogfileGroupId()`	Gets the ID of the tablespace's default logfile group
`getObjectStatus()`	Used to obtain the `Object::Status` of the `Tablespace` instance for which it is called
`getObjectVersion()`	Gets the object version of the `Tablespace` object for which it is invoked
`getObjectId()`	Gets the object ID of a `Tablespace` instance
`setName()`	Sets the name for the tablespace
`setExtentSize()`	Sets the size of the extents used by the tablespace
`setAutoGrowSpecification()`	Used to set the auto-grow characteristics of the tablespace
`setDefaultLogfileGroup()`	Sets the tablespace's default logfile group

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.4.3.8.1, “`Tablespace` Class Methods”.

Public Types. The Tablespace class defines no public types of its own; however, two of its methods make use of the `AutoGrowSpecification` data structure. Information about this structure can be found in Section 3.4.4, “The `AutoGrowSpecification` Structure”.

Class Diagram. This diagram shows all the available methods and enumerated types of the `Tablespace` class:

Tablespace
Tablespace() Tablespace(tablespace : const Tablespace&) ~Tablespace() getName() : const char* getExtentSize() : Uint32 getAutoGrowSpecification() : const AutoGrowSpecification& getDefaultLogfileGroup() : const char* getDefaultLogfileGroupId() : Uint32 getObjectStatus() : Object::Status getObjectVersion() : int getObjectId() : int setName(name : const char*) setExtentSize(size : Uint32) setAutoGrowSpecification(autoGrowSpec : const AutoGrowSpecification&) setDefaultLogfileGroup(name : const char*) setDefaultLogfileGroup(lGroup : class const LogfileGroup&)

3.4.3.8.1. `Tablespace` Class Methods

This section provides details of the public members of the `NDB` API's `Tablespace` class.

3.4.3.8.1.1. `Tablespace` Constructor

Description. These methods are used to create a new instance of `Tablespace`, or to copy an existing one.

> **Note**
>
> The `NdbDictionary::Dictionary` class also supplies methods for creating and dropping tablespaces. See Section 3.4.1, "The `Dictionary` Class".

Signatures. New instance:

```
Tablespace
    (
      void
    )
```

Copy constructor:

```
Tablespace
    (
      const Tablespace& tablespace
    )
```

Parameters. New instance: *None*. Copy constructor: a reference *`tablespace`* to an existing `Tablespace` instance.

Return Value. A `Tablespace` object.

Destructor. The class defines a virtual destructor `~Tablespace()` which takes no arguments and returns no value.

3.4.3.8.1.2. `Tablespace::getName()`

Description. This method retrieves the name of the tablespace.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the tablespace, a string value (as a character pointer).

3.4.3.8.1.3. `Tablespace::getExtentSize()`

Description. This method is used to retrieve the *extent size* — that is the size of the memory allocation units — used by the tablespace.

> **Note**
>
> The same extent size is used for all datafiles contained in a given tablespace.

Signature.

```
Uint32 getExtentSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The tablespace's extent size in bytes, as an unsigned 32-bit integer.

3.4.3.8.1.4. `Tablespace::getAutoGrowSpecification()`

Description.

Signature.

```
const AutoGrowSpecification& getAutoGrowSpecification
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A reference to the structure which describes the tablespace auto-grow characteristics — for details, see Section 3.4.4, “The `AutoGrowSpecification` Structure”.

3.4.3.8.1.5. `Tablespace::getDefaultLogfileGroup()`

Description. This method retrieves the name of the tablespace's default logfile group.

> **Note**
>
> Alternatively, you may wish to obtain the ID of the default logfile group — see Section 3.4.3.8.1.6, “`Tablespace::getDefaultLogfileGroupId()`”.

Signature.

```
const char* getDefaultLogfileGroup
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the logfile group (string value as character pointer).

3.4.3.8.1.6. `Tablespace::getDefaultLogfileGroupId()`

Description. This method retrieves the ID of the tablespace's default logfile group.

> **Note**
>
> You can also obtain directly the name of the default logfile group rather than its ID — see Section 3.4.3.8.1.5, "`Tablespace::getDefaultLogfileGroup()`".

Signature.

```
Uint32 getDefaultLogfileGroupId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The ID of the logfile group, as an unsigned 32-bit integer.

3.4.3.8.1.7. `Tablespace::getObjectStatus()`

Description. This method is used to retrieve the object status of a tablespace.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `Object::Status` value — see Section 3.4.3.1.3, "The `Object::Status` Type", for details.

3.4.3.8.1.8. `Tablespace::getObjectVersion()`

Description. This method gets the tablespace object version.

Signature.

```
virtual int getObjectVersion
  (
    void
  ) const
```

Parameters. *None*.

Return Value. The object version, as an integer.

3.4.3.8.1.9. `Tablespace::getObjectId()`

Description. This method retrieves the tablespace's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID, as an integer.

3.4.3.8.1.10. `Tablespace::setName()`

Description. This method sets the name of the tablespace.

Signature.

```
void setName
    (
      const char* name
    ) const
```

Parameters. The *`name`* of the tablespace, a string (character pointer).

Return Value. *None*.

3.4.3.8.1.11. `Tablespace::setExtentSize()`

Description. This method sets the tablespace's extent size.

Signature.

```
void setExtentSize
    (
      Uint32 size
    )
```

Parameters. The *`size`* to be used for this tablespace's extents, in bytes.

Return Value. *None*.

3.4.3.8.1.12. `Tablespace::setAutoGrowSpecification()`

Description. This method is used to set the auto-grow characteristics of the tablespace.

Signature.

```
void setAutoGrowSpecification
    (
      const AutoGrowSpecification& autoGrowSpec
    )
```

Parameters. This method takes a single parameter, an `AutoGrowSpecification` data structure. See Section 3.4.4, "The `AutoGrowSpecification` Structure".

Return Value. *None*.

3.4.3.8.1.13. `Tablespace::setDefaultLogfileGroup()`

Description. This method is used to set a tablespace's default logfile group.

Signature. This method can be called in two different ways. The first of these uses the name of the logfile group, as shown here:

```
void setDefaultLogfileGroup
    (
      const char* name
    )
```

This method can also be called by passing it a reference to a `LogfileGroup` object:

```
void setDefaultLogfileGroup
    (
      const class LogfileGroup& lGroup
    )
```

> **Note**
>
> There is no method for setting a logfile group as the the default for a tablespace by referencing the logfile group's ID. (In other words, there is no `set*()` method corresponding to `getDefaultLogfileGroupId()`.)

Parameters. Either the *`name`* of the logfile group to be assigned to the tablespace, or a reference *`lGroup`* to this logfile group.

Return Value. *None*.

3.4.3.9. The `Undofile` Class

The section discusses the `Undofile` class and its public methods.

Description. The `Undofile` class models a Cluster Disk Data undofile, which stores data used for rolling back transactions.

> **Note**
>
> In MySQL 5.1, only unindexed column data can be stored on disk. Indexes and indexes columns continue to be stored in memory as with previous versions of MySQL Cluster.
>
> Versions of MySQL prior to 5.1 do not support Disk Data storage and so do not support undofile; thus the `Undofile` class is unavailable for `NDB` API applications written against these MySQL versions.

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

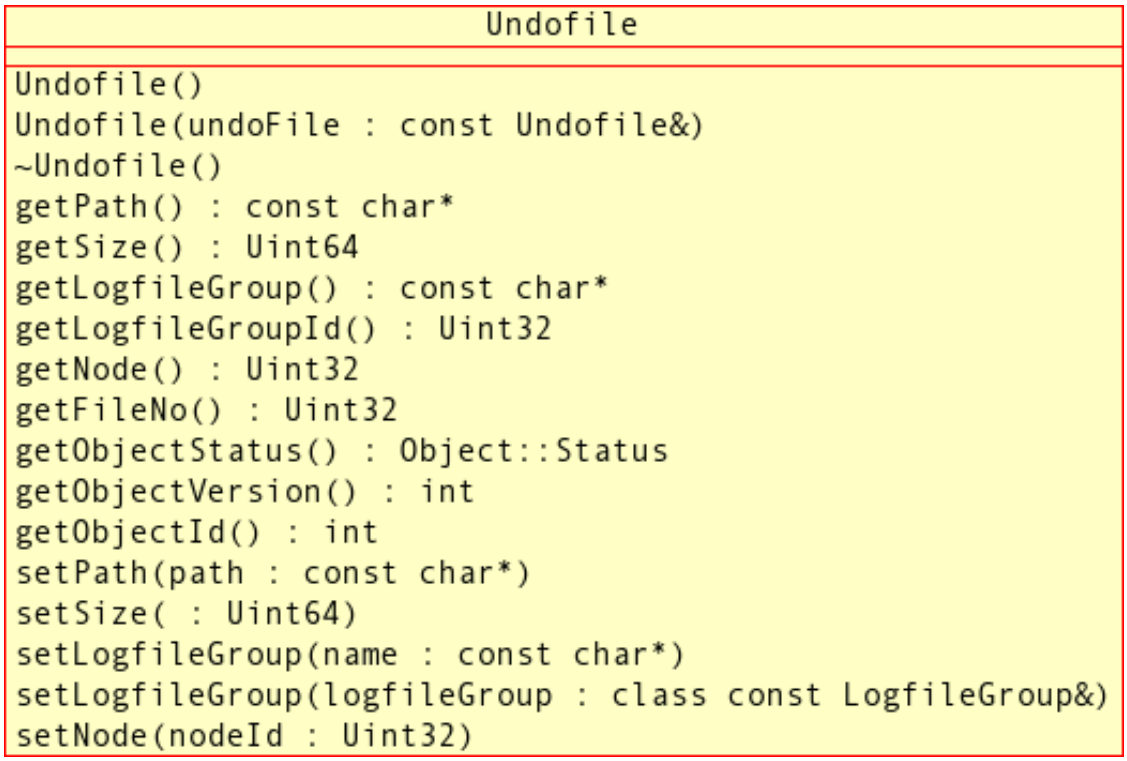
Method	Purpose / Use
`Undofile()`	Class constructor
`~Undofile()`	Virtual destructor
`getPath()`	Gets the undofile's filesystem path
`getSize()`	Gets the size of the undofile
`getLogfileGroup()`	Gets the name of the logfile group to which the undofile belongs
`getLogfileGroupId()`	Gets the ID of the logfile group to which the undofile belongs
`getNode()`	Gets the node where the undofile is located
`getFileNo()`	Gets the number of the undofile in the logfile group
`getObjectStatus()`	Gets the undofile's `Status`
`getObjectVersion()`	Gets the undofile's object version

Method	Purpose / Use
getObjectId()	Gets the undofile's object ID
setPath()	Sets the filesystem path for the undofile
setSize()	Sets the undofile's size
setLogfileGroup()	Sets the undofile's logfile group using the name of the logfile group or a reference to the corresponding LogfileGroup object
setNode()	Sets the node on which the undofile is located

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.4.3.9.1, "Undofile Class Methods".

Public Types. The Undofile class defines no public types.

Class Diagram. This diagram shows all the available methods of the Undofile class:



3.4.3.9.1. Undofile Class Methods

This section details the public methods of the Undofile class.

3.4.3.9.1.1.

Description. The class constructor can be used to create a new Undofile instance, or to copy an existing one.

Signatures. Creates a new instance:

```
Undofile
    (
      void
    )
```

Copy constructor:

```
Undofile
    (
      const Undofile& undoFile
    )
```

Parameters. New instance: *None*. The copy constructor takes a single argument — a reference to the `Undofile` object to be copied.

Return Value. An `Undofile` object.

Destructor. The class defines a virtual destructor which takes no arguments and has the return type `void`.

3.4.3.9.1.2. `Undofile::getPath()`

Description. This method retrieves the path matching the location of the undofile on the data node's filesystem.

Signature.

```
const char* getPath
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The filesystem path, a string (as a character pointer).

3.4.3.9.1.3. `Undofile::getSize()`

Description. This method gets the size of the undofile in bytes.

Signature.

```
Uint64 getSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The size in bytes of the undofile, as an unsigned 64-bit integer.

3.4.3.9.1.4. `Undofile::getLogfileGroup()`

Description. This method retrieves the name of the logfile group to which the undofile belongs.

Signature.

```
const char* getLogfileGroup
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the logfile group, a string value (as a character pointer).

3.4.3.9.1.5. `Undofile::getLogfileGroupId()`

Description. This method retrieves the ID of the logfile group to which the undofile belongs.

> **Note**
>
> It is also possible to obtain the name of the logfile group directly. See Section 3.4.3.9.1.4, "`Undofile::getLogfileGroup()`"

Signature.

```
Uint32 getLogfileGroupId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The ID of the logfile group, as an unsigned 32-bit integer.

3.4.3.9.1.6. `Undofile::getNode()`

Description. This method is used to retrieve the node ID of the node where the undofile is located.

Signature.

```
Uint32 getNode
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The node ID, as an unsigned 32-bit integer.

3.4.3.9.1.7. `Undofile::getFileNo()`

Description. The `getFileNo()` method gets the number of the undofile in the logfile group to which it belongs.

Signature.

```
Uint32 getFileNo
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of the undofile, as an unsigned 32-bit integer.

3.4.3.9.1.8. `Undofile::getObjectStatus()`

Description. This method is used to retrieve the object status of an undofile.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `Object::Status` value — see Section 3.4.3.1.3, "The `Object::Status` Type", for details.

3.4.3.9.1.9. `Undofile::getObjectVersion()`

Description. This method gets the undofile's object version.

Signature.

```
virtual int getObjectVersion
  (
    void
  ) const
```

Parameters. *None*.

Return Value. The object version, as an integer.

3.4.3.9.1.10. `Undofile::getObjectId()`

Description. This method retrieves the undofile's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID, as an integer.

3.4.3.9.1.11. `Undofile::setPath()`

Description. This method is used to set the filesystem path of the undofile on the data node where it resides.

Signature.

```
void setPath
    (
      const char* path
    )
```

Parameters. The desired *`path`* to the undofile.

Return Value. *None*.

3.4.3.9.1.12. `Undofile::setSize()`

Description. Sets the size of the undofile in bytes.

Signature.

```
void setSize
    (
      Uint64 size
    )
```

Parameters. The intended *size* of the undofile in bytes, as an unsigned 64-bit integer.

Return Value. *None*.

3.4.3.9.1.13. Undofile::setLogfileGroup()

Description. Given either a name or an object reference to a logfile group, the setLogfileGroup() method assigns the undofile to that logfile group.

Signature. Using a logfile group name:

```
void setLogfileGroup
    (
      const char* name
    )
```

Using a reference to a LogfileGroup instance:

```
void setLogfileGroup
    (
      const class LogfileGroup & logfileGroup
    )
```

Parameters. The *name* of the logfile group (a character pointer), or a reference to a LogfileGroup instance.

Return Value. *None*.

3.4.3.9.1.14. Undofile::setNode()

Description. Sets the node on which the logfile group is to reside.

Signature.

```
 void setNode
    (
      Uint32 nodeId
    )
```

Parameters. The *nodeId* of the data node where the undofile is to be placed; this is an unsigned 32-bit integer.

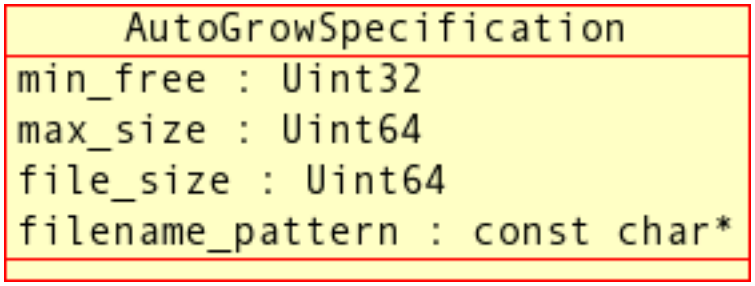
Return Value. *None*.

3.4.4. The AutoGrowSpecification Structure

This section describes the AutoGrowSpecification structure.

Description. The AutoGrowSpecification is a data structure defined in the NdbDictionary class, and is used as a parameter to or return value of some of the methods of the Tablespace and LogfileGroup classes. See Section 3.4.3.8, “The Tablespace Class”, and Section 3.4.3.6, “The LogfileGroup Class”, for more information.

Public Methods. AutoGrowSpecification has the following members, whose types are as shown in the following diagram:



The purpose and use of each member can be found in the following table:

Name	Description
`min_free`	???
`max_size`	???
`file_size`	???
`filename_pattern`	???

3.5. The `NdbEventOperation` Class

This section describes the `NdbEventOperation` class, which is used to monitor changes (events) in a database.

Description. `NdbEventOperation` represents a database event.

Creating an Instance of `NdbEventOperation`. This class has no public constructor or destructor. Instead, instances of `NdbEventOperation` are created as the result of method calls on `Ndb` and `NdbDictionary` objects:

1. There must exist an event which was created using `Dictionary::createEvent()`. This method returns an instance of the `NdbDictionary::Event` class. See Section 3.4.1.1.12, "`Dictionary::createEvent()`".

   An `NdbEventOperation` object is instantiated using `Ndb::createEventOperation()`, which acts on instance of `NdbDictionary::Event`. See Section 3.1.1.10, "`Ndb::createEventOperation`".

See also Section 3.4.3.4, "The `Event` Class".

An instance of this class is removed by invoking `Ndb::dropEventOperation`. See Section 3.1.1.11, "`Ndb::dropEventOperation()`", for more information.

> **Tip**
>
> A detailed example demonstrating creation and removal of event operations is provided in Section 6.6, "NDB API Event Handling Example".

Limitations. The following limitations apply to this class:

- The maximum number of active `NdbEventOperation` objects is currently fixed at compile time at 100. This may become a configurable parameter in a future release.
- The maximum number of `NdbEventOperation` objects associated with a single event in one process is 16.

Known Issues. The following issues may be encountered when working with event operations in the NDB API:

- When several instances of `NdbEventOperation` are tied to the same event in the same process, they share the circular buffer (set as the *`bufferLength`* parameter to `Ndb::createEventOperation()`), which is therefore the same for all such instances and determined when the first one is instantiated. For this reason, you should make sure to instantiate the "largest" instance first.

- Currently, all `INSERT`, `DELETE`, and `UPDATE` events — as well as all attribute changes — are sent to the API, even if only some attributes have been specified. However, these are hidden from the user and only relevant data is shown after calling `Ndb::nextEvent()`.

  Note that false exits from `Ndb::pollEvents()` may occur, and thus the following `nextEvent()` call returns zero, since there was no available data. In such cases, simply call `pollEvents()` again.

  See Section 3.1.1.12, "`Ndb::pollEvents()`", and Section 3.1.1.13, "`Ndb::nextEvent()`".

- Event code does *not* check the table schema version. When a table is dropped, make sure that you drop any associated events.

- On a replicated Cluster, each event is received multiple times — once for each replica. However, if a node fails, events are no longer received multiple times for data in the corresponding fragment.

- Following the failure of a node, not all events will be received any longer. You should drop any associated `NdbEventOperation` objects and re-create them after once the node is running again.

We intend to remedy these issues in future releases of MySQL Cluster and the NDB API.

> **Tip**
>
> To view the contents of the system table containing created events, you can use the `ndb_select_all` utility as shown here:
>
> ```
> ndb_select_all -d sys 'NDB$EVENTS_0'
> ```

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

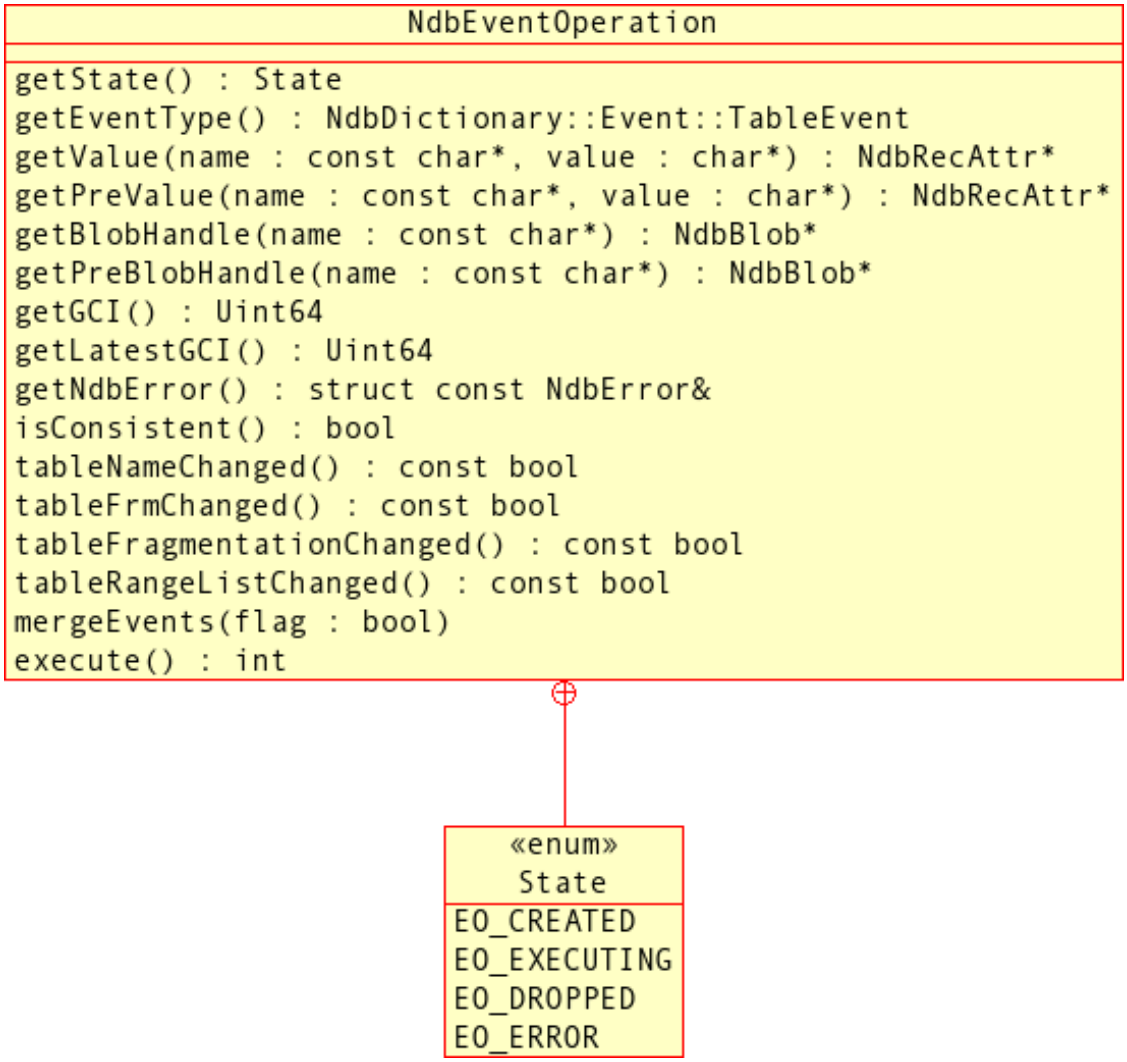
Method	Purpose / Use
`getState()`	Gets the current state of the event operation
`getEventType()`	Gets the event type
`getValue()`	Retrieves an attribute value
`getPreValue()`	Retrieves an attribute's previous value
`getBlobHandle()`	Gets a handle for reading blob attributes
`getPreBlobHandle()`	Gets a handle for reading the previous blob attribute
`getGCI()`	Retrieves the GCI of the most recently retrieved event
`getLatestGCI()`	Retrieves the most recent GCI (whether or not the corresponding event has been retrieved)
`getNdbError()`	Gets the most recent error

Method	Purpose / Use
isConsistent()	Detects event loss caused by node failure
tableNameChanged()	Checks to see whether the name of a table has changed
tableFrmChanged()	Checks to see whether a table .FRM file has changed
tableFragmentationChanged()	Checks to see whether the fragmentation for a table has changed
tableRangeListChanged()	Checks to see whether a table range partition list name has changed
mergeEvents()	Allows for events to be merged
execute()	Activates the NdbEventOperation

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.5.2, "NdbEventOperation Class Methods".

Public Types. NdbEventOperation defines one enumerated type. See Section 3.5.1, "The NdbEventOperation::State Type", for details.

Class Diagram. This diagram shows all the available methods and enumerated types of the NdbEventOperation class:



3.5.1. The `NdbEventOperation::State` Type

Description. This type describes the event operation's state.

Enumeration Values.

Value	Description
`EO_CREATED`	The event operation has been created, but `execute()` has not yet been called.
`EO_EXECUTING`	The `execute()` method has been invoked for this event operation.
`EO_DROPPED`	The event operation is waiting to be deleted, and is no longer usable.
`EO_ERROR`	An error has occurred, and the event operation is unusable.

A `State` value is returned by the `getState()` method. See Section 3.5.2.1, "`NdbEventOperation::getState()`", for more information.

3.5.2. `NdbEventOperation` Class Methods

This section contains definitions and descriptions of the public methods of the `NdbEventOperation` class.

3.5.2.1. `NdbEventOperation::getState()`

Description. This method gets the event operation's current state.

Signature.

```
State getState
    (
      void
    )
```

Parameters. *None*.

Return Value. A `State` value. See Section 3.5.1, "The `NdbEventOperation::State` Type".

3.5.2.2. `NdbEventOperation::getEventType()`

Description. This method is used to obtain the event's type (`TableEvent`).

Signature.

```
NdbDictionary::Event::TableEvent getEventType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `TableEvent` value. See Section 3.4.3.4.1.1, "The `Event::TableEvent` Type".

3.5.2.3. `NdbEventOperation::getValue()`

Description. This method defines the retrieval of an attribute value. The `NDB` API allocates memory for the `NdbRecAttr` object that is to hold the returned attribute value.

> **Important**
>
> This method does *not* fetch the attribute value from the database, and the `NdbRecAttr` object returned by this method is not readable or printable before calling the execute() method and `Ndb::nextEvent()` has returned not `NULL`.
>
> If a specific attribute has not changed, the corresponding `NdbRecAttr` will be in the state `UNDEFINED`. This can be checked by using `NdbRecAttr::isNULL()` which in such cases returns `-1`.

***`value`* Buffer Memory Allocation.** It is the application's responsibility to allocate sufficient memory for the *`value`* buffer (if not `NULL`), and this buffer must be aligned appropriately. The buffer is used directly (thus avoiding a copy penalty) only if it is aligned on a 4-byte boundary and the attribute size in bytes (calculated as `NdbRecAttr::attrSize()` times `NdbRecAttr::arraySize()`) is a multiple of 4.

> **Note**
>
> `getValue()` retrieves the current value. Use `getPreValue()` for retrieving the previous value. See Section 3.5.2.4, "`NdbEventOperation::getPreValue()`".

Signature.

```
NdbRecAttr* getValue
    (
      const char* name,
      char*       value = 0
    )
```

Parameters. This method takes two parameters:

- The *name* of the attribute (as a constant character pointer).
- A pointer to a *value*:
  - If the attribute value is not NULL, then the attribute value is returned in this parameter.
  - If the attribute value is NULL, then the attribute value is stored only in the NdbRecAttr object returned by this method.

  See *value* Buffer Memory Allocation for more information regarding this parameter.

Return Value. An NdbRecAttr object to hold the value of the attribute, or a NULL pointer indicating that an error has occurred. See Section 3.7, "The NdbRecAttr Class".

3.5.2.4. NdbEventOperation::getPreValue()

Description. This method performs identically to getValue(), except that it is used to define a retrieval operation of an attribute's previous value rather than the current value. See Section 3.5.2.3, "NdbEventOperation::getValue()", for details.

Signature.

```
NdbRecAttr* getPreValue
    (
      const char* name,
      char*       value = 0
    )
```

Parameters. This method takes two parameters:

- The *name* of the attribute (as a constant character pointer).
- A pointer to a *value*:
  - If the attribute value is not NULL, then the attribute value is returned in this parameter.
  - If the attribute value is NULL, then the attribute value is stored only in the NdbRecAttr object returned by this method.

  See *value* Buffer Memory Allocation for more information regarding this parameter.

Return Value. An NdbRecAttr object to hold the value of the attribute, or a NULL pointer indicating that an error has occurred. See Section 3.7, "The NdbRecAttr Class".

3.5.2.5. NdbEventOperation::getBlobHandle()

Description. This method is used in place of getValue() for blob attributes. The blob handle (NdbBlob)returned by this method supports read operations only.

> **Note**
>
> To obtain the previous value for a blob attribute, use `getPreBlobHandle`. See Section 3.5.2.6, "`NdbEventOperation::getPreBlobHandle()`".

Signature.

```
NdbBlob* getBlobHandle
    (
      const char* name
    )
```

Parameters. The *`name`* of the blob attribute.

Return Value. A pointer to an `NdbBlob` object. See Section 3.3, "The `NdbBlob` Class".

3.5.2.6. `NdbEventOperation::getPreBlobHandle()`

Description. This function is the same as `getBlobHandle()`, except that it is used to access the previous value of the blob attribute. See Section 3.5.2.5, "`NdbEventOperation::getBlobHandle()`".

Signature.

```
NdbBlob* getPreBlobHandle
  (
    const char* name
  )
```

Parameters. The *`name`* of the blob attribute.

Return Value. A pointer to an `NdbBlob`. See Section 3.3, "The `NdbBlob` Class".

3.5.2.7. `NdbEventOperation::getGCI()`

Description. This method retrieves the GCI for the most recently retrieved event.

Signature.

```
Uint64 getGCI
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The global checkpoint index of the most recently retrieved event (an integer).

3.5.2.8. `NdbEventOperation::getLatestGCI()`

Description. This method retrieves the most recent GCI.

> **Note**
>
> The GCI obtained using this method is not necessarily associated with an event.

Signature.

```
Uint64 getLatestGCI
    (
```

```
      void
    ) const
```

Parameters. *None*.

Return Value. The index of the latest global checkpoint, an integer.

3.5.2.9. NdbEventOperation::getNdbError()

Description. This method retrieves the most recent error.

Signature.

```
const struct NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A reference to an `NdbError` structure. See Section 4.1, "The `NdbError` Structure".

3.5.2.10. NdbEventOperation::isConsistent()

Description. This method is used to determine whether event loss has taken place following the failure of a node.

Signature.

```
bool isConsistent
    (
      void
    ) const
```

Parameters. *None*.

Return Value. If event loss has taken place, then this method returns `false`; otherwise, it returns `true`.

3.5.2.11. NdbEventOperation::tableNameChanged()

Description. This method tests whether a table name has changed as the result of a `TE_ALTER` table event. (See Section 3.4.3.4.1.1, "The `Event::TableEvent` Type".)

Signature.

```
const bool tableNameChanged
    (
      void
    ) const
```

Parameters. *None*.

Return Value. Returns `true` if the name of the table has changed; otherwise, the method returns `false`.

3.5.2.12. NdbEventOperation::tableFrmChanged()

Description. Use this method to determine whether a table `.FRM` file has changed in connection with a

TE_ALTER event. (See Section 3.4.3.4.1.1, “The Event::TableEvent Type”.)

Signature.

```
const bool tableFrmChanged
    (
      void
    ) const
```

Parameters. *None*.

Return Value. Returns `true` if the table `.FRM` file has changed; otherwise, the method returns `false`.

3.5.2.13. NdbEventOperation::tableFragmentationChanged()

Description. This method is used to test whether a table's fragmentation has changed in connection with a `TE_ALTER` event. (See Section 3.4.3.4.1.1, “The Event::TableEvent Type”.)

Signature.

```
const bool tableFragmentationChanged
    (
      void
    ) const
```

Parameters. *None*.

Return Value. Returns `true` if the table's fragmentation has changed; otherwise, the method returns `false`.

3.5.2.14. NdbEventOperation::tableRangeListChanged()

Description. Use this method to check whether a table range partition list name has changed in connection with a `TE_ALTER` event.

Signature.

```
const bool tableRangeListChanged
    (
      void
    ) const
```

Parameters. *None*.

Return Value. This method returns `true` if range or list partition name has changed; otherwise it returns `false`.

3.5.2.15. NdbEventOperation::mergeEvents()

Description. This method is used to set the merge events flag. For information about event merging, see Section 3.4.3.4.2.20, “Event::mergeEvents()”.

> **Note**
>
> The merge events flag is `false` by default.

Signature.

```
void mergeEvents
    (
```

```
      bool flag
    )
```

Parameters. A Boolean *`flag`*.

Return Value. *None*.

3.5.2.16. `NdbEventOperation::execute()`

Description. Activates the `NdbEventOperation`, so that it can begin receiving events. Changed attribute values may be retrieved after `Ndb::nextEvent()` has returned not `NULL`. `getValue()` or `getPreValue()` must be called before invoking `execute()`.

> **Important**
>
> Before attempting to use this method, you should have read the explanations provided in Section 3.1.1.13, "`Ndb::nextEvent()`", and Section 3.5.2.3, "`NdbEventOperation::getValue()`". Also see Section 3.5, "The `NdbEventOperation` Class".

Signature.

```
int execute
    (
      void
    )
```

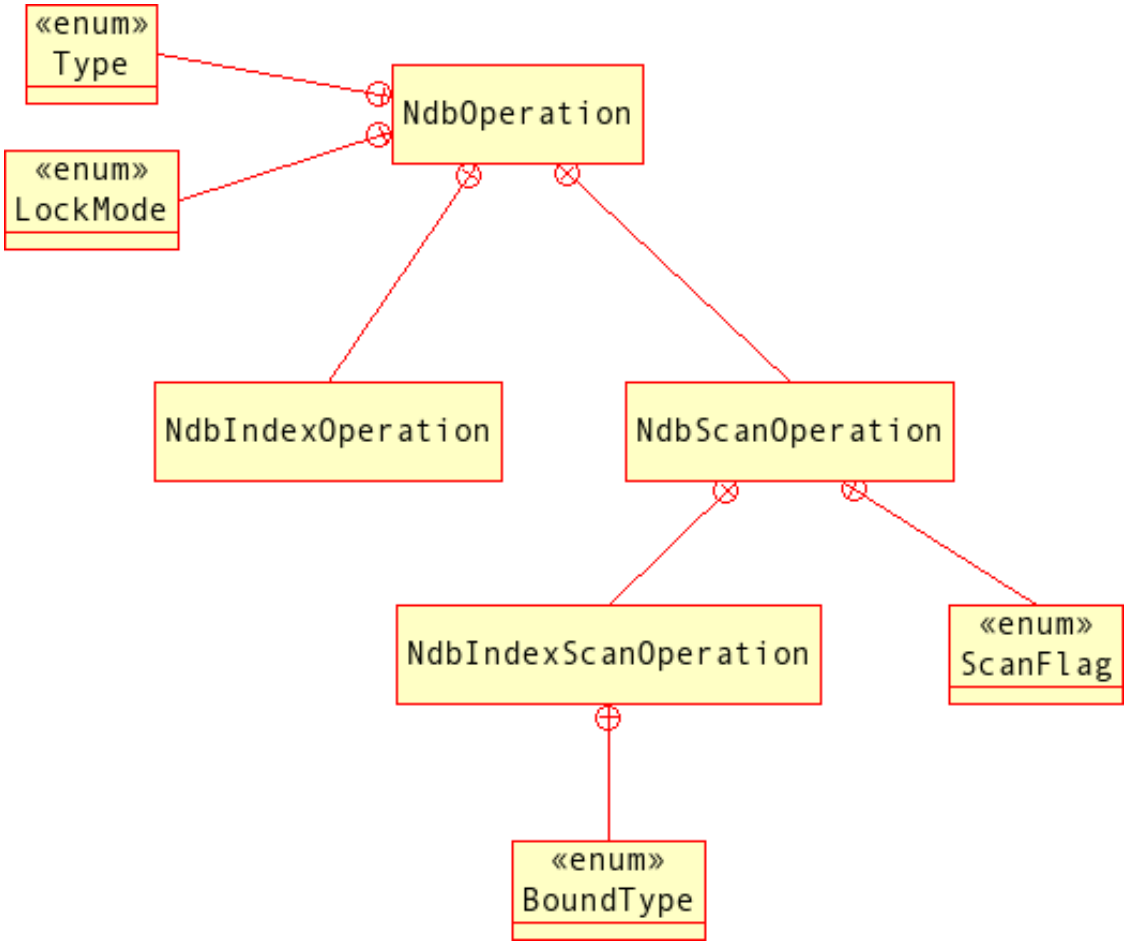
Parameters. *None*.

Return Value. This method returns `0` on success and `-1` on failure.

3.6. The `NdbOperation` Class

This section discusses the `NdbOperation` class. Its subclasses `NdbIndexOperation`, `NdbScanOperation`, and `NdbIndexScanOperation` are described in subsections of this section.

`NdbOperation` Subclasses. This diagram shows the relationships of `NdbOperation`, it subclasses, and their public types:



Description. `NdbOperation` represents a "generic" data operation. Its subclasses represent more specific types of operations. See Section 3.6.1.1, "The `NdbOperation::Type` Type" for a listing of operation types and their corresponding `NdbOperation` subclasses.

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`getValue()`	Prepares an attribute value for access
`getBlobHandle()`	Used to access blob attributes
`getTableName()`	Gets the name of the table used for this operation
`getTable()`	Gets the table object used for this operation
`getNdbError()`	Gets the latest error
`getNdbErrorLine()`	Gets the number of the method where the latest error occurred
`getType()`	Gets the type of operation
`getLockMode()`	Gets the operation's lock mode
`equal()`	Defines a search condition using equality
`setValue()`	Defines an attribute to set or update
`insertTuple()`	Adds a new tuple to a table
`updateTuple()`	Updates an existing tuple in a table
`readTuple()`	Reads a tuple from a table

Method	Purpose / Use
writeTuple()	Inserts or updates a tuple
deleteTuple()	Removes a tuple from a table

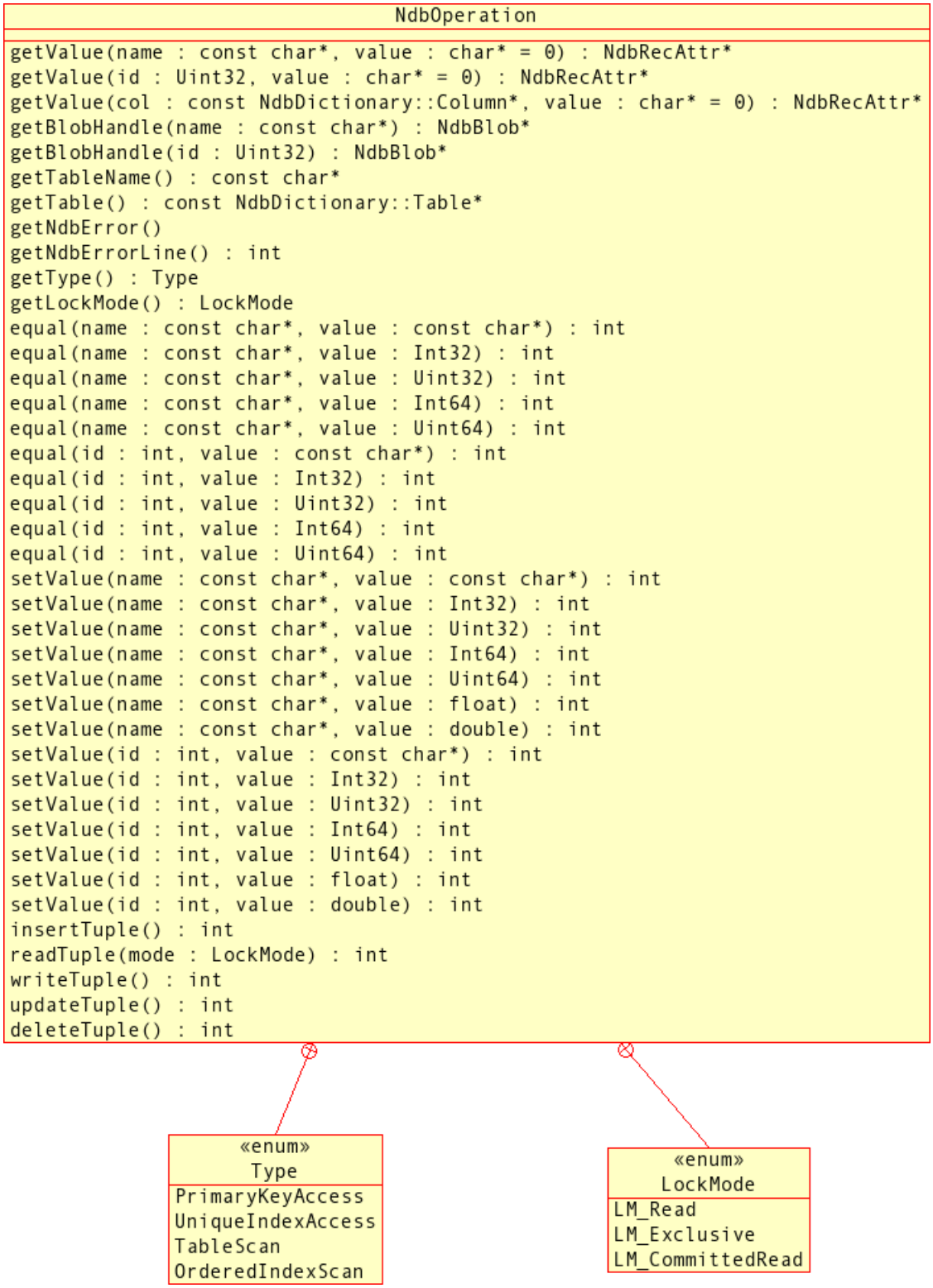
For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.6.2, "NdbOperation Class Methods".

Public Types. The NdbOperation class defines two public types, as shown in the following table:

Type	Purpose / Use
Type	Operation access types
LockMode	The type of lock used when performing a read operation

For a discussion of each of these types, along with its possible values, see Section 3.6.1, "NdbOperation Types".

Class Diagram. This diagram shows all the available methods and enumerated types of the NdbOperation class:



Note

For more information about the use of `NdbOperation`, see Section 1.3.2.3.1, “Single-row operations”.

3.6.1. NdbOperation Types

This section details the public types belonging to the NdbOperation class.

3.6.1.1. The NdbOperation::Type Type

Description. Type is used to describe the operation access type. Each access type is supported by NdbOperation or one of its subclasses, as shown in the following table:

Enumeration Values.

Value	Description	Class
PrimaryKeyAccess	A read, insert, update, or delete operation using the table's primary key	NdbOperation
UniqueIndexAccess	A read, update, or delete operation using a unique index	NdbIndexOperation
TableScan	A full table scan	NdbScanOperation
OrderedIndexScan	An ordered index scan	NdbIndexScanOperation

3.6.1.2. The NdbOperation::LockMode Type

Description. This type describes the lock mode used when performing a read operation.

Enumeration Values.

Value	Description
LM_Read	Read with shared lock
LM_Exclusive	Read with exclusive lock
LM_CommittedRead	Ignore locks; read last committed

> **Note**
>
> There is also support for dirty reads (LM_Dirty), but this is normally for internal purposes only, and should not be used for applications deployed in a production setting.

3.6.2. NdbOperation Class Methods

This section lists and describes the public methods of the NdbOperation class.

> **Note**
>
> This class has no public constructor. To create an instance of NdbOperation, you must use NdbTransaction::getNdbOperation(). See Section 3.9.2.1, “NdbTransaction::getNdbOperation()”, for more information.

3.6.2.1. NdbOperation::getValue()

Description. This method defines the retrieval of an attribute value. The NDB API allocates memory for the NdbRecAttr object that is to hold the returned attribute value.

> **Important**

> This method does *not* fetch the attribute value from the database, and the `NdbRecAttr` object returned by this method is not readable or printable before calling `NdbTransaction::execute()`.
>
> If a specific attribute has not changed, the corresponding `NdbRecAttr` will be in the state `UNDEFINED`. This can be checked by using `NdbRecAttr::isNULL()` which in such cases returns `-1`.
>
> See Section 3.9.2.5, "`NdbTransaction::execute()`", and Section 3.7.1.4, "`NdbRecAttr::isNULL()`".

Signature. There are three versions of this method, each having different parameters:

```
NdbRecAttr* getValue
    (
      const char* name,
      char*       value = 0
    )

NdbRecAttr* getValue
    (
      Uint32 id,
      char*  value = 0
    )

NdbRecAttr* getValue
    (
      const NdbDictionary::Column* col,
      char*                        value = 0
    )
```

Parameters. All three forms of this method require two parameters. They differ with regard to the type of the first parameter, which can be any one of the following:

- The attribute *`name`*
- The attribute *`id`*
- The *`column`* on which the attribute is defined

In all three cases, the second parameter is a character buffer in which a non-`NULL` attribute value is returned. In the event that the attribute is `NULL`, is it stored only in the `NdbRecAttr` object returned by this method.

Return Value. An `NdbRecAttr` object to hold the value of the attribute, or a `NULL` pointer, indicating an error.

3.6.2.2. `NdbOperation::getBlobHandle()`

Description. This method is used in place of `getValue()` or `setValue()` for blob attributes. It creates a blob handle (`NdbBlob` object). A second call with the same argument returns the previously created handle. The handle is linked to the operation and is maintained automatically. See Section 3.3, "The `NdbBlob` Class", for details.

Signature. This method has two forms, depending on whether it is called with the name or the ID of the blob attribute:

```
virtual NdbBlob* getBlobHandle
    (
      const char* name
    )

virtual NdbBlob* getBlobHandle
    (
```

```
      Uint32 id
    )
```

Parameters. This method takes a single parameter, which can be either one of the following:

- The *`name`* of the attribute
- The *`id`* of the attribute

Return Value. Regardless of parameter type used, this method return a pointer to an instance of `NdbBlob`.

3.6.2.3. `NdbOperation::getTableName()`

Description. This method retrieves the name of the table used for the operation.

Signature.

```
const char* getTableName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the table.

3.6.2.4. `NdbOperation::getTable()`

Description. This method is used to retrieve the table object associated with the operation.

Signature.

```
const NdbDictionary::Table* getTable
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An instance of `Table`. For more information, see Section 3.4.3.7, "The `Table` Class".

3.6.2.5. `NdbOperation::getNdbError()`

Description. This method gets the most recent error (an `NdbError` object).

Signature.

```
const NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `NdbError` object. See Section 4.1, "The `NdbError` Structure".

3.6.2.6. NdbOperation::getNdbErrorLine()

Description. This method retrieves the method number in which the latest error occurred.

Signature.

```
int getNdbErrorLine
    (
      void
    )
```

Parameters. *None*.

Return Value. The method number (an integer).

3.6.2.7. NdbOperation::getType()

Description. This method is used to retrieve the access type for this operation.

Signature.

```
const Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `Type` value. See Section 3.6.1.1, "The NdbOperation::Type Type".

3.6.2.8. NdbOperation::getLockMode()

Description. This method gets the operation's lock mode.

Signature.

```
LockMode getLockMode
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `LockMode` value. See Section 3.6.1.2, "The NdbOperation::LockMode Type".

3.6.2.9. NdbOperation::equal()

Description. This method define a search condition with an equality. The condition is true if the attribute has the given value. To set search conditions on multiple attributes, use several calls to `equal()`; in such cases all of them must be satisfied for the tuple to be selected.

> **Important**
>
> If the attribute is of a fixed size, its value must include all bytes. In particular a `Char` value must be native-blank padded. If the attribute is of variable size, its value must start with 1 or 2 little-endian length bytes (2 if its type is `Long*`).

> **Note**

When using `insertTuple()`, you may also define the search key with `setValue()`. See Section 3.6.2.10, “`NdbOperation::setValue()`”.

Signature. There are 10 versions of `equal()`, each having slightly different parameters. All of these are listed here:

```
int  equal
    (
      const char* name,
      const char* value
    )

int  equal
    (
      const char* name,
      Int32       value
    )

int  equal
    (
      const char* name,
      Uint32      value
    )

int  equal
    (
      const char* name,
      Int64       value
    )

int  equal
    (
      const char* name,
      Uint64      value
    )

int  equal
    (
      Uint32      id,
      const char* value
    )

int  equal
    (
      Uint32 id,
      Int32  value
    )

int  equal
    (
      Uint32 id,
      Uint32 value
    )

int  equal
    (
      Uint32 id,
      Int64  value
    )

int  equal
    (
      Uint32 id,
      Uint64 value
    )
```

Parameters. This method requires two parameters:

- The first parameter can be either of the following:
  - The *`name`* of the attribute (a string)
  - The *`id`* of the attribute (an unsigned 32-bit integer)

- The second parameter is the attribute *value* to be tested; it can be any one of the following 5 types:
  - String
  - 32-bit integer
  - Unsigned 32-bit integer
  - 64-bit integer
  - Unsigned 64-bit integer

Return Value. Returns -1 in the event of an error.

3.6.2.10. NdbOperation::setValue()

Description. This method defines an attribute to be set or updated.

> **Important**
>
> There are a number of NdbOperation::setValue() methods that take a certain type as input (pass by value rather than passing a pointer). It is the responsibility of the application programmer to use the correct types.
>
> However, the NDB API does check that the application sends a correct length to the interface as given in the length parameter. A char* value can contain any datatype or any type of array. If the length is not provided, or if it is set to zero, then the API assumes that the pointer is correct, and does not check it.

> **Tip**
>
> To set a NULL value, use the following construct:
>
> ```
> setValue("ATTR_NAME", (char*)NULL);
> ```

> **Note**
>
> When you use insertTuple(), the NDB API will automatically detect that it is supposed to use equal() instead.
>
> In addition, it is not necessary when using insertTuple() to use setValue() on key attributes before other attributes.

Signature. There are 14 versions of NdbOperation::setValue(), each with slightly different parameters, as listed here (and summarised in the *Parameters* section following):

```
int setValue
    (
      const char* name,
      const char* value
)

int setValue
    (
      const char* name,
      Int32       value
    )

int setValue
    (
      const char* name,
```

```
      Uint32      value
    )

int setValue
    (
      const char* name,
      Int64       value
    )

int setValue
    (
      const char* name,
      Uint64      value
    )

int setValue
    (
      const char* name,
      float       value
    )

int setValue
    (
      const char* name,
      double      value
    )

int setValue
    (
      Uint32      id,
      const char* value
    )

int setValue
    (
      Uint32 id,
      Int32  value
    )

int setValue
    (
      Uint32 id,
      Uint32 value
    )

int setValue
    (
      Uint32 id,
      Int64  value
    )

int setValue
    (
      Uint32 id,
      Uint64 value
    )

int setValue
    (
      Uint32 id,
      float  value
    )

int setValue
    (
      Uint32 id,
      double value
    )
```

Parameters. This method requires two parameters:

- The first parameter identified the attribute to be set, and may be either one of:
  - The attribute *`name`* (a string)
  - The attribute *`id`* (an unsigned 32-bit integer)

- The second parameter is the *value* to which the attribute is to be set; its type may be any one of the following 7 types:
  - String (const char*)
  - 32-bit integer
  - Unsigned 32-bit integer
  - 64-bit integer
  - Unsigned 64-bit integer
  - Double
  - Float

See Section 3.6.2.9, "NdbOperation::equal()", for important information regarding the value's format and length.

Return Value. Returns -1 in the event of failure.

3.6.2.11. NdbOperation::insertTuple()

Description. This method defines the NdbOperation to be an INSERT operation. When the NdbTransaction::execute() method is called, this operation adds a new tuple to the table. See Section 3.9.2.5, "NdbTransaction::execute()".

Signature.

```
virtual int insertTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. 0 on success, -1 on failure.

3.6.2.12. NdbOperation::readTuple()

Description. This method define the NdbOperation as a READ operation. When the NdbTransaction::execute() method is invoked, the operation reads a tuple. See Section 3.9.2.5, "NdbTransaction::execute()".

Signature.

```
virtual int readTuple
    (
      LockMode mode
    )
```

Parameters. *mode* specifies the locking mode used by the read operation. See Section 3.6.1.2, "The NdbOperation::LockMode Type", for possible values.

Return Value. 0 on success, -1 on failure.

3.6.2.13. NdbOperation::writeTuple()

Description. This method defines the `NdbOperation` as a `WRITE` operation. When the `NdbTransaction::execute()` method is invoked, the operation writes a tuple to the table. If the tuple already exists, it is updated; otherwise an insert takes place. See Section 3.9.2.5, "`NdbTransaction::execute()`".

Signature.

```
virtual int writeTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

3.6.2.14. `NdbOperation::updateTuple()`

Description. This method defines the `NdbOperation` as an `UPDATE` operation. When the `NdbTransaction::execute()` method is invoked, the operation updates a tuple found in the table. See Section 3.9.2.5, "`NdbTransaction::execute()`".

Signature.

```
virtual int updateTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

3.6.2.15. `NdbOperation::deleteTuple()`

Description. This method defines the `NdbOperation` as a `DELETE` operation. When the `NdbTransaction::execute()` method is invoked, the operation deletes a tuple from the table. See Section 3.9.2.5, "`NdbTransaction::execute()`".

Signature.

```
virtual int deleteTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

3.6.3. The `NdbIndexOperation` Class

This section describes the `NdbIndexOperation` class and its public methods.

Description. `NdbIndexOperation` represents an index operation for use in transactions. This class inherits from `NdbOperation`; see Section 3.6, "The `NdbOperation` Class", for more information.

Note

> `NdbIndexOperation` can be used only with unique hash indexes; to work with ordered indexes, use `NdbIndexScanOperation`. See Section 3.6.4.3, "The `NdbIndexScanOperation` Class".

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

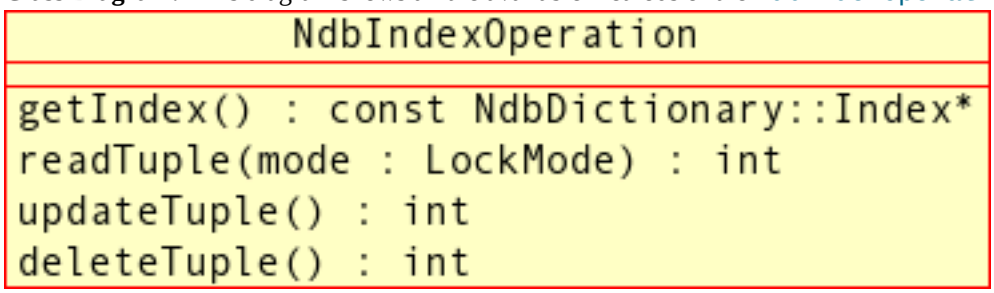
Method	Purpose / Use
`getIndex()`	Gets the index used by the operation
`readTuple()`	Reads a tuple from a table
`updateTuple()`	Updates an existing tuple in a table
`deleteTuple()`	Removes a tuple from a table

> **Note**
>
> Index operations are not permitted to insert tuples.

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.6.2, "`NdbOperation` Class Methods".

Public Types. The `NdbIndexOperation` class defines no public types of its own.

Class Diagram. This diagram shows all the available methods of the `NdbIndexOperation` class:



> **Note**
>
> For more information about the use of `NdbIndexOperation`, see Section 1.3.2.3.1, "Single-row operations".

3.6.3.1. `NdbIndexOperation` Class Methods

This section lists and describes the public methods of the `NdbIndexOperation` class.

> **Note**
>
> This class has no public constructor. To create an instance of `NdbIndexOperation`, it is necessary to use the `NdbTransaction::getNdbIndexOperation()` method. See Section 3.9.2.4, "`NdbTransaction::getNdbIndexOperation()`".

3.6.3.1.1. `NdbIndexOperation::getIndex()`

Description.

Signature.

```
const NdbDictionary::Index* getIndex
```

```
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to an `Index` object. See Section 3.4.3.5, “The `Index` Class”.

3.6.3.1.2. `NdbIndexOperation::readTuple()`

Description. This method define the `NdbIndexOperation` as a `READ` operation. When the `NdbTransaction::execute()` method is invoked, the operation reads a tuple. See Section 3.9.2.5, “`NdbTransaction::execute()`”.

Signature.

```
int readTuple
    (
      LockMode mode
    )
```

Parameters. *`mode`* specifies the locking mode used by the read operation. See Section 3.6.1.2, “The `NdbOperation::LockMode` Type”, for possible values.

Return Value. `0` on success, `-1` on failure.

3.6.3.1.3. `NdbIndexOperation::updateTuple()`

Description. This method defines the `NdbIndexOperation` as an `UPDATE` operation. When the `NdbTransaction::execute()` method is invoked, the operation updates a tuple found in the table. See Section 3.9.2.5, “`NdbTransaction::execute()`”.

Signature.

```
int writeTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

3.6.3.1.4. `NdbIndexOperation::deleteTuple()`

Description. This method defines the `NdbIndexOperation` as a `DELETE` operation. When the `NdbTransaction::execute()` method is invoked, the operation deletes a tuple from the table. See Section 3.9.2.5, “`NdbTransaction::execute()`”.

Signature.

```
int deleteTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

3.6.4. The NdbScanOperation Class

This section describes the NdbScanOperation class and its class members.

Description. The NdbScanOperation class represents a scanning operation used in a transaction. This class inherits from NdbOperation. For more information, see Section 3.6, "The NdbOperation Class".

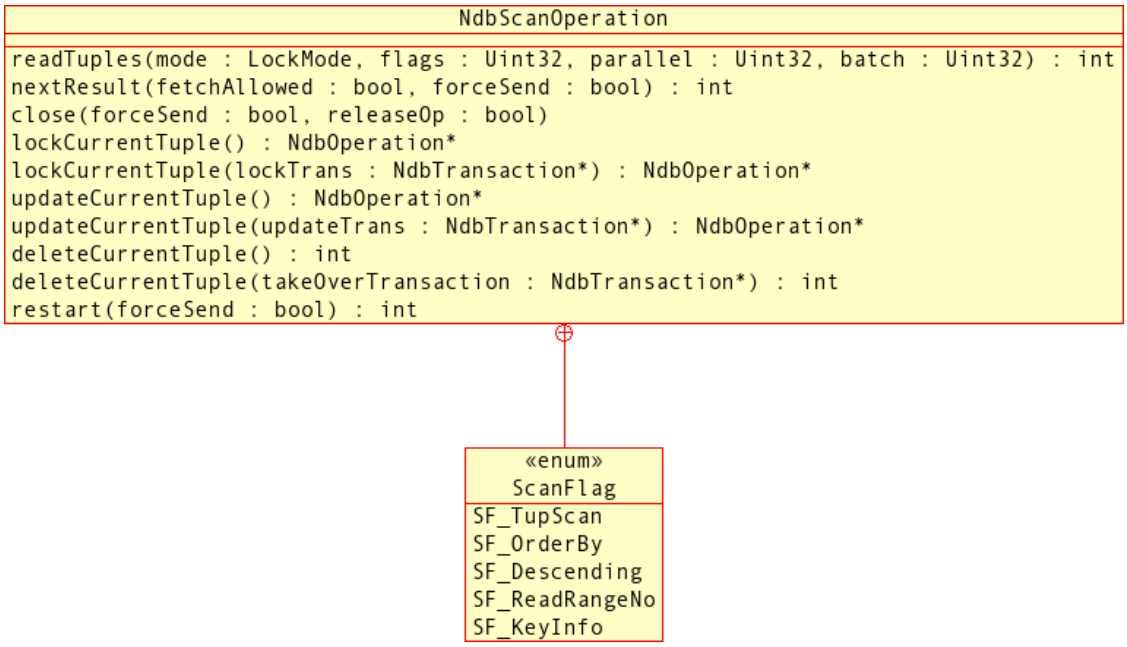
Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
readTuples()	Reads tuples
nextResult()	Gets the next tuple
close()	Closes the scan
lockCurrentTuple()	Locks the current tuple
updateCurrentTuple()	Updates the current tuple
deleteCurrentTuple()	Deletes the current tuple
restart()	Restarts the scan

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.6.4.2, "NdbScanOperation Class Methods".

Public Types. This class defines a single public type ScanFlag. See Section 3.6.4.1, "The NdbScanOperation::ScanFlag Type", for details.

Class Diagram. This diagram shows all the available members of the NdbScanOperation class:



> **Note**
>
> For more information about the use of NdbScanOperation, see Section 1.3.2.3.2, "Scan Operations", and Section 1.3.2.3.3, "Using Scans to Update or Delete Rows"

3.6.4.1. The NdbScanOperation::ScanFlag Type

Description. Values of this type are the scan flags used with the readTuples() method. More than one may be used, in which case, they are OR'ed together as the second argument to that method. See Section 3.6.4.2.1, "NdbScanOperation::readTuples()", for more information.

Enumeration Values.

Value	Description
SF_TupScan	TUP scan
SF_OrderBy	Ordered index scan (ascending)
SF_Descending	Ordered index scan (descending)
SF_ReadRangeNo	Enables NdbIndexScanOperation::get_range_no()
SF_KeyInfo	Requests KeyInfo to be sent back to the caller

3.6.4.2. NdbScanOperation Class Methods

This section lists and describes the public methods of the NdbScanOperation class.

> **Note**
>
> This class has no public constructor. To create an instance of NdbScanOperation, it is necessary to use the NdbTransaction::getNdbScanOperation() method. See Section 3.9.2.2, "NdbTransaction::getNdbScanOperation()".

3.6.4.2.1. NdbScanOperation::readTuples()

Description. This method is used to perform a scan.

Signature.

```
virtual int readTuples
    (
      LockMode mode = LM_Read,
      Uint32   flags = 0,
      Uint32   parallel = 0,
      Uint32   batch = 0
    )
```

Parameters. This method takes four parameters, as shown here:

- The lock *mode*; this is a LockMode value as described in Section 3.6.1.2, "The NdbOperation::LockMode Type".
- One or more ScanFlag values. Multiple values are OR'ed together
- The number of fragments to scan in *parallel*; use 0 to require that the maximum possible number be used.
- The *batch* parameter specifies how many records will be returned to the client from the server by the next NdbScanOperation::nextResult(true) method call. Use 0 to specify the maximum automatically.

> **Note**

> This parameter was ignored prior to MySQL 5.1.12, and the maximum was used. (Bug#20252 [http://bugs.mysql.com/20252])

Return Value. `0` on success, `-1` on failure.

3.6.4.2.2. `NdbScanOperation::nextResult()`

Description. This method is used to fetch the next tuple in a scan transaction. Following each call to `nextResult()`, the buffers and `NdbRecAttr` objects defined in `NdbOperation::getValue()` are updated with values from the scanned tuple.

Signature.

```
int nextResult
    (
      bool fetchAllowed = true,
      bool forceSend = false
    )
```

Parameters. This method takes two parameters:

- Normally, the `NDB` API contacts the `NDB` kernel for more tuples whenever it is necessary; setting *`fetchAllowed`* to `false` keeps this from happening.

  Disabling *`fetchAllowed`* by setting it to `false` forces `NDB` to process any records it already has in its caches. When there are no more cached records it returns `2`. You must then call `nextResult()` with *`fetchAllowed`* equal to `true` in order to contact `NDB` for more records.

  While `nextResult(false)` returns `0`, you should transfer the record to another transaction. When `nextResult(false)` returns `2`, you must execute and commit the other transaction. This causes any locks to be transferred to the other transaction, updates or deletes to be made, and then, the locks to be released. Following this, call `nextResult(true)` — this fetches more records and caches them in the `NDB` API.

  > **Note**
  >
  > If you do not transfer the records to another transaction, the locks on those records will be released the next time that the `NDB` Kernel is contacted for more records.

  Disabling *`fetchAllowed`* can be useful when you want to update or delete all of the records obtained in a given transaction, as doing so saves time and speeds up updates or deletes of scanned records.

- *`forceSend`* defaults to `false`, and can normally be omitted. However, setting this parameter to `true` means that transactions are sent immediately. See Section 1.3.4, “The Adaptive Send Algorithm”, for more information.

Return Value. This method returns one of the following 4 integer values:

- `-1`: Indicates that an error has occurred.
- `0`: Another tuple has been received.
- `1`: There are no more tuples to scan.

- 2: There are no more cached records (invoke `nextResult(true)` to fetch more records).

3.6.4.2.3. `NdbScanOperation::close()`

Description. Calling this method closes a scan.

Signature.

```
void close
    (
      bool forceSend = false,
      bool releaseOp = false
    )
```

Parameters. This method takes two parameters:

- *`forceSend`* defaults to `false`; call `close()` with this parameter set to `true` in order to force transactions to be sent.
- *`releaseOp`* also defaults to `false`; set to `true` in order to release the operation.

Return Value. *None*.

3.6.4.2.4. `NdbScanOperation::lockCurrentTuple()`

Description. This method locks the current tuple.

Signature.

```
NdbOperation* lockCurrentTuple
    (
      void
    )
```

or

```
NdbOperation* lockCurrentTuple
    (
      NdbTransaction* lockTrans
    )
```

Parameters. This method takes a single, optional parameter — the transaction that should perform the lock. If this is omitted, the transaction is the current one.

Return Value. This method returns a pointer to an `NdbOperation` object, or `NULL`. (See Section 3.6, “The `NdbOperation` Class”.)

3.6.4.2.5. `NdbScanOperation::updateCurrentTuple()`

Description. This method is used to update the current tuple.

Signature.

```
NdbOperation* updateCurrentTuple
    (
      void
    )
```

or

```
NdbOperation* updateCurrentTuple
    (
      NdbTransaction* updateTrans
    )
```

Parameters. This method takes a single, optional parameter — the transaction that should perform the lock. If this is omitted, the transaction is the current one.

Return Value. This method returns an `NdbOperation` object or `NULL`. (See Section 3.6, "The `NdbOperation` Class".)

3.6.4.2.6. `NdbScanOperation::deleteCurrentTuple()`

Description. This method is used to delete the current tuple.

Signature.

```
int deleteCurrentTuple
    (
      void
    )
```

or

```
int deleteCurrentTuple
    (
      NdbTransaction* takeOverTransaction
    )
```

Parameters. This method takes a single, optional parameter — the transaction that should perform the lock. If this is omitted, the transaction is the current one.

Return Value. `0` on success, `-1` on failure.

3.6.4.2.7. `NdbScanOperation::restart()`

Description. Use this method to restart a scan without changing any of its `getValue()` calls or search conditions.

Signature.

```
int restart
    (
      bool forceSend = false
    )
```

Parameters. Call this method with *`forceSend`* set to `true` in order to force the transaction to be sent.

Return Value. `0` on success, `-1` on failure.

3.6.4.3. The `NdbIndexScanOperation` Class

This section discusses the `NdbIndexScanOperation` class and its public members.

Description. The `NdbIndexScanOperation` class represents a scan operation using an ordered index. This class inherits from `NdbScanOperation` and `NdbOperation`. See Section 3.6.4, "The `NdbScanOperation` Class", and Section 3.6, "The `NdbOperation` Class", for more information about these classes.

Note

`NdbIndexScanOperation` is for use with ordered indexes only; to work with unique hash indexes, use `NdbIndexOperation`.

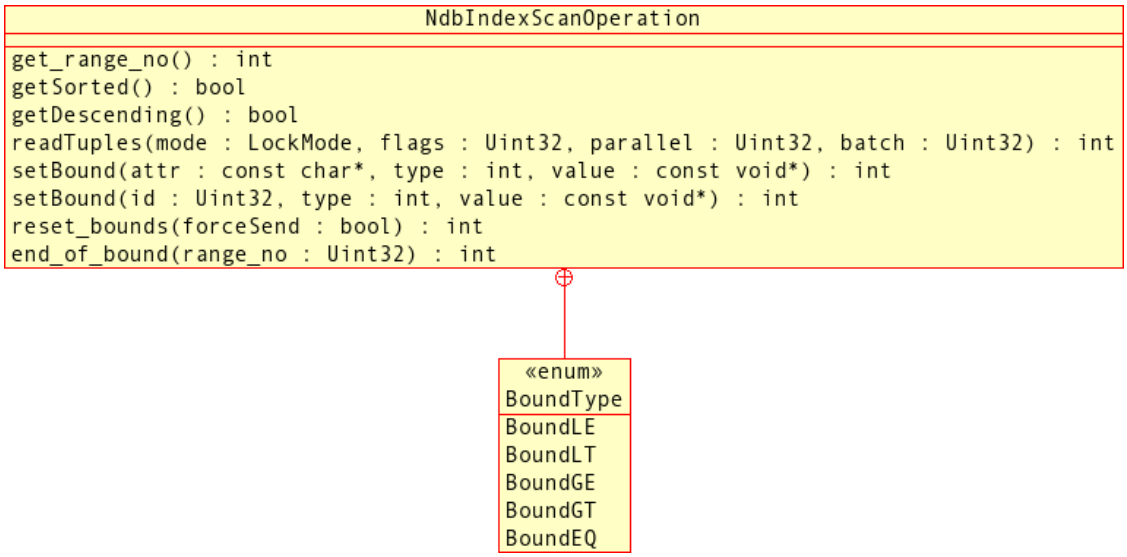
Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`get_range_no()`	Gets the range number for the current row
`getSorted()`	Checks whether the current scan is sorted
`getDescending()`	Checks whether the current scan is sorted
`readTuples()`	Reads tuples using an ordered index
`setBound()`	Defines a bound on the index key for a range scan
`reset_bounds()`	Resets bounds, puts the operation in the send queue
`end_of_bound()`	Marks the end of a bound

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.6.4.3.2, "`NdbIndexScanOperation` Class Methods".

Public Types. The `NdbIndexScanOperation` class defines one public type. See Section 3.6.4.3.1, "The `NdbIndexScanOperation::BoundType` Type".

Class Diagram. This diagram shows all the public members of the `NdbIndexScanOperation` class:



Note

For more information about the use of `NdbIndexScanOperation`, see Section 1.3.2.3.2, "Scan Operations", and Section 1.3.2.3.3, "Using Scans to Update or Delete Rows"

3.6.4.3.1. The `NdbIndexScanOperation::BoundType` Type

Description. This type is used to describe an ordered key bound.

Tip

The numeric values are fixed in the API and can be used explicitly — in other words, it is "safe" to calculate the values and use them.

Enumeration Values.

Value	Numeric Value	Description
BoundLE	0	Lower bound
BoundLT	1	Strict lower bound
BoundGE	2	Upper bound
BoundGT	3	Strict upper bound
BoundEQ	4	Equality

3.6.4.3.2. NdbIndexScanOperation Class Methods

This section lists and describes the public methods of the NdbIndexScanOperation class.

3.6.4.3.2.1. NdbIndexScanOperation::get_range_no()

Description. This method returns the range number for the current row.

Signature.

```
int get_range_no
    (
      void
    )
```

Parameters. *None*.

Return Value. The range number (an integer).

3.6.4.3.2.2. NdbIndexScanOperation::getSorted()

Description. This method is used to check whether the scan is sorted.

Signature.

```
bool getSorted
    (
      void
    ) const
```

Parameters. *None*.

Return Value. true if the scan is sorted, otherwise false.

3.6.4.3.2.3. NdbIndexScanOperation::getDescending()

Description. This method is used to check whether the scan is descending.

Signature.

```
bool getDescending
    (
      void
    ) const
```

Parameters. *None*.

Return Value. This method returns `true` if the scan is sorted in descending order; otherwise, it returns `false`.

3.6.4.3.2.4. `NdbIndexScanOperation::readTuples()`

Description. This method is used to read tuples, using an ordered index.

Signature.

```
virtual int readTuples
    (
      LockMode mode = LM_Read,
      Uint32   flags = 0,
      Uint32   parallel = 0,
      Uint32   batch = 0

    )
```

Parameters. The `readTuples()` method takes 3 parameters, as listed here:

- The lock *`mode`* used for the scan. This is a `LockMode` value; see Section 3.6.1.2, "The `NdbOperation::LockMode` Type" for more information, including permitted values.
- One or more scan flags; multiple *`flags`* are `OR`'ed together as they are when used with `NdbScanOperation::readTuples()`. See Section 3.6.4.1, "The `NdbScanOperation::ScanFlag` Type" for possible values.
- The number of fragments to scan in *`parallel`*; use `0` to specify the maximum automatically.
- The *`batch`* parameter specifies how many records will be returned to the client from the server by the next `NdbScanOperation::nextResult(true)` method call. Use `0` to specify the maximum automatically.

  > **Note**
  >
  > This parameter was ignored prior to MySQL 5.1.12, and the maximum was used.(Bug#20252 [http://bugs.mysql.com/20252])

Return Value. An integer: `0` indicates success; `-1` indicates failure.

3.6.4.3.2.5. `NdbIndexScanOperation::setBound`

Description. This method defines a bound on an index key used in a range scan.

Each index key can have a lower bound, upper bound, or both. Setting the key equal to a value defines both upper and lower bounds. Bounds can be defined in any order. Conflicting definitions gives rise to an error.

Bounds must be set on initial sequences of index keys, and all but possibly the last bound must be non-strict. This means, for example, that "a >= 2 AND b > 3" is permissible, but "a > 2 AND b >= 3" is not.

The scan may currently return tuples for which the bounds are not satisfied. For example, a `<= 2 && b <= 3` not only scans the index up to `(a=2, b=3)`, but also returns any `(a=1, b=4)` as well.

When setting bounds based on equality, it is better to use `BoundEQ` instead of the equivalent pair `BoundLE` and `BoundGE`. This is especially true when the table partition key is a prefix of the index key.

`NULL` is considered less than any non-`NULL` value and equal to another `NULL` value. To perform comparisons with `NULL`, use `setBound()` with a null pointer (`0`).

An index also stores all-`NULL` keys as well, and performing an index scan with an empty bound set returns all tuples from the table.

Signature.

```
int setBound
    (
      const char* name,
      int         type,
      const void* value
    )
```

or

```
int setBound
    (
      Uint32      id,
      int         type,
      const void* value
    )
```

Parameters. This method takes 3 parameters:

- Either the *`name`* or the *`id`* of the attribute on which the bound is to be set.
- The bound `type` — see Section 3.6.4.3.1, “The `NdbIndexScanOperation::BoundType` Type”.
- A pointer to the bound *`value`* (use `0` for `NULL`).

Return Value. Returns `0` on success, `-1` on failure.

3.6.4.3.2.6. `NdbIndexScanOperation::reset_bounds()`

Description. Reset the bounds, and put the operation into the list that will be sent on the next `NdbTransaction::execute()` call.

Signature.

```
int reset_bounds
    (
      bool forceSend = false
    )
```

Parameters. Set `forceSend` to true in order to force the operation to be sent immediately.

Return Value. `0` on success, `-1` on failure.

3.6.4.3.2.7. `NdbIndexScanOperation::end_of_bound()`

Description. This method is used to mark the end of a bound; used when batching index reads (that is, when employing multiple ranges).

Signature.

```
int end_of_bound
    (
      Uint32 range_no
    )
```

Parameters. The number of the range on which the bound occurs.

Return Value. `0` indicates success; `-1` indicates failure.

3.7. The `NdbRecAttr` Class

The section describes the `NdbRecAttr` class and its public methods.

Description. `NdbRecAttr` contains the value of an attribute. An `NdbRecAttr` object is used to store an attribute value after it has been retrieved the `NDB` Cluster using the `NdbOperation::getValue()`. This object is allocated by the `NDB` API. A brief example is shown here:

```
MyRecAttr = MyOperation->getValue("ATTR2", NULL);

if(MyRecAttr == NULL) goto error;

if(MyTransaction->execute(Commit) == -1)
  goto error;

ndbout << MyRecAttr->u_32_value();
```

For more examples, see Section 6.1, “Using Synchronous Transactions”.

> **Note**
>
> An `NdbRecAttr` object is instantiated with its value when `NdbTransaction::execute()` is invoked. Prior to this, the value is undefined. (Use `NdbRecAttr::isNULL()` to check whether the value is defined.) This means that an `NdbRecAttr` object has valid information only between the times that `NdbTransaction::execute()` and `Ndb::closeTransaction()` are called. The value of the null indicator is `-1` until `NdbTransaction::execute()` method is invoked.

Public Methods. `NdbRecAttr` has a number of methods for retrieving values of various simple types directly from an instance of this class. To obtain a reference to the value, use `NdbRecAttr::aRef()`. The following table lists all of the public methods of this class and the purpose or use of each method:

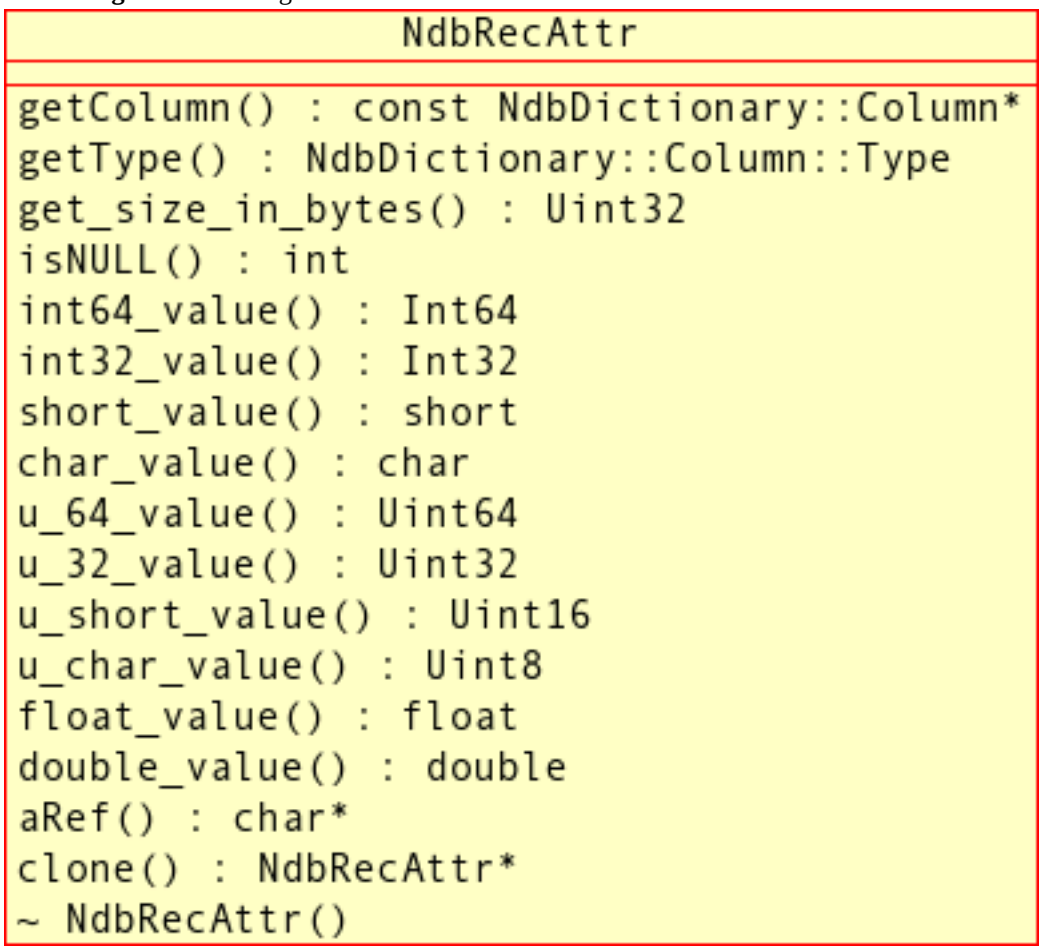
Method	Purpose / Use
`getColumn()`	Gets the column to which the attribute belongs
`getType()`	Gets the attribute's type (`Column::Type`)
`get_size_in_bytes()`	Gets the size of the attribute, in bytes
`isNULL()`	Tests whether the attribute is `NULL`
`int64_value()`	Retrieves the attribute value, as a 64-bit integer
`int32_value()`	Retrieves the attribute value, as a 32-bit integer
`short_value()`	Retrieves the attribute value, as a short integer
`char_value()`	Retrieves the attribute value, as a `char`
`u_64_value()`	Retrieves the attribute value, as an unsigned 64-bit integer
`u_32_value()`	Retrieves the attribute value, as an unsigned 32-bit integer
`u_short_value()`	Retrieves the attribute value, as an unsigned short integer
`u_char_value()`	Retrieves the attribute value, as an unsigned `char`
`float_value()`	Retrieves the attribute value, as a float
`double_value()`	Retrieves the attribute value, as a double
`aRef()`	Gets a pointer to the attribute value

Method	Purpose / Use
clone()	Makes a deep copy of the RecAttr object
~NdbRecAttr()	Destructor method

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.7.1, "NdbRecAttr Class Methods".

Public Types. The NdbRecAttr class defines no public types.

Class Diagram. This diagram shows all the available methods of the NdbRecAttr class:



3.7.1. NdbRecAttr Class Methods

This section lists and describes the public methods of the NdbRecAttr class.

Constructor and Destructor. The NdbRecAttr class has no public constructor; an instance of this object is created using NdbTransaction::execute(). The destructor method, which is public, is discussed in Section 3.7.1.17, "~NdbRecAttr()".

3.7.1.1. NdbRecAttr::getColumn()

Description. This method is used to obtain the column to which the attribute belongs.

Signature.

```
const NdbDictionary::Column* getColumn
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to a `Column` object. See Section 3.4.2, "The `Column` Class".

3.7.1.2. `NdbRecAttr::getType()`

Description. This method is used to obtain the column's datatype.

Signature.

```
NdbDictionary::Column::Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `NdbDictionary::Column::Type` value. See Section 3.4.2.1.3, "`Column::Type`" for more information, including permitted values.

3.7.1.3. `NdbRecAttr::get_size_in_bytes()`

Description. You can use this method to obtain the size of an attribute (element).

Signature.

```
Uint32 get_size_in_bytes
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The attribute size in bytes, as an unsigned 32-bit integer.

3.7.1.4. `NdbRecAttr::isNULL()`

Description. This method checks whether an attribute value is `NULL`.

Signature.

```
int isNULL
    (
      void
    ) const
```

Parameters. *None*.

Return Value. One of the following 3 values:

- `-1`: The attribute value is not defined, either due to an error, or because `NdbTransaction::execute()` has not yet been used.

- `0`: The attribute value is defined, but is not `NULL`.
- `1`: The attribute value is defined and is `NULL`.

3.7.1.5. NdbRecAttr::in64_value()

Description. This method gets the value stored in an `NdbRecAttr` object, and returns it as a 64-bit integer.

Signature.

```
Int64 int64_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 64-bit integer.

3.7.1.6. NdbRecAttr::int32_value()

Description. This method gets the value stored in an `NdbRecAttr` object, and returns it as a 32-bit integer.

Signature.

```
Int32 int32_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 32-bit integer.

3.7.1.7. NdbRecAttr::short_value()

Description. This method gets the value stored in an `NdbRecAttr` object, and returns it as a 16-bit integer (short).

Signature.

```
short short_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 16-bit integer.

3.7.1.8. NdbRecAttr::char_value()

Description. This method gets the value stored in an `NdbRecAttr` object, and returns it as a `char`.

Signature.

```
char char_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `char` value.

3.7.1.9. NdbRecAttr::u_64_value()

Description. This method gets the value stored in an `NdbRecAttr` object, and returns it as an unsigned 64-bit integer.

Signature.

```
Uint64 u_64_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An unsigned 64-bit integer.

3.7.1.10. NdbRecAttr::in64_value()

Description. This method gets the value stored in an `NdbRecAttr` object, and returns it as an unsigned 32-bit integer.

Signature.

```
Uint32 u_32_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An unsigned 32-bit integer.

3.7.1.11. NdbRecAttr::u_short_value()

Description. This method gets the value stored in an `NdbRecAttr` object, and returns it as an unsigned 16-bit (short) integer.

Signature.

```
Uint16 u_short_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An unsigned short (16-bit) integer.

3.7.1.12. NdbRecAttr::u_char_value()

Description. This method gets the value stored in an NdbRecAttr object, and returns it as an unsigned char.

Signature.

```
Uint8 u_char_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An unsigned 8-bit char value.

3.7.1.13. NdbRecAttr::float_value()

Description. This method gets the value stored in an NdbRecAttr object, and returns it as a float.

Signature.

```
float float_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A float.

3.7.1.14. NdbRecAttr::double_value()

Description. This method gets the value stored in an NdbRecAttr object, and returns it as a double.

Signature.

```
double double_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A double.

3.7.1.15. NdbRecAttr::aRef()

Description. This method is used to obtain a reference to an attribute value, as a char pointer. This pointer is aligned appropriately for the datatype. The memory is released by the NDB API when Ndb-Transaction::closeTransaction() is executed on the transaction which read the value.

Signature.

```
char* aRef
    (
      void
    ) const
```

Parameters. A pointer to the attribute value. Because this pointer is constant, this method can be called anytime after NdbOperation::getValue() has been called.

Return Value. *None*.

3.7.1.16. NdbRecAttr::clone()

Description. This method creates a deep copy of an NdbRecAttr object.

> **Note**
>
> The copy created by this method should be deleted by the application when no longer needed.

Signature.

```
NdbRecAttr* clone
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An NdbRecAttr object. This is a complete copy of the original, including all data.

3.7.1.17. ~NdbRecAttr()

Description. The NdbRecAttr class destructor method.

> **Important**
>
> You should delete only copies of NdbRecAttr objects that were created in your application using the clone() method. See Section 3.7.1.16, "NdbRecAttr::clone()".

Signature.

```
~NdbRecAttr
    (
      void
    )
```

Parameters. *None*.

Return Value. *None*.

3.8. The NdbScanFilter Class

This section discusses the NdbScanFliter class and it public members.

Description. NdbScanFilter provides an alternative means of specifying filters for scan operations.

> **Important**
>
> This interface is still under development and is likely to change in future releases.

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
NdbScanFilter()	Constructor method
~NdbScanFilter()	Destructor method

Method	Purpose / Use
begin()	Begins a compound (set of conditions)
end()	Ends a compound
cmp()	Compares a column value with an arbitrary value
eq()	Tests for equality
ne()	Tests for inequality
lt()	Tests for a less-than condition
le()	Tests for a less-than-or-equal condition
gt()	Tests for a greater-than condition
ge()	Tests for a greater-than-or-equal condition
isnull()	Tests whether a column value is NULL
isnotnull()	Tests whether a column value is not NULL

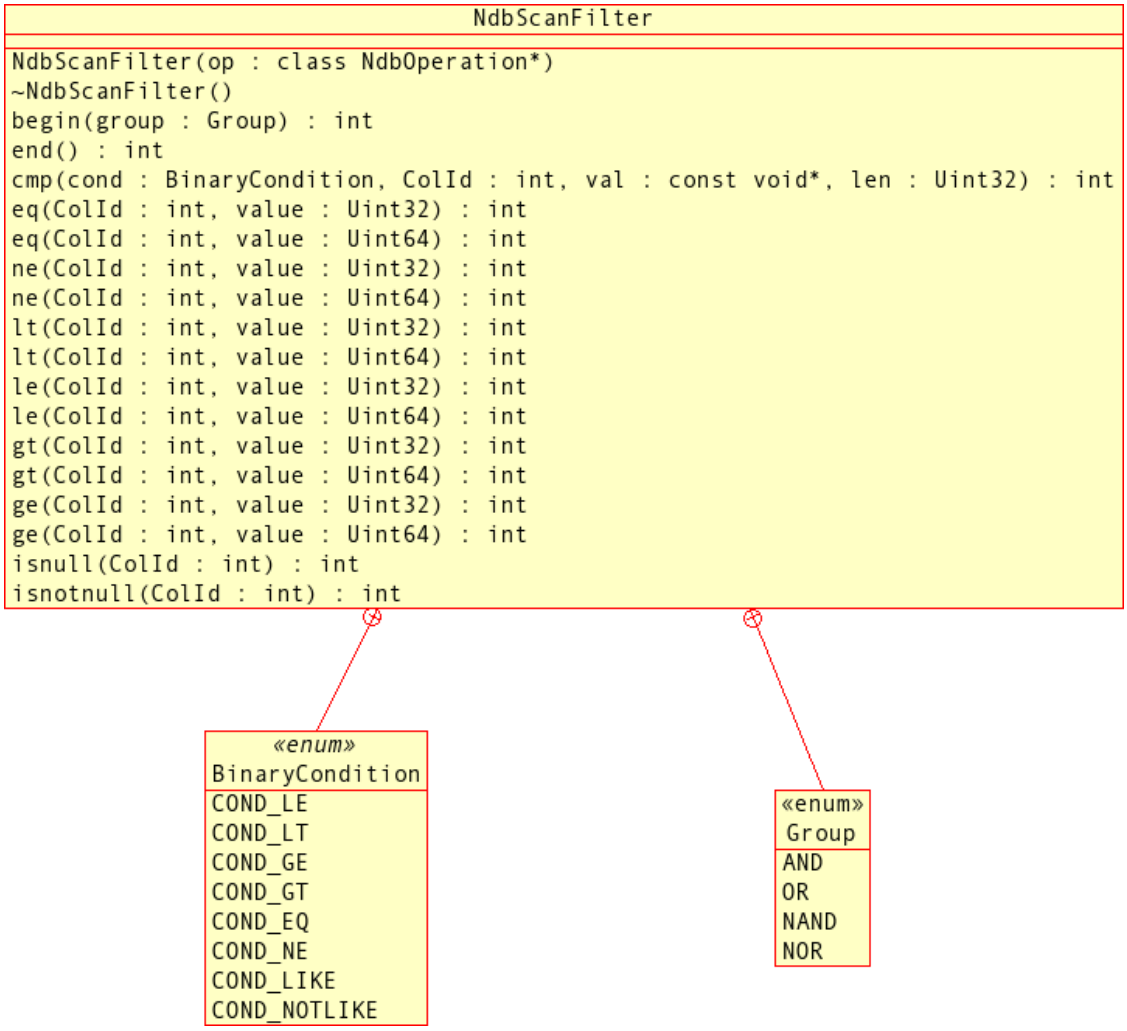
For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.8.2, "NdbScanFilter Class Methods".

Public Types. The NdbScanFilter class defines two public types:

- BinaryCondition: The type of condition, such as lower bound or upper bound.
- Group: A logical grouping operator, such as AND or OR.

For a discussion of each of these types, along with its possible values, see Section 3.8.1, "NdbScanFilter Types".

Class Diagram. This diagram shows all the public members of the NdbScanFilter class:



3.8.1. `NdbScanFilter` Types

This section details the public types belonging to the `NdbScanFilter` class.

3.8.1.1. The `NdbScanFilter::BinaryCondition` Type

Description. This type represents a condition based on the comparison of a column value with some integer value — that is, a bound condition.

Enumeration Values.

Value	Description
`COND_LE`	Lower bound (`<=`)
`COND_LT`	Strict lower bound (`<`)
`COND_GE`	Upper bound (`>=`)
`COND_GT`	Strict upper bound (`>`)
`COND_EQ`	Equality (`=`)
`COND_NE`	Inequality (`<>` or `!=`)
`COND_LIKE`	`LIKE` condition

Value	Description
COND_NOTLIKE	NOT LIKE condition

3.8.1.2. The NdbScanFilter::Group Type

Description. This type is used to describe logical (grouping) operators, and is used with the begin() method. (See Section 3.8.2.2, "NdbScanFilter::begin()".)

Enumeration Values.

Value	Description
AND	Logical AND: *A* AND *B* AND *C*
OR	Logical OR: *A* OR *B* OR *C*
NAND	Logical NOT AND: NOT (*A* AND *B* AND *C*)
NOR	Logical NOT OR: NOT (*A* OR *B* OR *C*)

3.8.2. NdbScanFilter Class Methods

This section lists and describes the public methods of the NdbScanFilter class.

3.8.2.1. NdbScanFilter Class Constructor

Description. This is the constructor method for NdbScanFilter, and creates a new instance of the class.

Signature.

```
NdbScanFilter
    (
      class NdbOperation* op
    )
```

Parameters. This method takes a single parameter: a pointer to the NdbOperation to which the filter applies.

Return Value. A new instance of NdbScanFilter.

Destructor. The destructor takes no arguments and does not return a value. It should be called to remove the NdbScanFilter object when it is no longer needed.

3.8.2.2. NdbScanFilter::begin()

Description. This method is used to start a compound, and specifies the logical operator used to group together the conditions making up the compound. The default is AND.

Signature.

```
int begin
    (
      Group group = AND
    )
```

Parameters. A Group value: one of AND, OR, NAND, or NOR. See Section 3.8.1.2, "The NdbScan-

`Filter::Group` Type”, for additional information.

Return Value. `0` on success, `-1` on failure.

3.8.2.3. `NdbScanFilter::end()`

Description. This method completes a compound, signalling that there are no more conditions to be added to it.

Signature.

```
int end
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

3.8.2.4. `NdbScanFilter::eq()`

Description. This method is used to perform an equality test on a column value and an integer.

Signature.

```
int eq
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int eq
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. This method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

3.8.2.5. `NdbScanFilter::ne()`

Description. This method is used to perform an inequality test on a column value and an integer.

Signature.

```
int ne
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int ne
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like `eq()` and the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

3.8.2.6. `NdbScanFilter::lt()`

Description. This method is used to perform a less-than (strict lower bound) test on a column value and an integer.

Signature.

```
int lt
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int lt
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like `eq()`, `ne()`, and the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

3.8.2.7. `NdbScanFilter::le()`

Description. This method is used to perform a less-than-or-equal test on a column value and an integer.

Signature.

```
int le
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int le
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

3.8.2.8. `NdbScanFilter::gt()`

Description. This method is used to perform a greater-than (strict upper bound) test on a column value and an integer.

Signature.

```
int gt
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int gt
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

3.8.2.9. `NdbScanFilter::ge()`

Description. This method is used to perform a greater-than-or-equal test on a column value and an integer.

Signature.

```
int ge
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int ge
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like `eq()`, `lt()`, `le()`, and the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

3.8.2.10. `NdbScanFilter::isnull()`

Description. This method is used to check whether a column value is `NULL`.

Signature.

```
int isnull
    (
      int ColId
    )
```

Parameters. The ID of the column whose value is to be tested.

Return Value. `0` if the value is `NULL`.

3.8.2.11. `NdbScanFilter::isnotnull()`

Description. This method is used to check whether a column value is not `NULL`.

Signature.

```
int isnotnull
    (
      int ColId
    )
```

Parameters. The ID of the column whose value is to be tested.

Return Value. `0` if the value is not `NULL`.

3.9. The `NdbTransaction` Class

This section describes the `NdbTransaction` class and its public members.

Description. A transaction is represented in the `NDB` API by an `NdbTransaction` object, which belongs to an `Ndb` object and is created using `Ndb::startTransaction()`. A transaction consists of a list of operations represented by the `NdbOperation` class, or by one of its subclasses — `NdbScanOperation`, `NdbIndexOperation`, or `NdbIndexScanOperation` (see Section 3.6, "The `NdbOperation` Class"). Each operation access exactly one table.

Using Transactions. After obtaining an `NdbTransaction` object, it is employed as follows:

- An operation is allocated to the transaction using one of these methods:
  - `getNdbOperation()`
  - `getNdbScanOperation()`
  - `getNdbIndexOperation()`
  - `getNdbIndexScanOperation()`

  Calling one of these methods defines the operation. Several operations can be defined on the same `NdbTransaction` object, in which case they are executed in parallel. When all operations are defined, the `execute()` method sends them to the `NDB` kernel for execution.
- The `execute()` method returns when the `NDB` kernel has completed execution of all operations previously defined.

  Important

  All allocated operations should be properly defined before calling the `execute()` method.
- `execute()` performs its task in one of 3 modes, listed here:
  - `NdbTransaction::NoCommit`: Executes operations without committing them.
  - `NdbTransaction::Commit`: Executes any remaining operation and then commits the complete transaction.
  - `NdbTransaction::Rollback`: Rolls back the entire transaction.

  `execute()` is also equipped with an extra error handling parameter, which provides two alternatives:
  - `NdbTransaction::AbortOnError`: Any error causes the transaction to be aborted. This is the default behaviour.
  - `NdbTransaction::AO_IgnoreError`: The transaction continues to be executed even if one or more of the operations defined for that transaction fails.

Public Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`getNdbOperation()`	Gets an `NdbOperation`
`getNdbScanOperation()`	Gets an `NdbScanOperation`
`getNdbIndexScanOperation()`	Gets an `NdbIndexScanOperation`
`getNdbIndexOperation()`	Gets an `NdbIndexOperation`
`execute`	Executes a transaction
`refresh()`	Keeps a transaction from timing out
`close()`	Closes a transaction
`getGCI()`	Gets a transaction's global checkpoint ID (GCI)
`getTransactionId()`	Gets the transaction ID
`commitStatus()`	Gets the transaction's commit status

Method	Purpose / Use
`getNdbError()`	Gets the most recent error
`getNdbErrorOperation()`	Gets the most recent operation which caused an error
`getNdbErrorLine()`	Gets the line number where the most recent error occurred
`getNextCompletedOperation()`	Gets operations that have been executed; used for finding errors

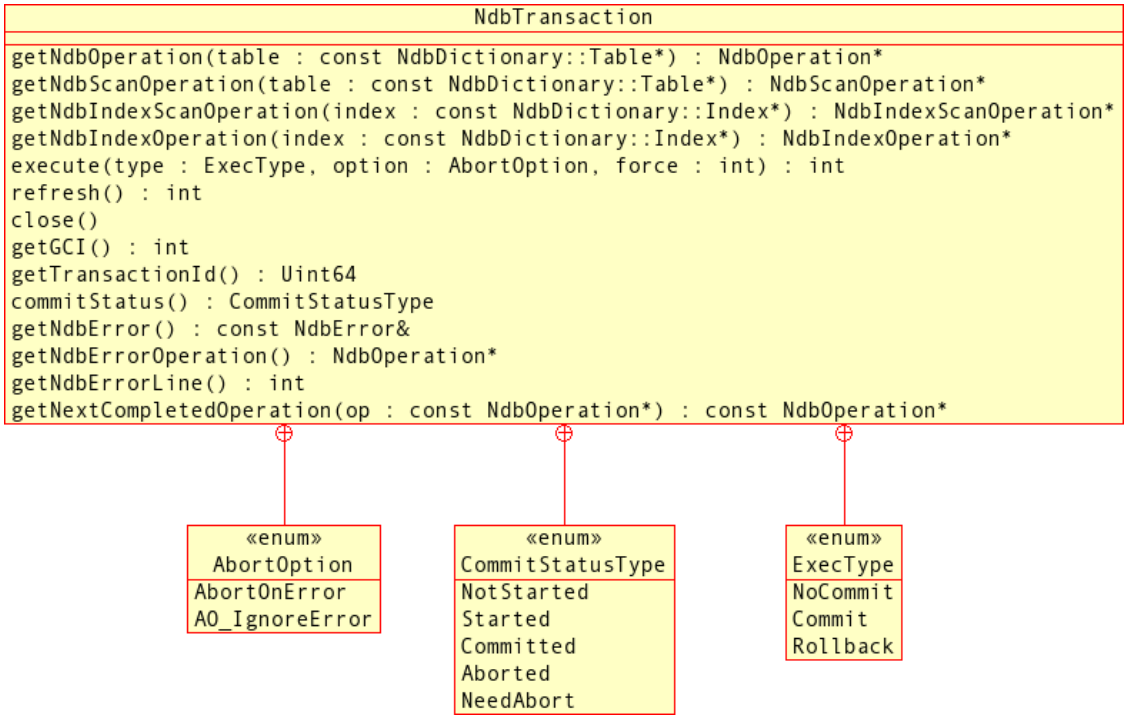
For detailed descriptions, signatures, and examples of use for each of these methods, see Section 3.9.2, "`NdbTransaction` Class Methods".

Public Types. `NdbTransaction` defines 3 public types as shown in the following table:

Type	Purpose / Use
`AbortOption`	Determines whether failed operations cause a transaction to abort
`CommitStatusType`	Describes the transaction's commit status
`ExecType`	Determines whether the transaction should be committed or rolled back

For a discussion of each of these types, along with its possible values, see Section 3.9.1, "`NdbTransaction` Types".

Class Diagram. This diagram shows all the available methods and enumerated types of the `...` class:



3.9.1. `NdbTransaction` Types

This section details the public types belonging to the `NdbTransaction` class.

3.9.1.1. The NdbTransaction::AbortOption Type

Description. This type is used to determine whether failed operations should force a transaction to be aborted. It is used as an argument to the `execute()` method — see Section 3.9.2.5, "`NdbTransaction::execute()`", for more information.

Enumeration Values.

Value	Description
`AbortOnError`	A failed operation causes the transaction to abort.
`AO_IgnoreOnError`	Failed operations are ignored; the transaction continues to execute.

3.9.1.2. The NdbTransaction::CommitStatusType Type

Description. This type is used to describe a transaction's commit status.

Enumeration Values.

Value	Description
`NotStarted`	The transaction has not yet been started.
`Started`	The transaction has started, but is not yet committed.
`Committed`	The transaction has completed, and has been committed.
`Aborted`	The transaction was aborted.
`NeedAbort`	The transaction has encountered an error, but has not yet been aborted.

A transaction's commit status ca be read using the `commitStatus()` method. See Section 3.9.2.10, "`NdbTransaction::commitStatus()`".

3.9.1.3. The NdbTransaction::ExecType Type

Description. This type sets the transaction's execution type — that is, whether it should execute, execute and commit, or abort. It is used as a parameter to the `execute()` method. (See Section 3.9.2.5, "`NdbTransaction::execute()`".)

Enumeration Values.

Value	Description
`NoCommit`	The transaction should execute, but not commit.
`Commit`	The transaction should execute and be committed.
`Rollback`	The transaction should be rolled back.

3.9.2. NdbTransaction Class Methods

This section lists and describes the public methods of the `NdbTransaction` class.

3.9.2.1. NdbTransaction::getNdbOperation()

Description. This method is used to create an `NdbOperation` associated with a given table.

> **Note**
>
> All operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbOperation* getNdbOperation
    (
      const NdbDictionary::Table* table
    )
```

Parameters. The `Table` object on which the operation is to be performed. See Section 3.4.3.7, "The `Table` Class".

Return Value. A pointer to the new `NdbOperation`. See Section 3.6, "The `NdbOperation` Class".

3.9.2.2. `NdbTransaction::getNdbScanOperation()`

Description. This method is used to create an `NdbScanOperation` associated with a given table.

> **Note**
>
> All scan operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbScanOperation* getNdbScanOperation
    (
      const NdbDictionary::Table* table
    )
```

Parameters. The `Table` object on which the operation is to be performed. See Section 3.4.3.7, "The `Table` Class".

Return Value. A pointer to the new `NdbScanOperation`. See Section 3.6.4, "The `NdbScanOperation` Class".

3.9.2.3. `NdbTransaction::getNdbIndexScanOperation()`

Description. This method is used to create an `NdbIndexScanOperation` associated with a given table.

> **Note**
>
> All index scan operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbIndexScanOperation* getNdbIndexScanOperation
    (
      const NdbDictionary::Index* index
    )
```

Parameters. The `Index` object on which the operation is to be performed. See Section 3.4.3.5, "The `Index` Class".

Return Value. A pointer to the new `NdbIndexScanOperation`. See Section 3.6.4.3, "The `Nd-`

bIndexScanOperation Class”.

3.9.2.4. NdbTransaction::getNdbIndexOperation()

Description. This method is used to create an NdbIndexOperation associated with a given table.

> **Note**
>
> All index operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbIndexOperation* getNdbIndexOperation
    (
      const NdbDictionary::Table* table
    )
```

Parameters. The Table object on which the operation is to be performed. See Section 3.4.3.7, “The Table Class”.

Return Value. A pointer to the new NdbIndexOperation. See Section 3.6.3, “The NdbIndexOperation Class”.

3.9.2.5. NdbTransaction::execute()

Description. This method is used to execute a transaction.

Signature.

```
int execute
    (
      ExecType execType,
      AbortOption abortOption = AbortOnError,
      int force = 0
    )
```

Parameters. The execute method takes 3 parameters, as described here:

- The execution type (ExecType value); see Section 3.9.1.3, “The NdbTransaction::ExecType Type”, for more information and possible values.
- An abort option (AbortOption value); see Section 3.9.1.3, “The NdbTransaction::ExecType Type”, for more information and possible values.
- A *force* parameter, which determines when operations should be sent to the NDB Kernel:
  - 0: Non-forced; detected by the adaptive send algorithm.
  - 1: Forced; detected by the adaptive send algorithm.
  - 2: Non-forced; not detected by the adaptive send algorithm.

  See Section 1.3.4, “The Adaptive Send Algorithm”.

Return Value. 0 on success, -1 on failure.

3.9.2.6. NdbTransaction::refresh()

Description. This method updates the transaction's timeout counter, and thus avoids aborting due to

transaction timeout.

> **Note**
>
> It is not advisable to take a lock on a record and maintain it for a extended time since this can impact other transactions.

Signature.

```
int refresh
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

3.9.2.7. `NdbTransaction::close()`

Description. This method closes a transaction. It is equivalent to calling `Ndb::closeTransaction()` (see Section 3.1.1.9, "`Ndb::closeTransaction()`").

Signature.

```
void close
    (
      void
    )
```

Parameters. *None*.

Return Value. *None*.

3.9.2.8. `NdbTransaction::getGCI()`

Description. This method retrieves the transaction's global checkpoint ID (GCI).

Each committed transaction belongs to a GCI. The log for the committed transaction is saved on disk when a global checkpoint occurs.

By comparing the GCI of a transaction with the value of the latest GCI restored in a restarted NDB Cluster, you can determine whether or not the transaction was restored.

> **Note**
>
> Whether or not the global checkpoint with this GCI has been saved on disk cannot be determined by this method.

> **Important**
>
> The GCI for a scan transaction is undefined, since no updates are performed in scan transactions.

Signature.

```
int getGCI
    (
      void
    )
```

Parameters. *None*.

Return Value. The transaction's GCI, or `-1` if none is available.

> **Note**
>
> No GCI is available until `execute()` has been called with `ExecType::Commit`.

3.9.2.9. `NdbTransaction::getTransactionId()`

Description. This method is used to obtain the transaction ID.

Signature.

```
Uint64 getTransactionId
    (
      void
    )
```

Parameters. *None*.

Return Value. The transaction ID, as an unsigned 64-bit integer.

3.9.2.10. `NdbTransaction::commitStatus()`

Description. This method gets the transaction's commit status.

Signature.

```
CommitStatusType commitStatus
    (
      void
    )
```

Parameters. *None*.

Return Value. The commit status of the transaction, a `CommitStatusType` value. See Section 3.9.1.2, "The `NdbTransaction::CommitStatusType` Type".

3.9.2.11. `NdbTransaction::getNdbError()`

Description. This method is used to obtain the most recent error (`NdbError`).

Signature.

```
const NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A reference to an `NdbError` object. See Section 4.1, "The `NdbError` Structure".

> **Note**
>
> For additional information about handling errors in transactions, see Section 1.3.2.3.5, "Error Handling".

3.9.2.12. NdbTransaction::getNdbErrorOperation()

Description. This method retrieves the operation that caused an error.

> **Tip**
>
> To obtain more information about the actual error, use the `NdbOperation::getNdbError()` method of the `NdbOperation` object returned by `getNdbErrorOperation()`. (See Section 3.6.2.5, "`NdbOperation::getNdbError()`".)

Signature.

```
NdbOperation* getNdbErrorOperation
    (
      void
    )
```

Parameters. *None*.

Return Value. A pointer to an `NdbOperation`.

> **Note**
>
> For additional information about handling errors in transactions, see Section 1.3.2.3.5, "Error Handling".

3.9.2.13. NdbTransaction::getNdbErrorLine()

Description. This method return the line number where the most recent error occurred.

Signature.

```
int getNdbErrorLine
    (
      void
    )
```

Parameters. *None*.

Return Value. The line number of the most recent error.

> **Note**
>
> For additional information about handling errors in transactions, see Section 1.3.2.3.5, "Error Handling".

3.9.2.14. NdbTransaction::getNextCompletedOperation()

Description. This method is used to retrieve a transaction's completed operations. It is typically used to fetch all operations belonging to a given transaction to check for errors.

`NdbTransaction::getNextCompletedOperation(NULL)` returns the transaction's first `NdbOperation` object; `NdbTransaction::getNextCompletedOperation(myOp)` returns the `NdbOperation` object defined after `NdbOperation` *`myOp`*.

> **Important**
>
> This method should only be used after the transaction has been executed, but before the transaction has been closed.

Signature.

```
const NdbOperation* getNextCompletedOperation
    (
      const NdbOperation* op
    ) const
```

Parameters. This method requires a single parameter *op*, which is an operation (`NdbOperation` object), or `NULL`.

Return Value. The operation following *op*, or the first operation defined for the transaction if `getNextCompletedOperation()` was called using `NULL`.

Chapter 4. NDB API ERRORS

This chapter discusses reporting and handling of errors potentially generated in NDB API applications. It includes information about the `NdbError` data structure, which is used to model errors, and about NDB API error codes, classifications, and messages.

4.1. The `NdbError` Structure

This section discusses the `NdbError` data structure, which contains status and other information about errors, including error codes, classifications, and messages.

Description. An `NdbError` consists of six parts:

1. *Error status*: This describes the impact of an error on the application, and reflects what the application should do when the error is encountered.

   The error status is described by a value of the `Status` type. See Section 4.1.1.1, "The `NdbError::Status` Type", for possible `Status` values and how they should be interpreted.

2. *Error classification*: This represents a logical error type or grouping.

   The error classification is described by a value of the `Classification` type. See Section 4.1.1.2, "The `NdbError::Classification` Type", for possible classifications and their interpretation. Additional information is provided in Section 4.2.2, "NDB Error Classifications".

3. *Error code*: This is an NDB API internal error code which uniquely identifies the error.

   > **Important**
   >
   > It is *not* recommended to write application programs which are dependent on specific error codes. Instead, applications should check error status and classification. More information about errors can also be obtained by checking error messages and (when available) error detail messages. However — like error codes — these error messages and error detail messages are subject to change.

   A listing of current error codes, broken down by classification, is provided in Section 4.2.1, "`NDB` Error Codes and Messages". This listing will be updated periodically, or you can check the file `storage/ndb/src/ndbapi/ndberror.c` in the MySQL 5.1 sources.

4. *MySQL Error code*: This is the corresponding MySQL Server error code. MySQL error codes are not discussed in this document; please see Server Error Codes and Messages [http://dev.mysql.com/doc/refman/5.1/en/error-messages-server.html], in the MySQL Manual, for information about these.

5. *Error message*: This is a generic, context-independent description of the error.

6. *Error details*: This can often provide additional information (not found in the error message) about an error, specific to the circumstances under which the error is encountered. However, it is not available in all cases.

   Where not specified, the error detail message is `NULL`.

> **Important**
>
> Specific NDB API error codes, messages, and detail messages are subject to change without notice.

Definition. The NdbError structure contains the following members, whose types are as shown:

- `Status status`

  The error status.

- `Classification classification`

  The error type (classification).

- `int code`

  The NDB API error code.

- `int mysql_code`

  The MySQL error code.

- `const char* message`

  The error message.

- `char* details`

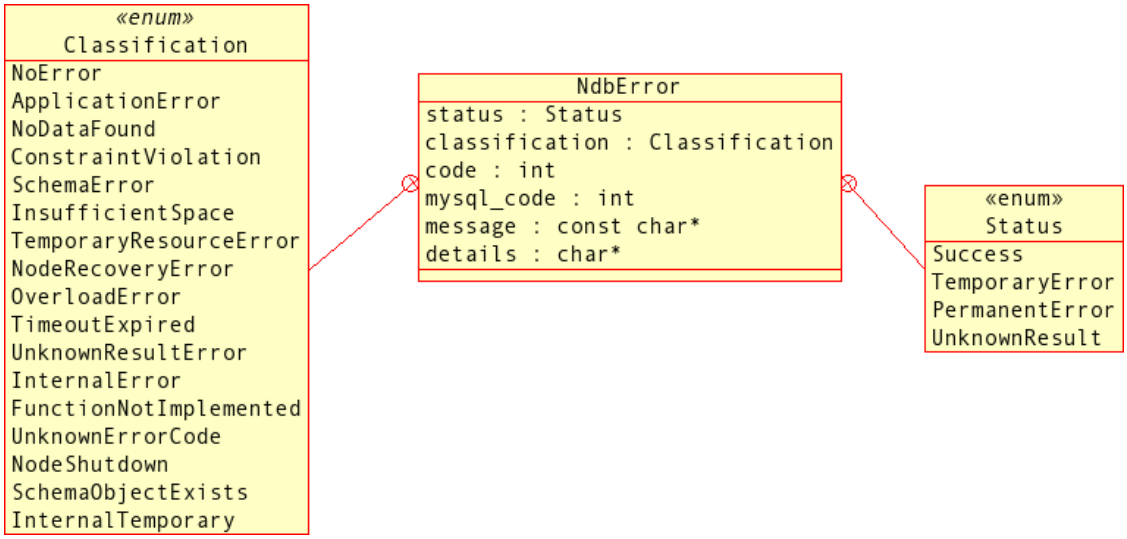
  The error detail message.

See the Description for more information about these members and their types.

Public Types. `NdbError` defines two datatypes:

- `Status`: The error status. See Section 4.1.1.1, “The `NdbError::Status` Type”.
- `Classification`: The type of error or the logical grouping to which it belongs. See Section 4.1.1.2, “The `NdbError::Classification` Type”.

Structure Diagram. This diagram shows all the available members and types of the `NdbError` structure:



4.1.1. `NdbError` Types

This section discusses the `Status` and `Classification` datatypes defined by `NdbError`.

4.1.1.1. The `NdbError::Status` Type

Description. This type is used to describe an error's status.

Enumeration Values.

Value	Description
`Success`	No error has occurred
`TemporaryError`	A temporary and usually recoverable error; the application should retry the operation giving rise to the error
`PermanentError`	Permanent error; not recoverable
`UnknownResult`	The operation's result or status is unknown

> **Note**
>
> Related information specific to certain error conditions may be found in Section 4.2.2, "NDB Error Classifications".

4.1.1.2. The `NdbError::Classification` Type

Description. This type describes the type of error, or the logical group to which it belongs.

Enumeration Values.

Value	Description
`NoError`	Indicates success (no error occurred)
`ApplicationError`	An error occurred in an application program
`NoDataFound`	A read operation failed due to one or more missing records.
`ConstraintViolation`	A constraint violation occurred, such as attempting to insert a tuple having a primary key value already in use in the target table.

Value	Description
SchemaError	An error took place when trying to create or use a table.
InsufficientSpace	There was insufficient memory for data or indexes.
TemporaryResourceError	This type of error is typically encountered when there are too many active transactions.
NodeRecoveryError	This is a temporary failure which was likely caused by a node recovery in progress, some examples being when information sent between an application and NDB is lost, or when there is a distribution change.
OverloadError	This type of error is often caused when there is insufficient logfile space.
TimeoutExpired	A timeout, often caused by a deadlock.
UnknownResultError	It is not known whether a transaction was committed.
InternalError	A serious error has occurred in NDB itself.
FunctionNotImplemented	The application attempted to use a function which is not yet implemented.
UnknownErrorCode	This is seen where the NDB error handler cannot determine the correct error code to report.
NodeShutdown	This is caused by a node shutdown.
SchemaObjectExists	The application attempted to create a schema object that already exists.
InternalTemporary	A request was sent to a non-master node.

> **Note**
>
> Related information specific to certain error conditions may be found in Section 4.2.1, "NDB Error Codes and Messages", and in Section 4.2.2, "NDB Error Classifications".

4.2. NDB Error Codes, Classifications, and Messages

This section contains listings of common NDB error codes and messages, along with their classifications and likely causes for which they might be raised.

> **Important**
>
> It is strongly recommended that you *not* depend on specific error codes in your NDB API applications, as they are subject to change over time. Instead, you should use the `NdbError::Status` and error classification in your source code, or consult the output of `perror --ndb error_code` to obtain information about a specific error code.
>
> If you find a situation in which you need to use a specific error code in your application, please file a bug report at http://bugs.mysql.com/ so that we can update the corresponding status and classification.

4.2.1. NDB Error Codes and Messages

This section contains a number of tables, one for each type of NDB API error. The error types include the following:

- No error

- Application error
- Scan application error
- Configuration or application error
- No data found
- Constraint violation
- Schema error
- User defined error
- Insufficient space
- Temporary Resource error
- Node Recovery error
- Overload error
- Timeout expired
- Node shutdown
- Internal temporary
- Unknown result error
- Unknown error code
- Internal error
- Function not implemented

The information in each table includes, for each error:

- The numeric `NDB` error code
- The corresponding MySQL error code
- The `NDB` classification code

  See Section 4.2.2, “NDB Error Classifications”, for the meanings of these classification codes.
- The text of the error message

Similar errors have been grouped together in each table. The last table — see Section 4.2.1.22, “Other Errors (Not Yet Categorised)” — contains errors that have not yet been assigned to a specific category.

> **Tip**
>
> You can always obtain the latest error codes and information from the file `storage/ndb/src/ndbapi/ndberror.c`.

4.2.1.1. No Error

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
0	0	*NE*	No error

4.2.1.2. NoDataFound Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
626	HA_ERR_KEY_NOT_FOUND	*ND*	Tuple did not exist

4.2.1.3. ConstraintViolation Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
630	HA_ERR_FOUND_DUPP_KEY	*CV*	Tuple already existed when attempting to insert
839	DMEC	*CV*	Illegal null attribute
840	DMEC	*CV*	Trying to set a NOT NULL attribute to NULL
893	HA_ERR_FOUND_DUPP_KEY	*CV*	Constraint violation e.g. duplicate value in unique index

4.2.1.4. Node recovery errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
286	DMEC	*NR*	Node failure caused abort of transaction
250	DMEC	*NR*	Node where lock was held crashed, restart scan transaction
499	DMEC	*NR*	Scan take over error, restart scan transaction
1204	DMEC	*NR*	Temporary failure, distribution changed
4002	DMEC	*NR*	Send to NDB failed
4010	DMEC	*NR*	Node failure caused abort of transaction
4025	DMEC	*NR*	Node failure caused abort of transaction
4027	DMEC	*NR*	Node failure caused abort of transaction
4028	DMEC	*NR*	Node failure caused abort of transaction
4029	DMEC	*NR*	Node failure caused abort of transaction
4031	DMEC	*NR*	Node failure caused abort of transaction
4033	DMEC	*NR*	Send to NDB failed
4115	DMEC	*NR*	Transaction was committed but all read information was not received due to node crash

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4119	DMEC	*NR*	Simple/dirty read failed due to node failure

4.2.1.5. Node Shutdown Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
280	DMEC	*NS*	Transaction aborted due to node shutdown
270	DMEC	*NS*	Transaction aborted due to node shutdown
1223	DMEC	*NS*	Read operation aborted due to node shutdown
4023	DMEC	*NS*	Transaction aborted due to node shutdown
4030	DMEC	*NS*	Transaction aborted due to node shutdown
4034	DMEC	*NS*	Transaction aborted due to node shutdown

4.2.1.6. Unknown Result Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4008	DMEC	*UR*	Receive from NDB failed
4009	DMEC	*UR*	Cluster Failure
4012	DMEC	*UR*	Request ndbd time-out, maybe due to high load or communication problems
4024	DMEC	*UR*	Time-out, most likely caused by simple read or cluster failure

4.2.1.7. Temporary Resource Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
217	DMEC	*TR*	217
218	DMEC	*TR*	218
219	DMEC	*TR*	219
233	DMEC	*TR*	Out of operation records in transaction coordinator (increase MaxNoOfConcurrentOperations)
275	DMEC	*TR*	275
279	DMEC	*TR*	Out of transaction markers in transaction coordinator
414	DMEC	*TR*	414
418	DMEC	*TR*	Out of transaction buffers in LQH
419	DMEC	*TR*	419
245	DMEC	*TR*	Too many active scans
488	DMEC	*TR*	Too many active scans
490	DMEC	*TR*	Too many active scans

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
805	DMEC	*TR*	Out of attrinfo records in tuple manager
830	DMEC	*TR*	Out of add fragment operation records
873	DMEC	*TR*	Out of attrinfo records for scan in tuple manager
899	DMEC	*TR*	Rowid already allocated
1217	DMEC	*TR*	Out of operation records in local data manager (increase MaxNoOfLocalOperations)
1220	DMEC	*TR*	REDO log files overloaded, consult online manual (decrease TimeBetweenLocalCheckpoints, and|or increase NoOfFragmentLogFiles)
1222	DMEC	*TR*	Out of transaction markers in LQH
4021	DMEC	*TR*	Out of Send Buffer space in NDB API
4022	DMEC	*TR*	Out of Send Buffer space in NDB API
4032	DMEC	*TR*	Out of Send Buffer space in NDB API
1501	DMEC	*TR*	Out of undo space
288	DMEC	*TR*	Out of index operations in transaction coordinator (increase MaxNoOfConcurrentIndexOperations)

4.2.1.8. Insufficient Space Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
623	HA_ERR_RECORD_FILE_FULL	*IS*	623
624	HA_ERR_RECORD_FILE_FULL	*IS*	624
625	HA_ERR_INDEX_FILE_FULL	*IS*	Out of memory in Ndb Kernel, hash index part (increase IndexMemory)
640	DMEC	*IS*	Too many hash indexes (should not happen)
826	HA_ERR_RECORD_FILE_FULL	*IS*	Too many tables and attributes (increase MaxNoOfAttributes or MaxNoOfTables)
827	HA_ERR_RECORD_FILE_FULL	*IS*	Out of memory in Ndb Kernel, table data (increase DataMemory)
902	HA_ERR_RECORD_FILE_FULL	*IS*	Out of memory in Ndb Kernel, ordered index data (increase DataMemory)
903	HA_ERR_INDEX_FILE_FULL	*IS*	Too many ordered indexes (increase MaxNoOfOrderedIndexes)
904	HA_ERR_INDEX_FILE_FULL	*IS*	Out of fragment records (increase MaxNoOfOrderedIndexes)
905	DMEC	*IS*	Out of attribute records (increase MaxNoOfAttributes)
1601	HA_ERR_RECORD_FILE_FULL	*IS*	Out extents, tablespace full

4.2.1.9. `TimeoutExpired` errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
266	HA_ERR_LOCK_WAIT_TIMEOUT	*TO*	Time-out in NDB, probably caused by deadlock
274	HA_ERR_LOCK_WAIT_TIMEOUT	*TO*	Time-out in NDB, probably caused by deadlock
296	HA_ERR_LOCK_WAIT_TIMEOUT	*TO*	Time-out in NDB, probably caused by deadlock
297	HA_ERR_LOCK_WAIT_TIMEOUT	*TO*	Time-out in NDB, probably caused by deadlock
237	HA_ERR_LOCK_WAIT_TIMEOUT	*TO*	Transaction had timed out when trying to commit it

4.2.1.10. Overload Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
701	DMEC	*OL*	System busy with other schema operation
711	DMEC	*OL*	System busy with node restart, schema operations not allowed
410	DMEC	*OL*	REDO log files overloaded, consult online manual (decrease TimeBetweenLocalCheckpoints, and|or increase NoOfFragmentLogFiles)
677	DMEC	*OL*	Index UNDO buffers overloaded (increase UndoIndexBuffer)
891	DMEC	*OL*	Data UNDO buffers overloaded (increase UndoDataBuffer)
1221	DMEC	*OL*	REDO buffers overloaded, consult online manual (increase RedoBuffer)
4006	DMEC	*OL*	Connect failure - out of connection objects (increase MaxNoOfConcurrentTransactions)

4.2.1.11. Internal Temporary Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
702	DMEC	*IT*	Request to non-master

4.2.1.12. Internal errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
892	DMEC	*IE*	Inconsistent hash index. The index needs to be dropped and recreated

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
896	DMEC	*IE*	Tuple corrupted - wrong checksum or column data in invalid format
901	DMEC	*IE*	Inconsistent ordered index. The index needs to be dropped and recreated
202	DMEC	*IE*	202
203	DMEC	*IE*	203
207	DMEC	*IE*	207
208	DMEC	*IE*	208
209	DMEC	*IE*	Communication problem, signal error
220	DMEC	*IE*	220
230	DMEC	*IE*	230
232	DMEC	*IE*	232
238	DMEC	*IE*	238
271	DMEC	*IE*	Simple Read transaction without any attributes to read
272	DMEC	*IE*	Update operation without any attributes to update
276	DMEC	*IE*	276
277	DMEC	*IE*	277
278	DMEC	*IE*	278
287	DMEC	*IE*	Index corrupted
290	DMEC	*IE*	Corrupt key in TC, unable to xfrm
631	DMEC	*IE*	631
632	DMEC	*IE*	632
706	DMEC	*IE*	Inconsistency during table creation
809	DMEC	*IE*	809
812	DMEC	*IE*	812
829	DMEC	*IE*	829
833	DMEC	*IE*	833
871	DMEC	*IE*	871
882	DMEC	*IE*	882
883	DMEC	*IE*	883
887	DMEC	*IE*	887
888	DMEC	*IE*	888
890	DMEC	*IE*	890
4000	DMEC	*IE*	MEMORY ALLOCATION ERROR
4001	DMEC	*IE*	Signal Definition Error
4005	DMEC	*IE*	Internal Error in NdbApi
4011	DMEC	*IE*	Internal Error in NdbApi
4107	DMEC	*IE*	Simple Transaction and Not Start
4108	DMEC	*IE*	Faulty operation type
4109	DMEC	*IE*	Faulty primary key attribute length

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4110	DMEC	*IE*	Faulty length in ATTRINFO signal
4111	DMEC	*IE*	Status Error in NdbConnection
4113	DMEC	*IE*	Too many operations received
4320	DMEC	*IE*	Cannot use the same object twice to create table
4321	DMEC	*IE*	Trying to start two schema transactions
4344	DMEC	*IE*	Only DBDICT and TRIX can send requests to TRIX
4345	DMEC	*IE*	TRIX block is not available yet, probably due to node failure
4346	DMEC	*IE*	Internal error at index create/build
4347	DMEC	*IE*	Bad state at alter index
4348	DMEC	*IE*	Inconsistency detected at alter index
4349	DMEC	*IE*	Inconsistency detected at index usage
4350	DMEC	*IE*	Transaction already aborted

4.2.1.13. Application Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
763	DMEC	*AE*	Alter table requires cluster nodes to have exact same version
823	DMEC	*AE*	Too much attrinfo from application in tuple manager
831	DMEC	*AE*	Too many nullable/bitfields in table definition
876	DMEC	*AE*	876
877	DMEC	*AE*	877
878	DMEC	*AE*	878
879	DMEC	*AE*	879
880	DMEC	*AE*	Tried to read too much - too many getValue calls
884	DMEC	*AE*	Stack overflow in interpreter
885	DMEC	*AE*	Stack underflow in interpreter
886	DMEC	*AE*	More than 65535 instructions executed in interpreter
897	DMEC	*AE*	Update attempt of primary key via ndbcluster internal api (if this occurs via the MySQL server it is a bug, please report)
4256	DMEC	*AE*	Must call Ndb::init() before this function
4257	DMEC	*AE*	Tried to read too much - too many getValue calls

4.2.1.14. Scan Application Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
242	DMEC	*AE*	Zero concurrency in scan

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
244	DMEC	*AE*	Too high concurrency in scan
269	DMEC	*AE*	No condition and attributes to read in scan
4600	DMEC	*AE*	Transaction is already started
4601	DMEC	*AE*	Transaction is not started
4602	DMEC	*AE*	You must call getNdbOperation before executeScan
4603	DMEC	*AE*	There can only be ONE operation in a scan transaction
4604	DMEC	*AE*	takeOverScanOp, to take over a scanned row one must explicitly request keyinfo on readTuples call
4605	DMEC	*AE*	You may only call openScanRead or openScanExclusive once for each operation
4607	DMEC	*AE*	There may only be one operation in a scan transaction
4608	DMEC	*AE*	You can not takeOverScan unless you have used openScanExclusive
4609	DMEC	*AE*	You must call nextScanResult before trying to takeOverScan
4232	DMEC	*AE*	Parallelism can only be between 1 and 240

4.2.1.15. Event Schema Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4713	DMEC	*SE*	Column defined in event does not exist in table

4.2.1.16. Event Application Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4707	DMEC	*AE*	Too many event have been defined
4708	DMEC	*AE*	Event name is too long
4709	DMEC	*AE*	Can't accept more subscribers
746	DMEC	*OE*	Event name already exists
747	DMEC	*IS*	Out of event records
748	DMEC	*TR*	Busy during read of event table
4710	DMEC	*AE*	Event not found
4711	DMEC	*AE*	Creation of event failed
4712	DMEC	*AE*	Stopped event operation does not exist. Already stopped?

4.2.1.17. Event Internal Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4731	DMEC	*IE*	Event not found

4.2.1.18. Schema Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
311	DMEC	*AE*	Undefined partition used in setPartitionId
703	DMEC	*SE*	Invalid table format
704	DMEC	*SE*	Attribute name too long
705	DMEC	*SE*	Table name too long
707	DMEC	*SE*	No more table metadata records (increase MaxNoOfTables)
708	DMEC	*SE*	No more attribute metadata records (increase MaxNoOfAttributes)
709	HA_ERR_NO_SUCH_TABLE	*SE*	No such table existed
710	DMEC	*SE*	Internal: Get by table name not supported, use table id.
721	HA_ERR_TABLE_EXIST	*OE*	Table or index with given name already exists
723	HA_ERR_NO_SUCH_TABLE	*SE*	No such table existed
736	DMEC	*SE*	Unsupported array size
737	HA_WRONG_CREATE_OPTION	*SE*	Attribute array size too big
738	HA_WRONG_CREATE_OPTION	*SE*	Record too big
739	HA_WRONG_CREATE_OPTION	*SE*	Unsupported primary key length
740	HA_WRONG_CREATE_OPTION	*SE*	Nullable primary key not supported
741	DMEC	*SE*	Unsupported alter table
743	HA_WRONG_CREATE_OPTION	*SE*	Unsupported character set in table or index
744	DMEC	*SE*	Character string is invalid for given character set
745	HA_WRONG_CREATE_OPTION	*SE*	Distribution key not supported for char attribute (use binary attribute)
771	HA_WRONG_CREATE_OPTION	*AE*	Given NODEGROUP doesn't exist in this cluster
772	HA_WRONG_CREATE_OPTION	*IE*	Given fragmentType doesn't exist
749	HA_WRONG_CREATE_OPTION	*IE*	Primary Table in wrong state
763	HA_WRONG_CREATE_	*SE*	Invalid undo buffer size

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
	OPTION		
764	HA_WRONG_CREATE_OPTION	*SE*	Invalid extent size
765	DMEC	*SE*	Out of filegroup records
750	IE	*SE*	Invalid file type
751	DMEC	*SE*	Out of file records
752	DMEC	*SE*	Invalid file format
753	IE	*SE*	Invalid filegroup for file
754	IE	*SE*	Invalid filegroup version when creating file
755	HA_WRONG_CREATE_OPTION	*SE*	Invalid tablespace
756	DMEC	*SE*	Index on disk column is not supported
757	DMEC	*SE*	Varsize bitfield not supported
758	DMEC	*SE*	Tablespace has changed
759	DMEC	*SE*	Invalid tablespace version
760	DMEC	*SE*	File already exists
761	DMEC	*SE*	Unable to drop table as backup is in progress
762	DMEC	*SE*	Unable to alter table as backup is in progress
766	DMEC	*SE*	Can't drop file, no such file
767	DMEC	*SE*	Can't drop filegroup, no such filegroup
768	DMEC	*SE*	Can't drop filegroup, filegroup is used
769	DMEC	*SE*	Drop undofile not supported, drop logfile group instead
770	DMEC	*SE*	Can't drop file, file is used
774	DMEC	*SE*	Invalid schema object for drop
241	HA_ERR_TABLE_DEF_CHANGED	*SE*	Invalid schema object version
283	HA_ERR_NO_SUCH_TABLE	*SE*	Table is being dropped
284	HA_ERR_TABLE_DEF_CHANGED	*SE*	Table not defined in transaction coordinator
285	DMEC	*SE*	Unknown table error in transaction coordinator
881	DMEC	*SE*	Unable to create table, out of data pages (increase DataMemory)
906	DMEC	*SE*	Unsupported attribute type in index
907	DMEC	*SE*	Unsupported character set in table or index
908	DMEC	*IS*	Invalid ordered index tree node size
1225	DMEC	*SE*	Table not defined in local query handler
1226	DMEC	*SE*	Table is being dropped
1228	DMEC	*SE*	Cannot use drop table for drop index
1229	DMEC	*SE*	Too long frm data supplied

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
1231	DMEC	*SE*	Invalid table or index to scan
1232	DMEC	*SE*	Invalid table or index to scan
1502	DMEC	*IE*	Filegroup already exists
1503	DMEC	*SE*	Out of filegroup records
1504	DMEC	*SE*	Out of logbuffer memory
1505	DMEC	*IE*	Invalid filegroup
1506	DMEC	*IE*	Invalid filegroup version
1507	DMEC	*IE*	File no already inuse
1508	DMEC	*SE*	Out of file records
1509	DMEC	*SE*	File system error, check if path,permissions etc
1510	DMEC	*IE*	File meta data error
1511	DMEC	*IE*	Out of memory
1512	DMEC	*SE*	File read error
1513	DMEC	*IE*	Filegroup not online
1514	DMEC	*SE*	Currently there is a limit of one logfile group
773	DMEC	*SE*	Out of string memory, please modify String-Memory config parameter
775	DMEC	*SE*	Create file is not supported when Diskless=1
776	DMEC	*AE*	Index created on temporary table must itself be temporary
777	DMEC	*AE*	Cannot create a temporary index on a non-temporary table
778	DMEC	*AE*	A temporary table or index must be specified as not logging

4.2.1.19. `FunctionNotImplemented` Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4003	DMEC	*NI*	Function not implemented yet

4.2.1.20. Backup Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
1300	DMEC	*IE*	Undefined error
1301	DMEC	*IE*	Backup issued to not master (reissue command to master)
1302	DMEC	*IE*	Out of backup record
1303	DMEC	*IS*	Out of resources
1304	DMEC	*IE*	Sequence failure

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
1305	DMEC	*IE*	Backup definition not implemented
1306	DMEC	*AE*	Backup not supported in diskless mode (change Diskless)
1321	DMEC	*UD*	Backup aborted by user request
1322	DMEC	*IE*	Backup already completed
1323	DMEC	*IE*	1323
1324	DMEC	*IE*	Backup log buffer full
1325	DMEC	*IE*	File or scan error
1326	DMEC	*IE*	Backup abortet due to node failure
1327	DMEC	*IE*	1327
1340	DMEC	*IE*	Backup undefined error
1342	DMEC	*AE*	Backup failed to allocate buffers (check configuration)
1343	DMEC	*AE*	Backup failed to setup fs buffers (check configuration)
1344	DMEC	*AE*	Backup failed to allocate tables (check configuration)
1345	DMEC	*AE*	Backup failed to insert file header (check configuration)
1346	DMEC	*AE*	Backup failed to insert table list (check configuration)
1347	DMEC	*AE*	Backup failed to allocate table memory (check configuration)
1348	DMEC	*AE*	Backup failed to allocate file record (check configuration)
1349	DMEC	*AE*	Backup failed to allocate attribute record (check configuration)
1329	DMEC	*AE*	Backup during software upgrade not supported

4.2.1.21. Node ID Allocation Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
1700	DMEC	*IE*	Undefined error
1701	DMEC	*AE*	Node already reserved
1702	DMEC	*AE*	Node already connected
1703	DMEC	*AE*	Node failure handling not completed
1704	DMEC	*AE*	Node type mismatch

4.2.1.22. Other Errors (Not Yet Categorised)

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
720	DMEC	*AE*	Attribute name reused in table definition
1405	DMEC	*NR*	Subscriber manager busy with node recovery

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
1407	DMEC	*SE*	Subscription not found in subscriber manager
1411	DMEC	*TR*	Subscriber manager busy with adding/removing a subscriber
1412	DMEC	*IS*	Can't accept more subscribers, out of space in pool
1413	DMEC	*TR*	Subscriber manager busy with adding the subscription
1414	DMEC	*TR*	Subscriber manager has subscribers on this subscription
1415	DMEC	*SE*	Subscription not unique in subscriber manager
1416	DMEC	*IS*	Can't accept more subscriptions, out of space in pool
1417	DMEC	*SE*	Table in suscription not defined, probably dropped
1418	DMEC	*SE*	Subscription dropped, no new subscribers allowed
1419	DMEC	*SE*	Subscription already dropped
1420	DMEC	*TR*	Subscriber manager busy with adding/removing a table
4004	DMEC	*AE*	Attribute name or id not found in the table
4100	DMEC	*AE*	Status Error in NDB
4101	DMEC	*AE*	No connections to NDB available and connect failed
4102	DMEC	*AE*	Type in NdbTamper not correct
4103	DMEC	*AE*	No schema connections to NDB available and connect failed
4104	DMEC	*AE*	Ndb Init in wrong state, destroy Ndb object and create a new
4105	DMEC	*AE*	Too many Ndb objects
4106	DMEC	*AE*	All Not NULL attribute have not been defined
4114	DMEC	*AE*	Transaction is already completed
4116	DMEC	*AE*	Operation was not defined correctly, probably missing a key
4117	DMEC	*AE*	Could not start transporter, configuration error
4118	DMEC	*AE*	Parameter error in API call
4300	DMEC	*AE*	Tuple Key Type not correct
4301	DMEC	*AE*	Fragment Type not correct
4302	DMEC	*AE*	Minimum Load Factor not correct
4303	DMEC	*AE*	Maximum Load Factor not correct
4304	DMEC	*AE*	Maximum Load Factor smaller

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
			than Minimum
4305	DMEC	*AE*	K value must currently be set to 6
4306	DMEC	*AE*	Memory Type not correct
4307	DMEC	*AE*	Invalid table name
4308	DMEC	*AE*	Attribute Size not correct
4309	DMEC	*AE*	Fixed array too large, maximum 64000 bytes
4310	DMEC	*AE*	Attribute Type not correct
4311	DMEC	*AE*	Storage Mode not correct
4312	DMEC	*AE*	Null Attribute Type not correct
4313	DMEC	*AE*	Index only storage for non-key attribute
4314	DMEC	*AE*	Storage Type of attribute not correct
4315	DMEC	*AE*	No more key attributes allowed after defining variable length key attribute
4316	DMEC	*AE*	Key attributes are not allowed to be NULL attributes
4317	DMEC	*AE*	Too many primary keys defined in table
4318	DMEC	*AE*	Invalid attribute name or number
4319	DMEC	*AE*	createAttribute called at erroneus place
4322	DMEC	*AE*	Attempt to define distribution key when not prepared to
4323	DMEC	*AE*	Distribution Key set on table but not defined on first attribute
4324	DMEC	*AE*	Attempt to define distribution group when not prepared to
4325	DMEC	*AE*	Distribution Group set on table but not defined on first attribute
4326	DMEC	*AE*	Distribution Group with erroneus number of bits
4327	DMEC	*AE*	Distribution Group with 1 byte attribute is not allowed
4328	DMEC	*AE*	Disk memory attributes not yet supported
4329	DMEC	*AE*	Variable stored attributes not yet supported
4400	DMEC	*AE*	Status Error in NdbSchemaCon
4401	DMEC	*AE*	Only one schema operation per schema transaction
4402	DMEC	*AE*	No schema operation defined before calling execute

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4501	DMEC	*AE*	Insert in hash table failed when getting table information from Ndb
4502	DMEC	*AE*	GetValue not allowed in Update operation
4503	DMEC	*AE*	GetValue not allowed in Insert operation
4504	DMEC	*AE*	SetValue not allowed in Read operation
4505	DMEC	*AE*	NULL value not allowed in primary key search
4506	DMEC	*AE*	Missing getValue/setValue when calling execute
4507	DMEC	*AE*	Missing operation request when calling execute
4200	DMEC	*AE*	Status Error when defining an operation
4201	DMEC	*AE*	Variable Arrays not yet supported
4202	DMEC	*AE*	Set value on tuple key attribute is not allowed
4203	DMEC	*AE*	Trying to set a NOT NULL attribute to NULL
4204	DMEC	*AE*	Set value and Read/Delete Tuple is incompatible
4205	DMEC	*AE*	No Key attribute used to define tuple
4206	DMEC	*AE*	Not allowed to equal key attribute twice
4207	DMEC	*AE*	Key size is limited to 4092 bytes
4208	DMEC	*AE*	Trying to read a non-stored attribute
4209	DMEC	*AE*	Length parameter in equal/setValue is incorrect
4210	DMEC	*AE*	Ndb sent more info than the length he specified
4211	DMEC	*AE*	Inconsistency in list of NdbRecAttr-objects
4212	DMEC	*AE*	Ndb reports NULL value on Not NULL attribute
4213	DMEC	*AE*	Not all data of an attribute has been received
4214	DMEC	*AE*	Not all attributes have been received
4215	DMEC	*AE*	More data received than reported in TCKEYCONF message
4216	DMEC	*AE*	More than 8052 bytes in setValue cannot be handled

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4217	DMEC	*AE*	It is not allowed to increment any other than unsigned ints
4218	DMEC	*AE*	Currently not allowed to increment NULL-able attributes
4219	DMEC	*AE*	Maximum size of interpretative attributes are 64 bits
4220	DMEC	*AE*	Maximum size of interpretative attributes are 64 bits
4221	DMEC	*AE*	Trying to jump to a non-defined label
4222	DMEC	*AE*	Label was not found, internal error
4223	DMEC	*AE*	Not allowed to create jumps to yourself
4224	DMEC	*AE*	Not allowed to jump to a label in a different subroutine
4225	DMEC	*AE*	All primary keys defined, call setValue/getValue
4226	DMEC	*AE*	Bad number when defining a label
4227	DMEC	*AE*	Bad number when defining a subroutine
4228	DMEC	*AE*	Illegal interpreter function in scan definition
4229	DMEC	*AE*	Illegal register in interpreter function definition
4230	DMEC	*AE*	Illegal state when calling getValue, probably not a read
4231	DMEC	*AE*	Illegal state when calling interpreter routine
4233	DMEC	*AE*	Calling execute (synchronous) when already prepared asynchronous transaction exists
4234	DMEC	*AE*	Illegal to call setValue in this state
4235	DMEC	*AE*	No callback from execute
4236	DMEC	*AE*	Trigger name too long
4237	DMEC	*AE*	Too many triggers
4238	DMEC	*AE*	Trigger not found
4239	DMEC	*AE*	Trigger with given name already exists
4240	DMEC	*AE*	Unsupported trigger type
4241	DMEC	*AE*	Index name too long
4242	DMEC	*AE*	Too many indexes
4243	DMEC	*AE*	Index not found
4244	HA_ERR_TABLE_EXIST	*OE*	Index or table with given name already exists
4247	DMEC	*AE*	Illegal index/trigger create/

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
			drop/alter request
4248	DMEC	*AE*	Trigger/index name invalid
4249	DMEC	*AE*	Invalid table
4250	DMEC	*AE*	Invalid index type or index logging option
4251	HA_ERR_FOUND_DUPP_UNIQUE	*AE*	Cannot create unique index, duplicate keys found
4252	DMEC	*AE*	Failed to allocate space for index
4253	DMEC	*AE*	Failed to create index table
4254	DMEC	*AE*	Table not an index table
4255	DMEC	*AE*	Hash index attributes must be specified in same order as table attributes
4258	DMEC	*AE*	Cannot create unique index, duplicate attributes found in definition
4259	DMEC	*AE*	Invalid set of range scan bounds
4260	DMEC	*UD*	NdbScanFilter: Operator is not defined in NdbScanFilter::Group
4261	DMEC	*UD*	NdbScanFilter: Column is NULL
4262	DMEC	*UD*	NdbScanFilter: Condition is out of bounds
4263	DMEC	*IE*	Invalid blob attributes or invalid blob parts table
4264	DMEC	*AE*	Invalid usage of blob attribute
4265	DMEC	*AE*	The method is not valid in current blob state
4266	DMEC	*AE*	Invalid blob seek position
4267	DMEC	*IE*	Corrupted blob value
4268	DMEC	*IE*	Error in blob head update forced rollback of transaction
4269	DMEC	*IE*	No connection to ndb management server
4270	DMEC	*IE*	Unknown blob error
4335	DMEC	*AE*	Only one autoincrement column allowed per table. Having a table without primary key uses an autoincremented hidden key, i.e. a table without a primary key can not have an autoincremented column
4271	DMEC	*AE*	Invalid index object, not retrieved via getIndex()
4272	DMEC	*AE*	Table definition has undefined column
4273	DMEC	*IE*	No blob table in dict cache

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4274	DMEC	*IE*	Corrupted main table PK in blob operation
4275	DMEC	*AE*	The blob method is incompatible with operation type or lock mode

4.2.2. NDB Error Classifications

The following table lists the classification codes used in Section 4.2, "NDB Error Codes, Classifications, and Messages", and their descriptions. These can also be found in the file `/storage/ndb/src/ndbapi/ndberror.c`.

Classification Code	Error Status	Description
NE	*Success*	No error
AE	*Permanent error*	Application error
CE	*Permanent error*	Configuration or application error
ND	*Permanent error*	No data found
CV	*Permanent error*	Constraint violation
SE	*Permanent error*	Schema error
UD	*Permanent error*	User defined error
IS	*Permanent error*	Insufficient space
TR	*Temporary error*	Temporary Resource error
NR	*Temporary error*	Node Recovery error
OL	*Temporary error*	Overload error
TO	*Temporary error*	Timeout expired
NS	*Temporary error*	Node shutdown
IT	*Temporary error*	Internal temporary
UR	*Unknown result*	Unknown result error
UE	*Unknown result*	Unknown error code
IE	*Permanent error*	Internal error
NI	*Permanent error*	Function not implemented

Chapter 5. THE MGM API

This chapter discusses the MySQL Cluster Management API, a C language API that is used for administrative tasks such as starting and stopping Cluster nodes, backups, and logging. It also covers MGM concepts, programming constructs, and event types.

5.1. General Concepts

Each MGM API function needs a management server handle of type `NdbMgmHandle`. This handle is created by calling the function `ndb_mgm_create_handle()` and freed by calling `ndb_mgm_destroy_handle()`.

See Section 5.2.3.1, “`ndb_mgm_create_handle()`”, and Section 5.2.3.3, “`ndb_mgm_destroy_handle()`”, for more information about these two functions.

> **Important**
>
> You should not share an `NdbMgmHandle` between threads. While it is possible to do so (if you implement your own locks), this is not recommended, and each thread should use its own management server handle.

A function can return any of the following:

- An integer value, with a value of `-1` indicating an error.
- A non-constant pointer value. A `NULL` value indicates an error; otherwise, the return value must be freed by the programmer.
- A constant pointer value, with a `NULL` value indicating an error. The returned value should not be freed.

Error conditions can be identified by using the appropriate error-reporting functions `ndb_mgm_get_latest_error()` and `ndb_mgm_error()`.

Here is an example using the MGM API (without error handling for brevity's sake):

```
NdbMgmHandle handle= ndb_mgm_create_handle();
ndb_mgm_connect(handle,0,0,0);
struct ndb_mgm_cluster_state *state= ndb_mgm_get_status(handle);
for(int i=0; i > state->no_of_nodes; i++)
{
  struct ndb_mgm_node_state *node_state= &state->node_states[i];
  printf("node with ID=%d ", node_state->node_id);

  if(node_state->version != 0)
    printf("connected\n");
  else
    printf("not connected\n");
}
free((void*)state);
ndb_mgm_destroy_handle(&handle);
```

5.1.1. Working with Log Events

Data nodes and management servers regularly and on specific occasions report on various log events that occur in the cluster. These log events are written to the cluster log. Optionally an MGM API client may listen to these events using the method `ndb_mgm_listen_event()`. Each log event belongs to a category `ndb_mgm_event_category`) and has a severity `ndb_mgm_event_severity` associ-

ated with it. Each log event also has a level (0-15) associated with it.

Which log events that come out is controlled with `ndb_mgm_listen_event()`, `ndb_mgm_set_clusterlog_loglevel()`, and `ndb_mgm_set_clusterlog_severity_filter()`.

This is an example showing how to listen to events related to backup:

```
int filter[] = { 15, NDB_MGM_EVENT_CATEGORY_BACKUP, 0 };
int fd = ndb_mgm_listen_event(handle, filter);
```

5.1.2. Structured Log Events

The following steps are involved:

1. Create an `NdbEventLogHandle` using `ndb_mgm_create_logevent_handle()`.
2. Wait for and store log events using `ndb_logevent_get_next()`.
3. The log event data is available in the structure `ndb_logevent`. The data which is specific to a particular event is stored in a union between structures; use `ndb_logevent::type` to decide which structure is valid.

The following sample code demonstrates listening to events related to backups:

```
int filter[] = { 15, NDB_MGM_EVENT_CATEGORY_BACKUP, 0 };
NdbEventLogHandle le_handle= ndb_mgm_create_logevent_handle(handle, filter);
struct ndb_logevent le;
int r= ndb_logevent_get_next(le_handle, &le, 0);
if(r < 0)
  /*  error  */
else if(r == 0)
  /*  no event  */

switch(le.type)
{
  case NDB_LE_BackupStarted:
    ... le.BackupStarted.starting_node;
    ... le.BackupStarted.backup_id;
    break;
  case NDB_LE_BackupFailedToStart:
    ... le.BackupFailedToStart.error;
    break;
  case NDB_LE_BackupCompleted:
    ... le.BackupCompleted.stop_gci;
    break;
  case NDB_LE_BackupAborted:
    ... le.BackupStarted.backup_id;
    break;
  default:
    break;
}
```

For more information, see Section 5.2.1, “Log Event Functions”.

> **Note**
>
> Available log event types are listed in Section 5.3.4, “The `Ndb_logevent_type` Type”, as well as in the file `/storage/ndb/include/mgmapi/ndb_logevent.h` in the MySQL 5.1 sources.

5.2. MGM C API Function Listing

This section covers the structures and functions used in the MGM API. Listings are grouped by purpose or use.

5.2.1. Log Event Functions

This section discusses functions that are used for listening to log events.

5.2.1.1. ndb_mgm_listen_event()

Description. This function is used to listen to log events, which are read from the return file descriptor. Events use a text-based format, the same as in the cluster log.

Signature.

```
int ndb_mgm_listen_event
    (
      NdbMgmHandle handle,
      const int    filter[]
    )
```

Parameters. This function takes two arguments:

- An `NdbMgmHandle` *`handle`*.
- A *`filter`* which consists of a series of `{level, ndb_mgm_event_category}` pairs (in a single array) that are pushed to a file descriptor. Use `0` for the level to terminate the list.

Return Value. The file descriptor from which events are to be read.

5.2.1.2. ndb_mgm_create_logevent_handle()

Description. This function is used to create a log event handle.

Signature.

```
NdbLogEventHandle ndb_mgm_create_logevent_handle
    (
      NdbMgmHandle handle,
      const int    filter[]
    )
```

Parameters. This function takes two arguments:

- An `NdbMgmHandle` *`handle`*.
- A *`filter`* which consists of a series of `{level, ndb_mgm_event_category}` pairs (in a single array) that are pushed to a file descriptor. Use `0` for the level to terminate the list.

Return Value. A log event handle.

5.2.1.3. ndb_mgm_destroy_logevent_handle()

Description. Use this function to destroy a log event handle when there is no further need for it.

Signature.

```
void ndb_mgm_destroy_logevent_handle
    (
      NdbLogEventHandle* handle
    )
```

Parameters. A pointer to a log event *`handle`*.

Return Value. *None*.

5.2.1.4. `ndb_logevent_get_fd()`

Description. This function retrieves a file descriptor from an `NdbMgmLogEventHandle`. It was implemented in MySQL 5.1.12.

> **Warning**
>
> Do not attempt to read from the file descriptor returned by this function.

Signature.

```
int ndb_logevent_get_fd
    (
      const NdbLogEventHandle handle
    )
```

Parameters. A `LogEventHandle`.

Return Value. A file descriptor. In the event of failure, `-1` is returned.

5.2.1.5. `ndb_logevent_get_next()`

Description. This function is used to retrieve the next log event, using the event's data to fill in the supplied `ndb_logevent` structure.

Signature.

```
int ndb_logevent_get_next
    (
      const NdbLogEventHandle handle,
      struct ndb_logevent*    logevent,
      unsigned                timeout
    )
```

Parameters. Three parameters are expected by this functions:

- An `NdbLogEventHandle`
- A pointer to an `ndb_logevent` data structure
- The number of milliseconds to wait for the event before timing out

Return Value. The value returned by this function is interpreted as follows:

- `> 0`: The event exists, and it data was retrieved into the *`logevent`*
- `0`: A timeout occurred while waiting for the event (more than *`timeout`* milliseconds elapsed)
- `< 0`: An error occurred.

If the return value is less than or equal to zero, then the *logevent* is not altered or affected in any way.

5.2.1.6. ndb_logevent_get_latest_error()

Description. This function retrieves the error code from the most recent error.

> **Note**
>
> You may prefer to use ndb_logevent_get_latest_error_msg() instead. See Section 5.2.1.7, "ndb_logevent_get_latest_error_msg()"

Signature.

```
int ndb_logevent_get_latest_error
    (
      const NdbLogEventHandle handle
    )
```

Parameters. A log event handle.

Return Value. An error code.

5.2.1.7. ndb_logevent_get_latest_error_msg()

Description. Retrieves the text of the most recent error obtained while trying to read log events.

Signature.

```
const char* ndb_logevent_get_latest_error_msg
    (
      const NdbLogEventHandle handle
    )
```

Parameters. A log event handle.

Return Value. The text of the error message.

5.2.2. MGM API Error Handling Functions

The MGM API used for Error handling are discussed in this section.

Each MGM API error is characterised by an error code and an error message. There may also be an error description that may provide additional information about the error. The API provides functions to obtain this information in the event of an error.

5.2.2.1. ndb_mgm_get_latest_error()

Description. This function is used to get the latest error code associated with a given management server handle.

Signature.

```
int ndb_mgm_get_latest_error
    (
      const NdbMgmHandle handle
    )
```

Parameters. An NdbMgMHandle.

Return Value. An error code corresponding to an ndb_mgm_error value; see Section 5.3.3, "The

ndb_mgm_error Type". You can obtain the related error message using ndb_mgm_get_latest_error_msg(); see Section 5.2.2.2, "ndb_mgm_get_latest_error_msg()".

5.2.2.2. ndb_mgm_get_latest_error_msg()

Description. This function is used to obtain the latest general error message associated with an Ndb-MgmHandle.

Signature.

```
const char* ndb_mgm_get_latest_error_msg
    (
      const NdbMgmHandle handle
    )
```

Parameters. An NdbMgmHandle.

Return Value. The error message text. More specific information can be obtained using ndb_mgm_get_latest_error_desc(); see Section 5.2.2.3, "ndb_mgm_get_latest_error_desc()", for details.

5.2.2.3. ndb_mgm_get_latest_error_desc()

Description. Get the most recent error description associated with an NdbMgmHandle; this description provides additional information regarding the error message.

Signature.

```
const char* ndb_mgm_get_latest_error_desc
    (
      const NdbMgmHandle handle
    )
```

Parameters. An NdbMgmHandle.

Return Value. The error description text.

5.2.2.4. ndb_mgm_set_error_stream()

Description. The function can be used to set the error output stream.

Signature.

```
void ndb_mgm_set_error_stream
    (
      NdbMgmHandle handle,
      FILE*        file
    )
```

Parameters. This function requires two parameters:

- An NdbMgmHandle
- A pointer to the file to which errors are to be sent.

Return Value. *None*.

5.2.3. Management Server Handle Functions

This section contains information about the MGM API functions used to create and destroy management server handles.

5.2.3.1. ndb_mgm_create_handle()

Description. This function is used to create a handle to a management server.

Signature.

```
NdbMgmHandle ndb_mgm_create_handle
    (
      void
    )
```

Parameters. *None*.

Return Value. An `NdbMgmHandle`.

5.2.3.2. ndb_mgm_set_name()

Description. This function can be used to set a name for the management server handle, which is then reported in the Cluster log.

Signature.

```
void ndb_mgm_set_name
    (
      NdbMgmHandle handle,
      const char*  name
    )
```

Parameters. This function takes two arguments:

- A management server *`handle`*.
- The desired *`name`* for the *`handle`*.

Return Value. *None*.

5.2.3.3. ndb_mgm_destroy_handle()

Description. This function destroys a management server handle

Signature.

```
void ndb_mgm_destroy_handle
    (
      NdbMgmHandle* handle
    )
```

Parameters. A pointer to the `NdbMgmHandle` to be destroyed.

Return Value. *None*.

5.2.4. Management Server Connection Functions

This section discusses MGM API functions that are used to initiate, configure, and terminate connections to an NDB management server.

5.2.4.1. ndb_mgm_get_connectstring()

Description. This function retrieves the connectstring used for a connection.

> **Note**
>
> This function returns the default connectstring if no call to `ndb_mgm_set_connectstring()` has been performed. In addition, the returned connectstring may be formatted slightly differently than the original in that it may contain specifiers not present in the original.
>
> The connectstring format is the same as that discussed for Section 5.2.4.6, "`ndb_mgm_set_connectstring()`".

Signature.

```
const char* ndb_mgm_get_connectstring
    (
      NdbMgmHandle handle,
      char*        buffer,
      int          size
    )
```

Parameters. This function takes three arguments:

- An `NdbMgmHandle`.
- A pointer to a *`buffer`* in which to place the result.
- The *`size`* of the buffer.

Return Value. The connectstring — this is the same value that is pushed to the *`buffer`*.

5.2.4.2. ndb_mgm_get_configuration_nodeid()

Description. This function gets the ID of the node to which the connection is being (or was) made.

Signature.

```
int ndb_mgm_get_configuration_nodeid
    (
      NdbMgmHandle handle
    )
```

Parameters. A management server handle.

Return Value. A node ID.

5.2.4.3. ndb_mgm_get_connected_port()

Description. This function retrieves the number of the port used by the connection.

Signature.

```
int ndb_mgm_get_connected_port
```

```
    (
      NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return Value. A port number.

5.2.4.4. ndb_mgm_get_connected_host()

Description. This function is used to obtain the name of the host to which the connection is made.

Signature.

```
const char* ndb_mgm_get_connected_host
    (
      NdbMgmHandle handle
    )
```

Parameters. A management server *`handle`*.

Return Value. A hostname.

5.2.4.5. ndb_mgm_is_connected()

Description. Used to determine whether a connection has been established.

Signature.

```
int ndb_mgm_is_connected
    (
      NdbMgmHandle handle
    )
```

Parameters. A management server *`handle`*.

Return Value. This function returns an integer, whose value is interpreted as follows:

- `0`: Not connected to the management node.
- Any non-zero value: A connection has been established with the management node.

5.2.4.6. ndb_mgm_set_connectstring()

Description. This function is used to set the connectstring for a management server connection to a node.

Signature.

```
int ndb_mgm_set_connectstring
    (
      NdbMgmHandle handle,
      const char*  connectstring
    )
```

Parameters. `ndb_mgm_set_connectstring()` takes two parameters:

- A management server *handle*.
- A *connectstring* whose format is shown here:

```
connectstring :=
    [nodeid-specification,]host-specification[,host-specification]
```

(It is possible to establish connections with multiple management servers using a single connect-string.)

```
nodeid-specification := nodeid=id
host-specification := host[:port]
```

id, *port*, and *host* are defined as follows:

- *id*: An integer greater than `0` identifying a node in `config.ini`.
- *port*: An integer referring to a standard Unix port.
- *host*: A string containing a valid network host address.

Section 5.2.4.1, "`ndb_mgm_get_connectstring()`" also uses this format for connectstrings.

Return Value. This function returns `-1` in the event of failure.

5.2.4.7. `ndb_mgm_set_configuration_nodeid()`

Description. This function sets the connection node ID.

Signature.

```
int ndb_mgm_set_configuration_nodeid
    (
      NdbMgmHandle handle,
      int          id
    )
```

Parameters. This function requires two parameters:

- An `NdbMgmHandle`.
- The `id` of the node to connect to.

Return Value. This function returns `-1` in the event of failure.

5.2.4.8. `ndb_mgm_connect()`

Description. This function establishes a connection to a management server specified by the connect-string set by Section 5.2.4.6, "`ndb_mgm_set_connectstring()`".

Signature.

```
int ndb_mgm_connect
    (
      NdbMgmHandle handle,
      int          retries,
      int          delay,
      int          verbose
    )
```

Parameters. This function takes 4 arguments:

- A management server *`handle`*.
- The number of `retries` to make when attempting to connect. `0` for this value means that one connection attempt is made.
- The number of seconds to *`delay`* between connection attempts.
- If verbose is `1`, then a message is printed for each connection attempt.

Return Value. This function returns `-1` in the event of failure.

5.2.4.9. `ndb_mgm_disconnect()`

Description. This function terminates a management server connection.

Signature.

```
int ndb_mgm_disconnect
    (
      NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return Value. Returns `-1` if unable to disconnect.

5.2.5. Cluster Status Functions

This section discusses how to obtain the status of `NDB` Cluster nodes.

5.2.5.1. `ndb_mgm_get_status()`

Description. This function is used to obtain the status of the nodes in an `NDB` Cluster.

> **Note**
>
> The caller must free the pointer returned by this function.

Signature.

```
struct ndb_mgm_cluster_state* ndb_mgm_get_status
    (
      NdbMgmHandle handle
    )
```

Parameters. This function takes a single parameter — a management server *`handle`*.

Return Value. A pointer to an `ndb_mgm_cluster_state` data structure. See Section 5.4.3, “The `ndb_mgm_cluster_state` Structure”, for more information.

5.2.6. Functions for Starting & Stopping Nodes

The MGM API provides several functions which can be used to start, stop, and restart one or more Cluster data nodes. These functions are discussed in this section.

Starting, Stopping, and Restarting Nodes. You can start, stop, and restart Cluster nodes using the following functions:

- **Starting Nodes.** Use `ndb_mgm_start()`.
- **Stopping Nodes.** Use `ndb_mgm_stop()`, `ndb_mgm_stop2()`, or `ndb_mgm_stop3()`.
- **Restarting Nodes.** Use `ndb_mgm_restart()`, `ndb_mgm_restart2()`, or `ndb_mgm_restart3()`.

These functions are detailed in the next few sections.

5.2.6.1. `ndb_mgm_start()`

Description. This function can be used to start one or more Cluster nodes. The nodes to be started must have been started with the no-start option (`-n`), meaning that the data node binary was started and is waiting for a `START` management command which actually enables the node.

Signature.

```
int ndb_mgm_start
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list
    )
```

Parameters. ndb_mgm_start() takes 3 parameters:

- An `NdbMgmHandle`.
- A *`number`* of nodes to be started. Use `0` to start all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be started.

Return Value. The number of nodes actually started; in the event of failure, `-1` is returned.

5.2.6.2. `ndb_mgm_stop()`

Description. This function stops one or more data nodes.

Signature.

```
int ndb_mgm_stop
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list
    )
```

Parameters. ndb_mgm_stop() takes 3 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.

Calling this function is equivalent to calling `ndb_mgm_stop2(handle, number, list, 0)`. See Section 5.2.6.3, "`ndb_mgm_stop2()`".

Return Value. The number of nodes actually stopped; in the event of failure, `-1` is returned.

5.2.6.3. `ndb_mgm_stop2()`

Description. Like `ndb_mgm_stop()`, this function stops one or more data nodes. However, it offers the ability to specify whether or not the nodes shut down gracefully.

Signature.

```
int ndb_mgm_stop2
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          abort
    )
```

Parameters. ndb_mgm_stop2() takes 4 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.
- The value of *`abort`* determines how the nodes will be shut down. `1` indicates the nodes will shut down immediately; `0` indicates that the nodes will stop gracefully.

Return Value. The number of nodes actually stopped; in the event of failure, `-1` is returned.

5.2.6.4. `ndb_mgm_stop3()`

Description. Like `ndb_mgm_stop()` and `ndb_mgm_stop2()`, this function stops one or more data nodes. Like `ndb_mgm_stop2()`, it offers the ability to specify whether the nodes should shut down gracefully. In addition, it provides for a way to check to see whether disconnection is required prior to stopping a node.

Signature.

```
int ndb_mgm_stop3
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          abort,
      int*         disconnect
    )
```

Parameters. ndb_mgm_stop3() takes 5 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.

- The value of *`abort`* determines how the nodes will be shut down. `1` indicates the nodes will shut down immediately; `0` indicates that the nodes will stop gracefully.
- If *`disconnect`* returns `1` (`true`), this means the you must disconnect before you can apply the command to stop. For example, disconnecting is required when stopping the management server to which the handle is connected.

Return Value. The number of nodes actually stopped; in the event of failure, `-1` is returned.

5.2.6.5. `ndb_mgm_restart()`

Description. This function can be used to restart one or more Cluster data nodes.

Signature.

```
int ndb_mgm_restart
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list
    )
```

Parameters. ndb_mgm_restart() takes 3 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.

Calling this function is equivalent to calling

```
ndb_mgm_restart2(handle, number, list, 0, 0, 0);
```

See Section 5.2.6.6, “`ndb_mgm_restart2()`”, for more information.

Return Value. The number of nodes actually restarted; `-1` on failure.

5.2.6.6. `ndb_mgm_restart2()`

Description. Like `ndb_mgm_restart()`, this function can be used to restart one or more Cluster data nodes. However, `ndb_mgm_restart2()` provides additional restart options, including initial restart, waiting start, and immediate (forced) restart.

Signature.

```
int ndb_mgm_restart2
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          initial
      int          nostart,
      int          abort
    )
```

Parameters. ndb_mgm_restart2() takes 6 parameters:

- An `NdbMgmHandle`.

- The *number* of nodes to be stopped. Use 0 to stop all of the data nodes in the cluster.
- A *list* of the node IDs of the nodes to be stopped.
- If *initial* is true (1), then each node undergoes an initial restart — that is, its filesystem is removed.
- If nostart is true, then the nodes are not actually started, but instead are left ready for a start command.
- If *abort* is true, then the nodes are restarted immediately, bypassing any graceful restart.

Return Value. The number of nodes actually restarted; -1 on failure.

5.2.6.7. ndb_mgm_restart3()

Description. Like ndb_mgm_restart2(), this function can be used to cause an initial restart, waiting restart, and immediate (forced) restart on one or more Cluster data nodes. However, ndb_mgm_restart3() provides additional the additional options of checking whether disconnection is required prior to the restart.

Signature.

```
int ndb_mgm_restart3
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          initial
      int          nostart,
      int          abort,
      int*         disconnect
    )
```

Parameters. ndb_mgm_restart() takes 7 parameters:

- An NdbMgmHandle.
- The *number* of nodes to be stopped. Use 0 to stop all of the data nodes in the cluster.
- A *list* of the node IDs of the nodes to be stopped.
- If *initial* is true (1), then each node undergoes an initial restart — that is, its filesystem is removed.
- If nostart is true, then the nodes are not actually started, but instead are left ready for a start command.
- If *abort* is true, then the nodes are forced to restart immediately without performing a graceful restart.
- If *disconnect* returns 1 (true), this means the you must disconnect before you can apply the command to restart. For example, disconnecting is required when stopping the management server to which the handle is connected.

Return Value. The number of nodes actually restarted; -1 on failure.

5.2.7. Cluster Log Functions

This section covers the functions available in the MGM API for controlling the output of the cluster log.

5.2.7.1. ndb_mgm_get_clusterlog_severity_filter()

Description. This function is used to retrieve the cluster log severity filter currently in force.

Signature.

```
const unsigned int* ndb_mgm_get_clusterlog_severity_filter
    (
      NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return Value. A *severity filter*, which is a vector containing 7 elements. Each element equals `1` if the corresponding severity indicator is enabled, and `0` if it is not. A severity level is stored at position *`ndb_mgm_clusterlog_level`* — for example, the "error" level is stored at position `NDB_MGM_EVENT_SEVERITY_ERROR`. The first element in the vector (`NDB_MGM_EVENT_SEVERITY_ON`) signals whether the cluster log is enabled or disabled.

5.2.7.2. ndb_mgm_set_clusterlog_severity_filter()

Description. This function is used to set a cluster log severity filter.

Signature.

```
int ndb_mgm_set_clusterlog_severity_filter
    (
      NdbMgmHandle                handle,
      enum ndb_mgm_event_severity severity,
      int                         enable,
      struct ndb_mgm_reply*       reply
    )
```

Parameters. This function takes 4 parameters:

- A management server *`handle`*.
- A cluster log *`severity`* to filter.
- A flag to *`enable`* or disable the filter; `1` enables and `0` disables the filter.
- A pointer to an `ndb_mgm_reply` structure for a reply message. See Section 5.4.4, "The `ndb_mgm_reply` Structure".

Return Value. The function returns `-1` in the event of failure.

5.2.7.3. ndb_mgm_set_clusterlog_loglevel()

Description. This function is used to set the log category and levels for the cluster log.

Signature.

```
int ndb_mgm_set_clusterlog_loglevel
    (
      NdbMgmHandle                handle,
```

```
      int                          id,
      enum ndb_mgm_event_category category,
      int                          level,
      struct ndb_mgm_reply*        reply)
```

Parameters. This function takes 5 parameters:

- An `NdbMgmHandle`.
- The *`id`* of the node affected.
- An event *`category`* — this is one of the values listed in Section 5.3.7, “The `ndb_mgm_event_category` Type” .
- A logging *`level`*.
- A pointer to an `ndb_mgm_reply` structure for the *`reply`* message. (See Section 5.4.4, “The `ndb_mgm_reply` Structure”.)

Return Value. In the event of an error, this function returns `-1`.

5.2.8. Backup Functions

This section covers the functions provided in the MGM API for starting and stopping backups.

5.2.8.1. `ndb_mgm_start_backup()`

Description. This function is used to initiate a backup of a MySQL Cluster.

Signature.

```
int ndb_mgm_start_backup
    (
      NdbMgmHandle          handle,
      int                   wait,
      unsigned int*         id,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function requires 4 parameters:

- A management server *`handle`* (an `NdbMgmHandle`).
- A *`wait`* flag, with the following possible values:
  - `0`: Do not wait for confirmation of the backup.
  - `1`: Wait for the backup to be started.
  - `2`: Wait for the backup to be completed.
- A backup *`id`* to be returned by the function.

  > **Note**
  >
  > No backup *`id`* is returned if *`wait`* is set equal to 0.

- A pointer to an `ndb_mgm_reply` structure to accommodate a *`reply`*. See Section 5.4.4, “The

`ndb_mgm_reply` Structure”.

Return Value. In the event of failure, the function returns `-1`.

5.2.8.2. `ndb_mgm_abort_backup()`

Description. This function is used to stop a Cluster backup.

Signature.

```
int ndb_mgm_abort_backup
    (
      NdbMgmHandle          handle,
      unsigned int          id,
      struct ndb_mgm_reply* reply)
```

Parameters. This function takes 3 parameters:

- An `NdbMgmHandle`.
- The *`id`* of the backup to be aborted.
- A pointer to an `ndb_mgm_reply` structure.

Return Value. In case an error, this function returns `-1`.

5.2.9. Single-User Mode Functions

The MGM API allows the programmer to put the cluster into single-user mode — and to return it to normal mode again — from within an application. This section covers the functions that are used for these operations.

5.2.9.1. `ndb_mgm_enter_single_user()`

Description. This function is used to enter single-user mode on a given node.

Signature.

```
int ndb_mgm_enter_single_user
    (
      NdbMgmHandle          handle,
      unsigned int          id,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function takes 3 parameters:

- An `NdbMgmHandle`.
- The *`id`* of the node to be used in single-user mode.
- A pointer to an `ndb_mgm_reply` structure, used for a *`reply`* message.

Return Value. Returns `-1` in the event of failure.

5.2.9.2. ndb_mgm_exit_single_user()

Description. This function is used to exit single-user mode and to return to normal operation.

Signature.

```
int ndb_mgm_exit_single_user
    (
      NdbMgmHandle          handle,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function requires 2 arguments:

- An NdbMgmHandle.
- A pointer to an ndb_mgm_reply.

Return Value. Returns -1 in case of an error.

5.3. MGM Datatypes

This section discusses the datatypes defined by the MGM API.

> **Note**
>
> The types described in this section are all defined in the file /storage/ndb/include/mgmapi/mgmapi.h, with the exception of Ndb_logevent_type, ndb_mgm_event_severity, ndb_mgm_logevent_handle_error, and ndb_mgm_event_category, which are defined in /storage/ndb/include/mgmapi/ndb_logevent.h.

5.3.1. The ndb_mgm_node_type Type

Description. This is used to classify the different types of nodes in a MySQL Cluster.

Enumeration Values.

Value	Description
NDB_MGM_NODE_TYPE_UNKNOWN	Unknown
NDB_MGM_NODE_TYPE_API	API Node (SQL node)
NDB_MGM_NODE_TYPE_NDB	Data node
NDB_MGM_NODE_TYPE_MGM	Management node

5.3.2. The ndb_mgm_node_status Type

Description. This type describes a Cluster node's status.

Enumeration Values.

Value	Description
NDB_MGM_NODE_STATUS_UNKNOWN	The node's status is not known

Value	Description
NDB_MGM_NODE_STATUS_NO_CONTACT	The node cannot be contacted
NDB_MGM_NODE_STATUS_NOT_STARTED	The node has not yet executed the startup protocol
NDB_MGM_NODE_STATUS_STARTING	The node is executing the startup protocol
NDB_MGM_NODE_STATUS_STARTED	The node is running
NDB_MGM_NODE_STATUS_SHUTTING_DOWN	The node is shutting down
NDB_MGM_NODE_STATUS_RESTARTING	The node is restarting
NDB_MGM_NODE_STATUS_SINGLEUSER	The node is running in single-user (maintenance) mode
NDB_MGM_NODE_STATUS_RESUME	The node is in resume mode

5.3.3. The ndb_mgm_error Type

Description. The values for this type are the error codes that may be generated by MGM API functions.

Enumeration Values.

Type	Value	Description
Service Request Errors		
	NDB_MGM_ILLEGAL_CONNECT_STRING	Invalid connectstring
	NDB_MGM_ILLEGAL_SERVER_HANDLE	Invalid management server handle
	NDB_MGM_ILLEGAL_SERVER_REPLY	Invalid response from management server
	NDB_MGM_ILLEGAL_NUMBER_OF_NODES	Invalid number of nodes
	NDB_MGM_ILLEGAL_NODE_STATUS	Invalid node status
	NDB_MGM_OUT_OF_MEMORY	Memory allocation error
	NDB_MGM_SERVER_NOT_CONNECTED	Management server not connected
	NDB_MGM_COULD_NOT_CONNECT_TO_SOCKET	Not able to connect to socket
Node ID Allocation Errors		
	NDB_MGM_ALLOCID_ERROR	Generic error; may be possible to retry and recover
	NDB_MGM_ALLOCID_CONFIG_MISMATCH	Non-recoverable generic error
Service Errors	(Failure of a node or cluster to start, shut down, or restart	
	NDB_MGM_START_FAILED	Startup failure
	NDB_MGM_STOP_FAILED	Shutdown failure
	NDB_MGM_RESTART_FAILED	Restart failure
Backup Service Errors		
	NDB_MGM_COULD_NOT_START_BACKUP	Unable to initiate backup

Type	Value	Description
	NDB_MGM_COULD_NOT_ABORT_BACKUP	Unable to abort backup
Single-User Mode Service Errors		
	NDB_MGM_COULD_NOT_ENTER_SINGLE_USER_MODE	Unable to enter single-user mode
	NDB_MGM_COULD_NOT_EXIT_SINGLE_USER_MODE	Unable to exit single-user mode
Usage Errors		
	NDB_MGM_USAGE_ERROR	General usage error

See Section 5.2.2.1, "ndb_mgm_get_latest_error()".

5.3.4. The Ndb_logevent_type Type

Description. These are the types of log event events available in the MGM API, grouped by event category. (See Section 5.3.7, "The ndb_mgm_event_category Type".)

Enumeration Values.

Category	Type
NDB_MGM_EVENT_CATEGORY_CONNECTION	
	NDB_LE_Connected
	NDB_LE_Disconnected
	NDB_LE_CommunicationClosed
	NDB_LE_CommunicationOpened
	NDB_LE_ConnectedApiVersion
NDB_MGM_EVENT_CATEGORY_CHECKPOINT	
	NDB_LE_GlobalCheckpointStarted
	NDB_LE_GlobalCheckpointCompleted
	NDB_LE_LocalCheckpointStarted
	NDB_LE_LocalCheckpointCompleted
	NDB_LE_LCPStoppedInCalcKeepGci
	NDB_LE_LCPFragmentCompleted
NDB_MGM_EVENT_CATEGORY_STARTUP	
	NDB_LE_NDBStartStarted
	NDB_LE_NDBStartCompleted
	NDB_LE_STTORRYRecieved
	NDB_LE_StartPhaseCompleted
	NDB_LE_CM_REGCONF
	NDB_LE_CM_REGREF
	NDB_LE_FIND_NEIGHBOURS
	NDB_LE_NDBStopStarted
	NDB_LE_NDBStopCompleted

Category	Type
	NDB_LE_NDBStopForced
	NDB_LE_NDBStopAborted
	NDB_LE_StartREDOLog
	NDB_LE_StartLog
	NDB_LE_UNDORecordsExecuted
	NDB_LE_StartReport
NDB_MGM_EVENT_CATEGORY_NODE_RESTART	
	NDB_LE_NR_CopyDict
	NDB_LE_NR_CopyDistr
	NDB_LE_NR_CopyFragsStarted
	NDB_LE_NR_CopyFragDone
	NDB_LE_NR_CopyFragsCompleted
NDB_MGM_EVENT_CATEGORY_NODE_RESTART	
	NDB_LE_NodeFailCompleted
	NDB_LE_NODE_FAILREP
	NDB_LE_ArbitState
	NDB_LE_ArbitResult
	NDB_LE_GCP_TakeoverStarted
	NDB_LE_GCP_TakeoverCompleted
	NDB_LE_LCP_TakeoverStarted
	NDB_LE_LCP_TakeoverCompleted
NDB_MGM_EVENT_CATEGORY_STATISTIC	
	NDB_LE_TransReportCounters
	NDB_LE_OperationReportCounters
	NDB_LE_TableCreated
	NDB_LE_UndoLogBlocked
	NDB_LE_JobStatistic
	NDB_LE_SendBytesStatistic
	NDB_LE_ReceiveBytesStatistic
	NDB_LE_MemoryUsage
NDB_MGM_EVENT_CATEGORY_ERROR	
	NDB_LE_TransporterError
	NDB_LE_TransporterWarning
	NDB_LE_MissedHeartbeat
	NDB_LE_DeadDueToHeartbeat
	NDB_LE_WarningEvent
NDB_MGM_EVENT_CATEGORY_INFO	
	NDB_LE_SentHeartbeat
	NDB_LE_CreateLogBytes

Category	Type
	NDB_LE_InfoEvent
[Single-User Mode]	NDB_LE_SingleUser
	NDB_LE_EventBufferStatus
NDB_MGM_EVENT_CATEGORY_BACKUP	
	NDB_LE_BackupStarted
	NDB_LE_BackupFailedToStart
	NDB_LE_BackupCompleted
	NDB_LE_BackupAborted

5.3.5. The ndb_mgm_event_severity Type

Description. These are the log event severities used to filter the cluster log by ndb_mgm_set_clusterlog_severity_filter(), and to filter listening to events by ndb_mgm_listen_event().

Enumeration Values.

Value	Description
NDB_MGM_ILLEGAL_EVENT_SEVERITY	Invalid event severity specified
NDB_MGM_EVENT_SEVERITY_ON	Cluster logging is enabled
NDB_MGM_EVENT_SEVERITY_DEBUG	*Used for MySQL Cluster development only*
NDB_MGM_EVENT_SEVERITY_INFO	Informational messages
NDB_MGM_EVENT_SEVERITY_WARNING	Conditions that are not errors as such, but that might require special handling
NDB_MGM_EVENT_SEVERITY_ERROR	Non-fatal error conditions that should be corrected
NDB_MGM_EVENT_SEVERITY_CRITICAL	Critical conditions such as device errors or out of memory errors
NDB_MGM_EVENT_SEVERITY_ALERT	Conditions that require immediate attention, such as corruption of the cluster
NDB_MGM_EVENT_SEVERITY_ALL	All severity levels

See Section 5.2.7.2, “ndb_mgm_set_clusterlog_severity_filter()”, and Section 5.2.1.1, “ndb_mgm_listen_event()”, for information on how this type is used by those functions.

5.3.6. The ndb_logevent_handle_error Type

Description. This type is used to describe log event errors.

Enumeration Values.

Value	Description
NDB_LEH_NO_ERROR	No error
NDB_LEH_READ_ERROR	Read error
NDB_LEH_MISSING_EVENT_SPECIFIER	Invalid, incomplete, or missing log event specification

Value	Description
NDB_LEH_UNKNOWN_EVENT_TYPE	Unknown log event type
NDB_LEH_UNKNOWN_EVENT_VARIABLE	Unknown log event variable
NDB_LEH_INTERNAL_ERROR	Internal error

5.3.7. The ndb_mgm_event_category Type

Description. These are the log event categories referenced in Section 5.3.4, "The Ndb_logevent_type Type". They are also used by the MGM API functions ndb_mgm_set_clusterlog_loglevel() and ndb_mgm_listen_event().

Enumeration Values.

Value	Description
NDB_MGM_ILLEGAL_EVENT_CATEGORY	Invalid log event category
NDB_MGM_EVENT_CATEGORY_STARTUP	Log events occurring during startup
NDB_MGM_EVENT_CATEGORY_SHUTDOWN	Log events occurring during shutdown
NDB_MGM_EVENT_CATEGORY_STATISTIC	Statistics log events
NDB_MGM_EVENT_CATEGORY_CHECKPOINT	Log events related to checkpoints
NDB_MGM_EVENT_CATEGORY_NODE_RESTART	Log events occurring during node restart
NDB_MGM_EVENT_CATEGORY_CONNECTION	Log events relating to connections between cluster nodes
NDB_MGM_EVENT_CATEGORY_BACKUP	Log events relating to backups
NDB_MGM_EVENT_CATEGORY_CONGESTION	Log events relating to congestion
NDB_MGM_EVENT_CATEGORY_INFO	Uncategorised log events (severity level INFO)
NDB_MGM_EVENT_CATEGORY_ERROR	Uncategorised log events (severity level WARNING, ERROR, CRITICAL, or ALERT)

See Section 5.2.7.3, "ndb_mgm_set_clusterlog_loglevel()", and Section 5.2.1.1, "ndb_mgm_listen_event()", for more information.

5.4. MGM Structures

This section covers the programming structures available in the MGM API.

5.4.1. The ndb_logevent Structure

Description. This structure models a Cluster log event, and is used for storing and retrieving log event information.

Definition. ndb_logevent has 8 members, the first 7 of which are shown in the following list:

- void* *handle*: An NdbLogEventHandle, set by ndb_logevent_get_next(). This handle is used only for purposes of comparison.

  See Section 5.2.1.5, "ndb_logevent_get_next()".

- `enum Ndb_logevent_type type`: Tells which type of event this is.

  See Section 5.3.4, “The Ndb_logevent_type Type”, for possible values.

- `unsigned time`: The time at which the log event was registered with the management server.

- `enum ndb_mgm_event_category category`: The log event category.

  See Section 5.3.7, “The ndb_mgm_event_category Type”, for possible values.

- `enum ndb_mgm_event_severity severity`: The log event severity.

  See Section 5.3.5, “The ndb_mgm_event_severity Type”, for possible values.

- `unsigned level`: The log event level. This is a value in the range of 0 to 15, inclusive.

- `unsigned source_nodeid`: The node ID of the node that reported this event.

The 8th member of this structure contains data specific to the log event, and is dependent on its type. It is defined as the union of a number of data structures, each corresponding to a log event type. Which structure to use is determined by the value of `type`, and is shown in the following table:

`Ndb_logevent_type` Value	**Structure**
`NDB_LE_Connected`	`Connected`: `unsigned node`
`NDB_LE_Disconnected`	`Disconnected`: `unsigned node`
`NDB_LE_CommunicationClosed`	`CommunicationClosed`: `unsigned node`
`NDB_LE_CommunicationOpened`	`CommunicationOpened`: `unsigned node`

`NDB_LE_ConnectedApiVersion`	`ConnectedApiVersion`:
	`unsigned node`
	`unsigned version`

`NDB_LE_GlobalCheckpointStarted`	`GlobalCheckpointStarted`: `unsigned gci`
`NDB_LE_GlobalCheckpointCompleted`	`GlobalCheckpointCompleted`: `unsigned gci`

`NDB_LE_LocalCheckpointStarted`	`LocalCheckpointStarted`:
	`unsigned lci`
	`unsigned keep_gci`
	`unsigned restore_gci`

`NDB_LE_LocalCheckpointCompleted`	`LocalCheckpointCompleted`: `unsigned lci`

Ndb_logevent_type Value	**Structure**
NDB_LE_LCPStoppedInCalcKeepGci	LCPStoppedInCalcKeepGci: unsigned *data*
NDB_LE_LCPFragmentCompleted	LCPFragmentCompleted: unsigned *node* unsigned *table_id* unsigned *fragment_id*
NDB_LE_UndoLogBlocked	UndoLogBlocked: unsigned *acc_count* unsigned *tup_count*
NDB_LE_NDBStartStarted	NDBStartStarted: unsigned *version*
NDB_LE_NDBStartCompleted	NDBStartCompleted: unsigned *version*
NDB_LE_STTORRYRecieved	STTORRYRecieved: *[NONE]*
NDB_LE_StartPhaseCompleted	StartPhaseCompleted: unsigned *phase* unsigned *starttype*
NDB_LE_CM_REGCONF	CM_REGCONF: unsigned *own_id* unsigned *president_id* unsigned *dynamic_id*
NDB_LE_CM_REGREF	CM_REGREF: unsigned *own_id* unsigned *other_id* unsigned *cause*
NDB_LE_FIND_NEIGHBOURS	FIND_NEIGHBOURS: unsigned *own_id* unsigned *left_id* unsigned *right_id* unsigned *dynamic_id*
NDB_LE_NDBStopStarted	NDBStopStarted: unsigned *stoptype*
NDB_LE_NDBStopCompleted	NDBStopCompleted: unsigned *action* unsigned *signum*
NDB_LE_NDBStopForced	NDBStopForced: unsigned *action*

Ndb_logevent_type Value	**Structure**
	unsigned *signum* unsigned *error* unsigned *sphase* unsigned *extra*
NDB_LE_NDBStopAborted	NDBStopAborted: *[NONE]*
NDB_LE_StartREDOLog	StartREDOLog: unsigned *node* unsigned *keep_gci* unsigned *completed_gci* unsigned *restorable_gci*
NDB_LE_StartLog	StartLog: unsigned *log_part* unsigned *start_mb* unsigned *stop_mb* unsigned *gci*
NDB_LE_UNDORecordsExecuted	UNDORecordsExecuted: unsigned *block* unsigned *data1* unsigned *data2* unsigned *data3* unsigned *data4* unsigned *data5* unsigned *data6* unsigned *data7* unsigned *data8* unsigned *data9* unsigned *data10*
NDB_LE_NR_CopyDict	NR_CopyDict: *[NONE]*
NDB_LE_NR_CopyDistr	NR_CopyDistr: *[NONE]*
NDB_LE_NR_CopyFragsStarted	NR_CopyFragsStarted: unsigned *dest_node*
NDB_LE_NR_CopyFragDone	NR_CopyFragDone: unsigned *dest_node* unsigned *table_id* unsigned *fragment_id*
NDB_LE_NR_CopyFragsCompleted	NR_CopyFragsCompleted: unsigned *dest_node*
NDB_LE_NodeFailCompleted	NodeFailCompleted: unsigned *block* unsigned *failed_node* unsigned *completing_node* (For *block* and *completing_node*, 0 is inter-

Ndb_logevent_type Value	**Structure**
	preted as “all”.)
NDB_LE_NODE_FAILREP	NODE_FAILREP: unsigned *failed_node* unsigned *failure_state*
NDB_LE_ArbitState	ArbitState: unsigned *code* unsigned *arbit_node* unsigned *ticket_0* unsigned *ticket_1*
NDB_LE_ArbitResult	ArbitResult: unsigned *code* unsigned *arbit_node* unsigned *ticket_0* unsigned *ticket_1*
NDB_LE_GCP_TakeoverStarted	GCP_TakeoverStarted: *[NONE]*
NDB_LE_GCP_TakeoverCompleted	GCP_TakeoverCompleted: *[NONE]*
NDB_LE_LCP_TakeoverStarted	LCP_TakeoverStarted: *[NONE]*
NDB_LE_TransReportCounters	TransReportCounters: unsigned *trans_count* unsigned *commit_count* unsigned *read_count* unsigned *simple_read_count* unsigned *write_count* unsigned *attrinfo_count* unsigned *conc_op_count* unsigned *abort_count* unsigned *scan_count* unsigned *range_scan_count*
NDB_LE_OperationReportCounters	OperationReportCounters: unsigned *ops*
NDB_LE_TableCreated	TableCreated: unsigned *table_id*
NDB_LE_JobStatistic	JobStatistic: unsigned *mean_loop_count*
NDB_LE_SendBytesStatistic	SendBytesStatistic: unsigned *to_node* unsigned *mean_sent_bytes*
NDB_LE_ReceiveBytesStatistic	ReceiveBytesStatistic:

Ndb_logevent_type Value	**Structure**
	unsigned *from_node* unsigned *mean_received_bytes*
NDB_LE_MemoryUsage	MemoryUsage: int *gth* unsigned *page_size_kb* unsigned *pages_used* unsigned *pages_total* unsigned *block*
NDB_LE_TransporterError	TransporterError: unsigned *to_node* unsigned *code*
NDB_LE_TransporterWarning	TransporterWarning: unsigned *to_node* unsigned *code*
NDB_LE_MissedHeartbeat	MissedHeartbeat: unsigned *node* unsigned *count*
NDB_LE_DeadDueToHeartbeat	DeadDueToHeartbeat: unsigned *node*
NDB_LE_WarningEvent	WarningEvent: *[NOT YET IMPLEMENTED]*
NDB_LE_SentHeartbeat	SentHeartbeat: unsigned *node*
NDB_LE_CreateLogBytes	CreateLogBytes: unsigned *node*
NDB_LE_InfoEvent	InfoEvent: *[NOT YET IMPLEMENTED]*
NDB_LE_EventBufferStatus	EventBufferStatus: unsigned *usage* unsigned *alloc* unsigned *max* unsigned *apply_gci_l* unsigned *apply_gci_h* unsigned *latest_gci_l* unsigned *latest_gci_h*
NDB_LE_BackupStarted	BackupStarted: unsigned *starting_node* unsigned *backup_id*
NDB_LE_BackupFailedToStart	BackupFailedToStart:

Ndb_logevent_type Value	Structure
	unsigned *starting_node* unsigned *error*
NDB_LE_BackupCompleted	BackupCompleted: unsigned *starting_node* unsigned *backup_id* unsigned *start_gci* unsigned *stop_gci* unsigned *n_records* unsigned *n_log_records* unsigned *n_bytes* unsigned *n_log_bytes*
NDB_LE_BackupAborted	BackupAborted: unsigned *starting_node* unsigned *backup_id* unsigned *error*
NDB_LE_SingleUser	SingleUser: unsigned *type* unsigned *node_id*
NDB_LE_StartReport	StartReport: unsigned *report_type* unsigned *remaining_time* unsigned *bitmask_size* unsigned *bitmask_data[1]*

5.4.2. The ndb_mgm_node_state Structure

Description. Provides information on the status of a Cluster node.

Definition. This structure contains the following members:

- int *node_id*: The cluster node's node ID.
- enum ndb_mgm_node_type *node_type*: The node type.

  See Section 5.3.1, "The ndb_mgm_node_type Type", for permitted values.
- enum ndb_mgm_node_status *node_status*: The node's status.

  See Section 5.3.2, "The ndb_mgm_node_status Type", for permitted values.
- int *start_phase*: The start phase.

  This is valid only if the *node_type* is NDB_MGM_NODE_TYPE_NDB and the *node_status* is NDB_MGM_NODE_STATUS_STARTING.
- int *dynamic_id*: The ID for heartbeats and master takeover.

  Valid only for data (ndbd) nodes.
- int *node_group*: The node group to which the node belongs.

Valid only for data (ndbd) nodes.

- `int version`: Internal version number.
- `int connect_count`: The number of times this node has connected to or disconnected from the management server.
- `char connect_address[]`: The IP address of the node when it connected to the management server.

  This value will be empty if the management server has been restarted since the node last connected.

5.4.3. The `ndb_mgm_cluster_state` Structure

Description. Provides information on the status of all Cluster nodes. This structure is returned by `ndb_mgm_get_status()`.

Definition. This structure has the following two members;

- `int no_of_nodes`: The number of elements in the `node_states` array.
- `struct ndb_mgm_node_state node_states[]`: An array containing the states of the nodes.

  Each element of this array is an `ndb_mgm_node_state` structure. For more information, see Section 5.4.2, "The `ndb_mgm_node_state` Structure".

See Section 5.2.5.1, "`ndb_mgm_get_status()`".

5.4.4. The `ndb_mgm_reply` Structure

Description. Contains response information, consisting of a response code and a corresponding message, from the management server.

Definition. This structure contains two members, as shown here:

- `int return_code`: For a successful operation, this value is `0`; otherwise, it contains an error code.

  For error codes, see Section 5.3.3, "The `ndb_mgm_error` Type".
- `char message[256]`: contains the text of the response or error message.

See Section 5.2.2.1, "`ndb_mgm_get_latest_error()`", and Section 5.2.2.2, "`ndb_mgm_get_latest_error_msg()`".

Chapter 6. PRACTICAL EXAMPLES

This section provides code examples illustrating how to accomplish some basic tasks using the NDB API.

All of these examples can be compiled and run as provided, and produce sample output to demonstrate their effects.

6.1. Using Synchronous Transactions

This example illustrates the use of synchronous transactions in the NDB API.

The source code for this example can be found in `storage/ndb/ndbapi-examples/ndbapi_simple/ndbapi_simple` in the MySQL 5.1 tree.

The correct output from this program is as follows:

```
 ATTR1  ATTR2
 0      10
 1       1
 2      12
 Detected that deleted tuple doesn't exist!
 4      14
 5       5
 6      16
 7       7
 8      18
 9       9
```

```
#include <mysql.h>
#include <NdbApi.hpp>
// Used for cout
#include <stdio.h>
#include <iostream>

static void run_application(MYSQL &, Ndb_cluster_connection &);

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  // ndb_init must be called first
  ndb_init();

  // connect to mysql server and cluster and run application
  {
    char * mysqld_sock  = argv[1];
    const char *connectstring = argv[2];
    // Object representing the cluster
    Ndb_cluster_connection cluster_connection(connectstring);

    // Connect to cluster management server (ndb_mgmd)
    if (cluster_connection.connect(4 /* retries               */,
          5 /* delay between retries */,
          1 /* verbose               */))
    {
```

```
      std::cout << "Cluster management server was not ready within 30 secs.\n";
      exit(-1);
    }

    // Optionally connect and wait for the storage nodes (ndbd's)
    if (cluster_connection.wait_until_ready(30,0) < 0)
    {
      std::cout << "Cluster was not ready within 30 secs.\n";
      exit(-1);
    }

    // connect to mysql server
    MYSQL mysql;
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
           0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    // run the application code
    run_application(mysql, cluster_connection);
  }

  ndb_end(0);

  return 0;
}

static void create_table(MYSQL &);
static void drop_table(MYSQL &);
static void do_insert(Ndb &);
static void do_update(Ndb &);
static void do_delete(Ndb &);
static void do_read(Ndb &);

static void run_application(MYSQL &mysql,
          Ndb_cluster_connection &cluster_connection)
{
  /********************************************
   * Connect to database via mysql-c          *
   ********************************************/
  mysql_query(&mysql, "CREATE DATABASE TEST_DB_1");
  if (mysql_query(&mysql, "USE TEST_DB_1") != 0) MYSQLERROR(mysql);
  create_table(mysql);

  /********************************************
   * Connect to database via NdbApi           *
   ********************************************/
  // Object representing the database
  Ndb myNdb( &cluster_connection, "TEST_DB_1" );
  if (myNdb.init()) APIERROR(myNdb.getNdbError());

  /*
   * Do different operations on database
   */
  do_insert(myNdb);
  do_update(myNdb);
  do_delete(myNdb);
  do_read(myNdb);
  drop_table(mysql);
  mysql_query(&mysql, "DROP DATABASE TEST_DB_1");
}

/*********************************************************
 * Create a table named MYTABLENAME if it does not exist *
 *********************************************************/
static void create_table(MYSQL &mysql)
{
  if (mysql_query(&mysql,
      "CREATE TABLE"
      "  MYTABLENAME"
      "    (ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,"
      "     ATTR2 INT UNSIGNED NOT NULL)"
      "  ENGINE=NDB"))
    MYSQLERROR(mysql);
}

/*********************************
```

```
 * Drop a table named MYTABLENAME
 *********************************/
static void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql,
      "DROP TABLE"
      "  MYTABLENAME"))
    MYSQLERROR(mysql);
}

/*************************************************************************
 * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
 *************************************************************************/
static void do_insert(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 5; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
    myOperation->equal("ATTR1", i);
    myOperation->setValue("ATTR2", i);

    myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
    myOperation->equal("ATTR1", i+5);
    myOperation->setValue("ATTR2", i+5);

    if (myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/**************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 **************************************************************/
static void do_update(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 10; i+=2) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->updateTuple();
    myOperation->equal( "ATTR1", i );
    myOperation->setValue( "ATTR2", i+10);

    if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/**********************************************
 * Delete one tuple (the one with primary key 3) *
 **********************************************/
static void do_delete(Ndb &myNdb)
{
```

```
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTransaction= myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->deleteTuple();
  myOperation->equal( "ATTR1", 3 );

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);
}

/****************************
 * Read and print all tuples *
 ****************************/
static void do_read(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->readTuple(NdbOperation::LM_Read);
    myOperation->equal("ATTR1", i);

    NdbRecAttr *myRecAttr= myOperation->getValue("ATTR2", NULL);
    if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

    if(myTransaction->execute( NdbTransaction::Commit ) == -1)
      if (i == 3) {
  std::cout << "Detected that deleted tuple doesn't exist!" << std::endl;
      } else {
  APIERROR(myTransaction->getNdbError());
      }

    if (i != 3) {
      printf(" %2d    %2d\n", i, myRecAttr->u_32_value());
    }
    myNdb.closeTransaction(myTransaction);
  }
}
```

6.2. Using Synchronous Transactions and Multiple Clusters

This example demonstrates synchronous transactions and connecting to multiple clusters in a single NDB API application.

The source code for this program may be found in the MySQL 5.1 source tree, in the file `storage/ndb/ndbapi-examples/ndbapi_simple_dual/ndbapi_simple_dual.cpp`.

```
/*
 *  ndbapi_simple_dual.cpp: Using synchronous transactions in NDB API
 *
 *  Correct output from this program is:
```

```
 *
 *  ATTR1 ATTR2
 *    0    10
 *    1     1
 *    2    12
 *  Detected that deleted tuple doesn't exist!
 *    4    14
 *    5     5
 *    6    16
 *    7     7
 *    8    18
 *    9     9
 *  ATTR1 ATTR2
 *    0    10
 *    1     1
 *    2    12
 *  Detected that deleted tuple doesn't exist!
 *    4    14
 *    5     5
 *    6    16
 *    7     7
 *    8    18
 *    9     9
 *
 */

#include <mysql.h>
#include <NdbApi.hpp>
// Used for cout
#include <stdio.h>
#include <iostream>

static void run_application(MYSQL &, Ndb_cluster_connection &, const char* table, const char* db);

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

int main(int argc, char** argv)
{
  if (argc != 5)
  {
    std::cout << "Arguments are <socket mysqld1> <connect_string cluster 1> <socket mysqld2> <connect_s
    exit(-1);
  }
  // ndb_init must be called first
  ndb_init();
  {
    char * mysqld1_sock  = argv[1];
    const char *connectstring1 = argv[2];
    char * mysqld2_sock = argv[3];
    const char *connectstring2 = argv[4];

    // Object representing the cluster 1
    Ndb_cluster_connection cluster1_connection(connectstring1);
    MYSQL mysql1;
    // Object representing the cluster 2
    Ndb_cluster_connection cluster2_connection(connectstring2);
    MYSQL mysql2;

    // connect to mysql server and cluster 1 and run application
    // Connect to cluster 1  management server (ndb_mgmd)
    if (cluster1_connection.connect(4 /* retries               */,
            5 /* delay between retries */,
            1 /* verbose               */))
    {
      std::cout << "Cluster 1 management server was not ready within 30 secs.\n";
      exit(-1);
    }
    // Optionally connect and wait for the storage nodes (ndbd's)
    if (cluster1_connection.wait_until_ready(30,0) < 0)
    {
      std::cout << "Cluster 1 was not ready within 30 secs.\n";
```

```
      exit(-1);
    }
    // connect to mysql server in cluster 1
    if ( !mysql_init(&mysql1) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql1, "localhost", "root", "", "",
              0, mysqld1_sock, 0) )
      MYSQLERROR(mysql1);


    // connect to mysql server and cluster 2 and run application

    // Connect to cluster management server (ndb_mgmd)
    if (cluster2_connection.connect(4 /* retries               */,
            5 /* delay between retries */,
            1 /* verbose               */))
    {
      std::cout << "Cluster 2 management server was not ready within 30 secs.\n";
      exit(-1);
    }
    // Optionally connect and wait for the storage nodes (ndbd's)
    if (cluster2_connection.wait_until_ready(30,0) < 0)
    {
      std::cout << "Cluster 2 was not ready within 30 secs.\n";
      exit(-1);
    }
    // connect to mysql server in cluster 2
    if ( !mysql_init(&mysql2) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql2, "localhost", "root", "", "",
              0, mysqld2_sock, 0) )
      MYSQLERROR(mysql2);

    // run the application code
    run_application(mysql1, cluster1_connection, "MYTABLENAME1", "TEST_DB_1");
    run_application(mysql2, cluster2_connection, "MYTABLENAME2", "TEST_DB_2");
  }
  // Note: all connections must have been destroyed before calling ndb_end()
  ndb_end(0);

  return 0;
}

static void create_table(MYSQL &, const char* table);
static void drop_table(MYSQL &, const char* table);
static void do_insert(Ndb &, const char* table);
static void do_update(Ndb &, const char* table);
static void do_delete(Ndb &, const char* table);
static void do_read(Ndb &, const char* table);

static void run_application(MYSQL &mysql,
          Ndb_cluster_connection &cluster_connection,
          const char* table,
          const char* db)
{
  /****************************************
   * Connect to database via mysql-c          *
   ****************************************/
 char db_stmt[256];
 sprintf(db_stmt, "CREATE DATABASE %s\n", db);
  mysql_query(&mysql, db_stmt);
 sprintf(db_stmt, "USE %s", db);
  if (mysql_query(&mysql, db_stmt) != 0) MYSQLERROR(mysql);
  create_table(mysql, table);

  /****************************************
   * Connect to database via NdbApi           *
   ****************************************/
  // Object representing the database
  Ndb myNdb( &cluster_connection, db );
  if (myNdb.init()) APIERROR(myNdb.getNdbError());

  /*
   * Do different operations on database
   */
  do_insert(myNdb, table);
```

```
  do_update(myNdb, table);
  do_delete(myNdb, table);
  do_read(myNdb, table);
  /*
   * Drop the table
   */
  drop_table(mysql, table);
 sprintf(db_stmt, "DROP DATABASE %s\n", db);
  mysql_query(&mysql, db_stmt);
}

/********************************************************
 * Create a table named by table if it does not exist *
 ********************************************************/
static void create_table(MYSQL &mysql, const char* table)
{
  char create_stmt[256];

  sprintf(create_stmt, "CREATE TABLE %s \
             (ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,\
              ATTR2 INT UNSIGNED NOT NULL)\
             ENGINE=NDB", table);
  if (mysql_query(&mysql, create_stmt))
    MYSQLERROR(mysql);
}

/*****************************
 * Drop a table named by table
 *****************************/
static void drop_table(MYSQL &mysql, const char* table)
{
  char drop_stmt[256];

  sprintf(drop_stmt, "DROP TABLE IF EXISTS %s", table);
  if (mysql_query(&mysql, drop_stmt))
    MYSQLERROR(mysql);
}

/*************************************************************************
 * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
 *************************************************************************/
static void do_insert(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 5; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
    myOperation->equal("ATTR1", i);
    myOperation->setValue("ATTR2", i);

    myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
    myOperation->equal("ATTR1", i+5);
    myOperation->setValue("ATTR2", i+5);

    if (myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/**************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 **************************************************************/
static void do_update(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
```

```
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 10; i+=2) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->updateTuple();
    myOperation->equal( "ATTR1", i );
    myOperation->setValue( "ATTR2", i+10);

    if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/*********************************************
 * Delete one tuple (the one with primary key 3) *
 *********************************************/
static void do_delete(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTransaction= myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->deleteTuple();
  myOperation->equal( "ATTR1", 3 );

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);
}

/****************************
 * Read and print all tuples *
 ****************************/
static void do_read(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->readTuple(NdbOperation::LM_Read);
    myOperation->equal("ATTR1", i);

    NdbRecAttr *myRecAttr= myOperation->getValue("ATTR2", NULL);
    if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

    if(myTransaction->execute( NdbTransaction::Commit ) == -1)
      if (i == 3) {
  std::cout << "Detected that deleted tuple doesn't exist!" << std::endl;
      } else {
  APIERROR(myTransaction->getNdbError());
```

```
      }

    if (i != 3) {
      printf(" %2d    %2d\n", i, myRecAttr->u_32_value());
    }
    myNdb.closeTransaction(myTransaction);
  }
}
```

6.3. Handling Errors and Retrying Transactions

This program demonstrates handling errors and retrying failed transactions using the NDB API.

The source code for this example can be found in `storage/ndb/ndbapi-examples/ndbapi_retries/ndbapi_retries.cpp` in the MySQL 5.1 tree.

There are many ways to program using the NDB API. In this example, we perform two inserts in the same transaction using `NdbConnection::execute(NoCommit)`.

In NDB API applications, there are two types of failures to be taken into account:

1. **Transaction failures**: If non-permanent, these can be handled by re-executing the transaction.
2. **Application errors**: These are indicated by `APIERROR`; they must be handled by the application programmer.

```
/*
 *  ndbapi_retries.cpp: Error handling and retrying transactions
 */
#include <mysql.h>
#include <NdbApi.hpp>

// Used for cout
#include <iostream>

// Used for sleep (use your own version of sleep)
#include <unistd.h>
#define TIME_TO_SLEEP_BETWEEN_TRANSACTION_RETRIES 1

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }

//
//  APIERROR prints an NdbError object
//
#define APIERROR(error) \
  { std::cout << "API ERROR: " << error.code << " " << error.message \
              << std::endl \
              << "          " << "Status: " << error.status \
              << ", Classification: " << error.classification << std::endl\
              << "          " << "File: " << __FILE__ \
              << " (Line: " << __LINE__ << ")" << std::endl \
              ; \
  }

//
//  TRANSERROR prints all error info regarding an NdbTransaction
//
#define TRANSERROR(ndbTransaction) \
  { NdbError error = ndbTransaction->getNdbError(); \
    std::cout << "TRANS ERROR: " << error.code << " " << error.message \
              << std::endl \
              << "          " << "Status: " << error.status \
              << ", Classification: " << error.classification << std::endl \
              << "          " << "File: " << __FILE__ \
```

```
              << " (Line: " << __LINE__ << ")" << std::endl \
              ; \
    printTransactionError(ndbTransaction); \
  }

void printTransactionError(NdbTransaction *ndbTransaction) {
  const NdbOperation *ndbOp = NULL;
  int i=0;

  /****************************************************************
   * Print NdbError object of every operations in the transaction *
   ****************************************************************/
  while ((ndbOp = ndbTransaction->getNextCompletedOperation(ndbOp)) != NULL) {
    NdbError error = ndbOp->getNdbError();
    std::cout << "          OPERATION " << i+1 << ": "
        << error.code << " " << error.message << std::endl
        << "           Status: " << error.status
        << ", Classification: " << error.classification << std::endl;
    i++;
  }
}


//
//  Example insert
//  @param myNdb          Ndb object representing NDB Cluster
//  @param myTransaction  NdbTransaction used for transaction
//  @param myTable        Table to insert into
//  @param error          NdbError object returned in case of errors
//  @return -1 in case of failures, 0 otherwise
//
int insert(int transactionId, NdbTransaction* myTransaction,
     const NdbDictionary::Table *myTable) {
  NdbOperation   *myOperation;          // For other operations

  myOperation = myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) return -1;

  if (myOperation->insertTuple() ||
      myOperation->equal("ATTR1", transactionId) ||
      myOperation->setValue("ATTR2", transactionId)) {
    APIERROR(myOperation->getNdbError());
    exit(-1);
  }

  return myTransaction->execute(NdbTransaction::NoCommit);
}


//
//  Execute function which re-executes (tries 10 times) the transaction
//  if there are temporary errors (e.g. the NDB Cluster is overloaded).
//  @return -1 failure, 1 success
//
int executeInsertTransaction(int transactionId, Ndb* myNdb,
           const NdbDictionary::Table *myTable) {
  int result = 0;                       // No result yet
  int noOfRetriesLeft = 10;
  NdbTransaction   *myTransaction;         // For other transactions
  NdbError ndberror;

  while (noOfRetriesLeft > 0 && !result) {

    /*******************************
     * Start and execute transaction *
     *******************************/
    myTransaction = myNdb->startTransaction();
    if (myTransaction == NULL) {
      APIERROR(myNdb->getNdbError());
      ndberror = myNdb->getNdbError();
      result = -1;  // Failure
    } else if (insert(transactionId, myTransaction, myTable) ||
        insert(10000+transactionId, myTransaction, myTable) ||
        myTransaction->execute(NdbTransaction::Commit)) {
      TRANSERROR(myTransaction);
      ndberror = myTransaction->getNdbError();
      result = -1;  // Failure
    } else {
      result = 1;   // Success
    }
```

```
    /**********************************
     * If failure, then analyze error *
     **********************************/
    if (result == -1) {
      switch (ndberror.status) {
      case NdbError::Success:
  break;
      case NdbError::TemporaryError:
  std::cout << "Retrying transaction..." << std::endl;
  sleep(TIME_TO_SLEEP_BETWEEN_TRANSACTION_RETRIES);
  --noOfRetriesLeft;
  result = 0;   // No completed transaction yet
  break;

      case NdbError::UnknownResult:
      case NdbError::PermanentError:
  std::cout << "No retry of transaction..." << std::endl;
  result = -1;  // Permanent failure
  break;
      }
    }

    /********************
     * Close transaction *
     ********************/
    if (myTransaction != NULL) {
      myNdb->closeTransaction(myTransaction);
    }
  }

  if (result != 1) exit(-1);
  return result;
}

/*********************************************************
 * Create a table named MYTABLENAME if it does not exist *
 *********************************************************/
static void create_table(MYSQL &mysql)
{
  if (mysql_query(&mysql,
      "CREATE TABLE"
      "  MYTABLENAME"
      "    (ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,"
      "     ATTR2 INT UNSIGNED NOT NULL)"
      "  ENGINE=NDB"))
    MYSQLERROR(mysql);
}

/*********************************
 * Drop a table named MYTABLENAME
 *********************************/
static void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql,
      "DROP TABLE"
      "  MYTABLENAME"))
    MYSQLERROR(mysql);
}

int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char * mysqld_sock  = argv[1];
  const char *connectstring = argv[2];
  ndb_init();

  Ndb_cluster_connection *cluster_connection=
    new Ndb_cluster_connection(connectstring); // Object representing the cluster

  int r= cluster_connection->connect(5 /* retries               */,
              3 /* delay between retries */,
              1 /* verbose               */);
  if (r > 0)
  {
    std::cout
```

```
      << "Cluster connect failed, possibly resolved with more retries.\n";
    exit(-1);
  }
  else if (r < 0)
  {
    std::cout
      << "Cluster connect failed.\n";
    exit(-1);
  }

  if (cluster_connection->wait_until_ready(30,30))
  {
    std::cout << "Cluster was not ready within 30 secs." << std::endl;
    exit(-1);
  }
  // connect to mysql server
  MYSQL mysql;
  if ( !mysql_init(&mysql) ) {
    std::cout << "mysql_init failed\n";
    exit(-1);
  }
  if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
         0, mysqld_sock, 0) )
    MYSQLERROR(mysql);

  /*****************************************
   * Connect to database via mysql-c          *
   *****************************************/
  mysql_query(&mysql, "CREATE DATABASE TEST_DB_1");
  if (mysql_query(&mysql, "USE TEST_DB_1") != 0) MYSQLERROR(mysql);
  create_table(mysql);

  Ndb* myNdb= new Ndb( cluster_connection,
           "TEST_DB_1" );  // Object representing the database

  if (myNdb->init() == -1) {
    APIERROR(myNdb->getNdbError());
    exit(-1);
  }

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");
  if (myTable == NULL)
  {
    APIERROR(myDict->getNdbError());
    return -1;
  }
  /*********************************
   * Execute some insert transactions *
   *********************************/
  for (int i = 10000; i < 20000; i++) {
    executeInsertTransaction(i, myNdb, myTable);
  }

  delete myNdb;
  delete cluster_connection;

  drop_table(mysql);

  ndb_end(0);
  return 0;
}
```

6.4. Basic Scanning Example

This example illustrates how to use the NDB scanning API. It shows how to perform a scan, how to scan for an update, and how to scan for a delete, making use of the `NdbScanFilter` and `NdbScanOperation` classes.

(See Section 3.8, “The `NdbScanFilter` Class”, and Section 3.6.4, “The `NdbScanOperation` Class”.)

The source code for this example may found in MySQL 5.1 tree, in the file `storage/ndb/ndbapi-examples/ndbapi_scan/ndbapi_scan.cpp`.

This example makes use of the following classes and methods:

- `Ndb_cluster_connection`:
  - `connect()`
  - `wait_until_ready()`

  See Section 3.2, “The `Ndb_cluster_connection` Class”.
- `Ndb`:
  - `init()`
  - `getDictionary()`
  - `startTransaction()`
  - `closeTransaction()`

  See Section 3.1, “The `Ndb` Class”.
- `NdbTransaction`:
  - `getNdbScanOperation()`
  - `execute()`

  See Section 3.9, “The `NdbTransaction` Class”.
- `NdbOperation`:
  - `insertTuple()`
  - `equal()`
  - `setValue()`

  See Section 3.6, “The `NdbOperation` Class”.
- `NdbScanOperation`:
  - `getValue()`
  - `readTuples()`
  - `nextResult()`
  - `deleteCurrentTuple()`
  - `updateCurrentTuple()`

  See Section 3.6.4, “The `NdbScanOperation` Class”.
- `NdbDictionary`:
  - `Dictionary::getTable()`

    See Section 3.4.1, “The `Dictionary` Class”.

  - `Table::getColumn()`

    See Section 3.4.3.7, “The `Table` Class”.

  - `Column::getLength()`

    See Section 3.4.2, “The `Column` Class”.

- `NdbScanFilter`:

  - `begin()`

  - `eq()`

  - `end()`

  See Section 3.8, “The `NdbScanFilter` Class”.

```
#include <mysql.h>
#include <mysqld_error.h>
#include <NdbApi.hpp>
// Used for cout
#include <iostream>
#include <stdio.h>

/**
 * Helper sleep function
 */
static void
milliSleep(int milliseconds){
  struct timeval sleeptime;
  sleeptime.tv_sec = milliseconds / 1000;
  sleeptime.tv_usec = (milliseconds - (sleeptime.tv_sec * 1000)) * 1000000;
  select(0, 0, 0, 0, &sleeptime);
}


/**
 * Helper sleep function
 */
#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

struct Car
{
  /**
   * Note memset, so that entire char-fields are cleared
   *   as all 20 bytes are significant (as type is char)
   */
  Car() { memset(this, 0, sizeof(* this)); }

  unsigned int reg_no;
  char brand[20];
  char color[20];
};

/**
 * Function to drop table
 */
void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql, "DROP TABLE GARAGE"))
    MYSQLERROR(mysql);
}
```

```
/**
 * Function to create table
 */
void create_table(MYSQL &mysql)
{
  while (mysql_query(&mysql,
      "CREATE TABLE"
      "  GARAGE"
      "    (REG_NO INT UNSIGNED NOT NULL,"
      "     BRAND CHAR(20) NOT NULL,"
      "     COLOR CHAR(20) NOT NULL,"
      "     PRIMARY KEY USING HASH (REG_NO))"
      "  ENGINE=NDB"))
  {
    if (mysql_errno(&mysql) != ER_TABLE_EXISTS_ERROR)
      MYSQLERROR(mysql);
    std::cout << "MySQL Cluster already has example table: GARAGE. "
        << "Dropping it..." << std::endl;
    /******************
     * Recreate table *
     ******************/
    drop_table(mysql);
    create_table(mysql);
  }
}

int populate(Ndb * myNdb)
{
  int i;
  Car cars[15];

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("GARAGE");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Five blue mercedes
   */
  for (i = 0; i < 5; i++)
  {
    cars[i].reg_no = i;
    sprintf(cars[i].brand, "Mercedes");
    sprintf(cars[i].color, "Blue");
  }

  /**
   * Five black bmw
   */
  for (i = 5; i < 10; i++)
  {
    cars[i].reg_no = i;
    sprintf(cars[i].brand, "BMW");
    sprintf(cars[i].color, "Black");
  }

  /**
   * Five pink toyotas
   */
  for (i = 10; i < 15; i++)
  {
    cars[i].reg_no = i;
    sprintf(cars[i].brand, "Toyota");
    sprintf(cars[i].color, "Pink");
  }

  NdbTransaction* myTrans = myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  for (i = 0; i < 15; i++)
  {
    NdbOperation* myNdbOperation = myTrans->getNdbOperation(myTable);
    if (myNdbOperation == NULL)
      APIERROR(myTrans->getNdbError());
    myNdbOperation->insertTuple();
    myNdbOperation->equal("REG_NO", cars[i].reg_no);
```

```
    myNdbOperation->setValue("BRAND", cars[i].brand);
    myNdbOperation->setValue("COLOR", cars[i].color);
  }

  int check = myTrans->execute(NdbTransaction::Commit);

  myTrans->close();

  return check != -1;
}

int scan_delete(Ndb* myNdb,
    int column,
    const char * color)

{

  // Scan all records exclusive and delete
  // them one by one
  int                  retryAttempt = 0;
  const int            retryMax = 10;
  int deletedRows = 0;
  int check;
  NdbError             err;
  NdbTransaction  *myTrans;
  NdbScanOperation  *myScanOp;

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("GARAGE");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Loop as long as :
   *  retryMax not reached
   *  failed operations due to TEMPORARY erros
   *
   * Exit loop;
   *  retyrMax reached
   *  Permanent error (return -1)
   */
  while (true)
  {
    if (retryAttempt >= retryMax)
    {
      std::cout << "ERROR: has retried this operation " << retryAttempt
    << " times, failing!" << std::endl;
      return -1;
    }

    myTrans = myNdb->startTransaction();
    if (myTrans == NULL)
    {
      const NdbError err = myNdb->getNdbError();

      if (err.status == NdbError::TemporaryError)
      {
  milliSleep(50);
  retryAttempt++;
  continue;
      }
      std::cout <<  err.message << std::endl;
      return -1;
    }

   /**
    * Get a scan operation.
    */
    myScanOp = myTrans->getNdbScanOperation(myTable);
    if (myScanOp == NULL)
    {
      std::cout << myTrans->getNdbError().message << std::endl;
      myNdb->closeTransaction(myTrans);
      return -1;
    }

    /**
     * Define a result set for the scan.
     */
```

```
  if(myScanOp->readTuples(NdbOperation::LM_Exclusive) != 0)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Use NdbScanFilter to define a search critera
   */
  NdbScanFilter filter(myScanOp) ;
  if(filter.begin(NdbScanFilter::AND) < 0  ||
     filter.cmp(NdbScanFilter::COND_EQ, column, color) < 0 ||
     filter.end() < 0)
  {
    std::cout <<  myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Start scan    (NoCommit since we are only reading at this stage);
   */
  if(myTrans->execute(NdbTransaction::NoCommit) != 0){
    err = myTrans->getNdbError();
    if(err.status == NdbError::TemporaryError){
std::cout << myTrans->getNdbError().message << std::endl;
myNdb->closeTransaction(myTrans);
milliSleep(50);
continue;
    }
    std::cout << err.code << std::endl;
    std::cout << myTrans->getNdbError().code << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }


 /**
  * start of loop: nextResult(true) means that "parallelism" number of
  * rows are fetched from NDB and cached in NDBAPI
  */
  while((check = myScanOp->nextResult(true)) == 0){
    do
    {
if (myScanOp->deleteCurrentTuple() != 0)
{
  std::cout << myTrans->getNdbError().message << std::endl;
  myNdb->closeTransaction(myTrans);
  return -1;
}
deletedRows++;

/**
 * nextResult(false) means that the records
 * cached in the NDBAPI are modified before
 * fetching more rows from NDB.
 */
    } while((check = myScanOp->nextResult(false)) == 0);

    /**
     * Commit when all cached tuple have been marked for deletion
     */
    if(check != -1)
    {
check = myTrans->execute(NdbTransaction::Commit);
    }

    if(check == -1)
    {
/**
 * Create a new transaction, while keeping scan open
 */
check = myTrans->restart();
    }

    /**
     * Check for errors
     */
    err = myTrans->getNdbError();
```

```
      if(check == -1)
      {
  if(err.status == NdbError::TemporaryError)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    milliSleep(50);
    continue;
  }
      }
      /**
       * End of loop
       */
    }
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return 0;
  }

  if(myTrans!=0)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
  }
  return -1;
}


int scan_update(Ndb* myNdb,
    int update_column,
    const char * before_color,
    const char * after_color)

{

  // Scan all records exclusive and update
  // them one by one
  int                  retryAttempt = 0;
  const int            retryMax = 10;
  int updatedRows = 0;
  int check;
  NdbError             err;
  NdbTransaction  *myTrans;
  NdbScanOperation  *myScanOp;

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("GARAGE");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Loop as long as :
   *  retryMax not reached
   *  failed operations due to TEMPORARY erros
   *
   * Exit loop;
   *  retyrMax reached
   *  Permanent error (return -1)
   */
  while (true)
  {

    if (retryAttempt >= retryMax)
    {
      std::cout << "ERROR: has retried this operation " << retryAttempt
    << " times, failing!" << std::endl;
      return -1;
    }

    myTrans = myNdb->startTransaction();
    if (myTrans == NULL)
    {
      const NdbError err = myNdb->getNdbError();

      if (err.status == NdbError::TemporaryError)
      {
  milliSleep(50);
  retryAttempt++;
  continue;
```

```
    }
    std::cout <<  err.message << std::endl;
    return -1;
  }

 /**
  * Get a scan operation.
  */
  myScanOp = myTrans->getNdbScanOperation(myTable);
  if (myScanOp == NULL)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Define a result set for the scan.
   */
  if( myScanOp->readTuples(NdbOperation::LM_Exclusive) )
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Use NdbScanFilter to define a search critera
   */
  NdbScanFilter filter(myScanOp) ;
  if(filter.begin(NdbScanFilter::AND) < 0  ||
     filter.cmp(NdbScanFilter::COND_EQ, update_column, before_color) <0||
     filter.end() <0)
  {
    std::cout <<  myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Start scan    (NoCommit since we are only reading at this stage);
   */
  if(myTrans->execute(NdbTransaction::NoCommit) != 0)
  {
    err = myTrans->getNdbError();
    if(err.status == NdbError::TemporaryError){
std::cout << myTrans->getNdbError().message << std::endl;
myNdb->closeTransaction(myTrans);
milliSleep(50);
continue;
    }
    std::cout << myTrans->getNdbError().code << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * start of loop: nextResult(true) means that "parallelism" number of
   * rows are fetched from NDB and cached in NDBAPI
   */
  while((check = myScanOp->nextResult(true)) == 0){
    do {
/**
 * Get update operation
 */
NdbOperation * myUpdateOp = myScanOp->updateCurrentTuple();
if (myUpdateOp == 0)
{
  std::cout << myTrans->getNdbError().message << std::endl;
  myNdb->closeTransaction(myTrans);
  return -1;
}
updatedRows++;

/**
 * do the update
 */
myUpdateOp->setValue(update_column, after_color);
/**
 * nextResult(false) means that the records
```

```
   * cached in the NDBAPI are modified before
   * fetching more rows from NDB.
   */
      } while((check = myScanOp->nextResult(false)) == 0);

      /**
       * NoCommit when all cached tuple have been updated
       */
      if(check != -1)
      {
  check = myTrans->execute(NdbTransaction::NoCommit);
      }

      /**
       * Check for errors
       */
      err = myTrans->getNdbError();
      if(check == -1)
      {
  if(err.status == NdbError::TemporaryError){
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    milliSleep(50);
    continue;
  }
      }
      /**
       * End of loop
       */
    }

    /**
     * Commit all prepared operations
     */
    if(myTrans->execute(NdbTransaction::Commit) == -1)
    {
      if(err.status == NdbError::TemporaryError){
  std::cout << myTrans->getNdbError().message << std::endl;
  myNdb->closeTransaction(myTrans);
  milliSleep(50);
  continue;
      }
    }

    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return 0;
  }


  if(myTrans!=0)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
  }
  return -1;
}



int scan_print(Ndb * myNdb)
{
// Scan all records exclusive and update
  // them one by one
  int                  retryAttempt = 0;
  const int            retryMax = 10;
  int fetchedRows = 0;
  int check;
  NdbError             err;
  NdbTransaction  *myTrans;
  NdbScanOperation  *myScanOp;
  /* Result of reading attribute value, three columns:
     REG_NO, BRAND, and COLOR
   */
  NdbRecAttr *      myRecAttr[3];

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("GARAGE");

  if (myTable == NULL)
```

```
  APIERROR(myDict->getNdbError());

/**
 * Loop as long as :
 *  retryMax not reached
 *  failed operations due to TEMPORARY erros
 *
 * Exit loop;
 *  retyrMax reached
 *  Permanent error (return -1)
 */
while (true)
{

  if (retryAttempt >= retryMax)
  {
    std::cout << "ERROR: has retried this operation " << retryAttempt
  << " times, failing!" << std::endl;
    return -1;
  }

  myTrans = myNdb->startTransaction();
  if (myTrans == NULL)
  {
    const NdbError err = myNdb->getNdbError();

    if (err.status == NdbError::TemporaryError)
    {
milliSleep(50);
retryAttempt++;
continue;
    }
   std::cout << err.message << std::endl;
    return -1;
  }
  /*
   * Define a scan operation.
   * NDBAPI.
   */
  myScanOp = myTrans->getNdbScanOperation(myTable);
  if (myScanOp == NULL)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Read without locks, without being placed in lock queue
   */
  if( myScanOp->readTuples(NdbOperation::LM_CommittedRead) == -1)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Define storage for fetched attributes.
   * E.g., the resulting attributes of executing
   * myOp->getValue("REG_NO") is placed in myRecAttr[0].
   * No data exists in myRecAttr until transaction has commited!
   */
  myRecAttr[0] = myScanOp->getValue("REG_NO");
  myRecAttr[1] = myScanOp->getValue("BRAND");
  myRecAttr[2] = myScanOp->getValue("COLOR");
  if(myRecAttr[0] ==NULL || myRecAttr[1] == NULL || myRecAttr[2]==NULL)
  {
std::cout << myTrans->getNdbError().message << std::endl;
myNdb->closeTransaction(myTrans);
return -1;
  }
  /**
   * Start scan   (NoCommit since we are only reading at this stage);
   */
  if(myTrans->execute(NdbTransaction::NoCommit) != 0){
    err = myTrans->getNdbError();
    if(err.status == NdbError::TemporaryError){
std::cout << myTrans->getNdbError().message << std::endl;
myNdb->closeTransaction(myTrans);
```

```
  milliSleep(50);
  continue;
      }
      std::cout << err.code << std::endl;
      std::cout << myTrans->getNdbError().code << std::endl;
      myNdb->closeTransaction(myTrans);
      return -1;
    }

    /**
     * start of loop: nextResult(true) means that "parallelism" number of
     * rows are fetched from NDB and cached in NDBAPI
     */
    while((check = myScanOp->nextResult(true)) == 0){
      do {

  fetchedRows++;
  /**
   * print  REG_NO unsigned int
   */
  std::cout << myRecAttr[0]->u_32_value() << "\t";

  /**
   * print  BRAND character string
   */
  std::cout << myRecAttr[1]->aRef() << "\t";

  /**
   * print  COLOR character string
   */
  std::cout << myRecAttr[2]->aRef() << std::endl;

  /**
   * nextResult(false) means that the records
   * cached in the NDBAPI are modified before
   * fetching more rows from NDB.
   */
      } while((check = myScanOp->nextResult(false)) == 0);

    }
    myNdb->closeTransaction(myTrans);
    return 1;
  }
  return -1;

}


int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char * mysqld_sock  = argv[1];
  const char *connectstring = argv[2];
  ndb_init();
  MYSQL mysql;

  /*****************************************************************
   * Connect to mysql server and create table                      *
   *****************************************************************/
  {
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
          0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    mysql_query(&mysql, "CREATE DATABASE TEST_DB");
    if (mysql_query(&mysql, "USE TEST_DB") != 0) MYSQLERROR(mysql);

    create_table(mysql);
  }

  /*****************************************************************
   * Connect to ndb cluster                                        *
```

```
 *************************************************************/

Ndb_cluster_connection cluster_connection(connectstring);
if (cluster_connection.connect(4, 5, 1))
{
  std::cout << "Unable to connect to cluster within 30 secs." << std::endl;
  exit(-1);
}
// Optionally connect and wait for the storage nodes (ndbd's)
if (cluster_connection.wait_until_ready(30,0) < 0)
{
  std::cout << "Cluster was not ready within 30 secs.\n";
  exit(-1);
}

Ndb myNdb(&cluster_connection,"TEST_DB");
if (myNdb.init(1024) == -1) {      // Set max 1024  parallel transactions
  APIERROR(myNdb.getNdbError());
  exit(-1);
}

/****************************************
 * Check table definition                *
 ****************************************/
int column_color;
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *t= myDict->getTable("GARAGE");

  Car car;
  if (t->getColumn("COLOR")->getLength() != sizeof(car.color) ||
t->getColumn("BRAND")->getLength() != sizeof(car.brand))
  {
    std::cout << "Wrong table definition" << std::endl;
    exit(-1);
  }
  column_color= t->getColumn("COLOR")->getColumnNo();
}

if(populate(&myNdb) > 0)
  std::cout << "populate: Success!" << std::endl;

if(scan_print(&myNdb) > 0)
  std::cout << "scan_print: Success!" << std::endl  << std::endl;

std::cout << "Going to delete all pink cars!" << std::endl;

{
  /**
   * Note! color needs to be of exact the same size as column defined
   */
  Car tmp;
  sprintf(tmp.color, "Pink");
  if(scan_delete(&myNdb, column_color, tmp.color) > 0)
    std::cout << "scan_delete: Success!" << std::endl  << std::endl;
}

if(scan_print(&myNdb) > 0)
  std::cout << "scan_print: Success!" << std::endl  << std::endl;

{
  /**
   * Note! color1 & 2 need to be of exact the same size as column defined
   */
  Car tmp1, tmp2;
  sprintf(tmp1.color, "Blue");
  sprintf(tmp2.color, "Black");
  std::cout << "Going to update all " << tmp1.color
      << " cars to " << tmp2.color << " cars!" << std::endl;
  if(scan_update(&myNdb, column_color, tmp1.color, tmp2.color) > 0)
    std::cout << "scan_update: Success!" << std::endl  << std::endl;
}
if(scan_print(&myNdb) > 0)
  std::cout << "scan_print: Success!" << std::endl  << std::endl;

/**
 * Drop table
 */
drop_table(mysql);
```

```
  return 0;
}
```

6.5. Using Secondary Indexes in Scans

This program illustrates how to use secondary indexes in the NDB API.

The source code for this example may be found in the MySQL 5.1 source tree, in `storage/ndb/ndbapi-examples/ndbapi_simple_index/ndbapi_simple_index.cpp`.

The correct output from this program is shown here:

```
ATTR1 ATTR2
0      10
1       1
2      12
Detected that deleted tuple doesn't exist!
4      14
5       5
6      16
7       7
8      18
9       9
```

```
#include <mysql.h>
#include <NdbApi.hpp>

// Used for cout
#include <stdio.h>
#include <iostream>

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

int main(int argc, char** argv)
{
  if (argc != 3)
    {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char * mysqld_sock  = argv[1];
  const char *connectstring = argv[2];
  ndb_init();
  MYSQL mysql;

  /****************************************************************
   * Connect to mysql server and create table                     *
   ****************************************************************/
  {
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
          0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    mysql_query(&mysql, "CREATE DATABASE TEST_DB_1");
    if (mysql_query(&mysql, "USE TEST_DB_1") != 0) MYSQLERROR(mysql);

    if (mysql_query(&mysql,
        "CREATE TABLE"
        "  MYTABLENAME"
        "    (ATTR1 INT UNSIGNED,"
        "     ATTR2 INT UNSIGNED NOT NULL,"
```

```
      "     PRIMARY KEY USING HASH (ATTR1),"
      "     UNIQUE MYINDEXNAME USING HASH (ATTR2))"
      "  ENGINE=NDB"))
    MYSQLERROR(mysql);
}

/*************************************************************
 * Connect to ndb cluster                                    *
 *************************************************************/

Ndb_cluster_connection *cluster_connection=
  new Ndb_cluster_connection(connectstring); // Object representing the cluster

if (cluster_connection->connect(5,3,1))
{
  std::cout << "Connect to cluster management server failed.\n";
  exit(-1);
}

if (cluster_connection->wait_until_ready(30,30))
{
  std::cout << "Cluster was not ready within 30 secs.\n";
  exit(-1);
}

Ndb* myNdb = new Ndb( cluster_connection,
    "TEST_DB_1" );  // Object representing the database
if (myNdb->init() == -1) {
  APIERROR(myNdb->getNdbError());
  exit(-1);
}

const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");
if (myTable == NULL)
  APIERROR(myDict->getNdbError());
const NdbDictionary::Index *myIndex= myDict->getIndex("MYINDEXNAME$unique","MYTABLENAME");
if (myIndex == NULL)
  APIERROR(myDict->getNdbError());

/*************************************************************************
 * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
 *************************************************************************/
for (int i = 0; i < 5; i++) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->insertTuple();
  myOperation->equal("ATTR1", i);
  myOperation->setValue("ATTR2", i);

  myOperation = myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->insertTuple();
  myOperation->equal("ATTR1", i+5);
  myOperation->setValue("ATTR2", i+5);

  if (myTransaction->execute( NdbTransaction::Commit ) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/****************************************
 * Read and print all tuples using index *
 ****************************************/
std::cout << "ATTR1 ATTR2" << std::endl;

for (int i = 0; i < 10; i++) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbIndexOperation *myIndexOperation=
    myTransaction->getNdbIndexOperation(myIndex);
  if (myIndexOperation == NULL) APIERROR(myTransaction->getNdbError());
```

```
  myIndexOperation->readTuple(NdbOperation::LM_Read);
  myIndexOperation->equal("ATTR2", i);

  NdbRecAttr *myRecAttr= myIndexOperation->getValue("ATTR1", NULL);
  if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

  if(myTransaction->execute( NdbTransaction::Commit ) != -1)
    printf(" %2d    %2d\n", myRecAttr->u_32_value(), i);

  myNdb->closeTransaction(myTransaction);
}

/*************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 *************************************************************/
for (int i = 0; i < 10; i+=2) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbIndexOperation *myIndexOperation=
    myTransaction->getNdbIndexOperation(myIndex);
  if (myIndexOperation == NULL) APIERROR(myTransaction->getNdbError());

  myIndexOperation->updateTuple();
  myIndexOperation->equal( "ATTR2", i );
  myIndexOperation->setValue( "ATTR2", i+10);

  if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/*********************************************
 * Delete one tuple (the one with primary key 3) *
 *********************************************/
{
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbIndexOperation *myIndexOperation=
    myTransaction->getNdbIndexOperation(myIndex);
  if (myIndexOperation == NULL) APIERROR(myTransaction->getNdbError());

  myIndexOperation->deleteTuple();
  myIndexOperation->equal( "ATTR2", 3 );

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/****************************
 * Read and print all tuples *
 ****************************/
{
  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb->startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->readTuple(NdbOperation::LM_Read);
    myOperation->equal("ATTR1", i);

    NdbRecAttr *myRecAttr= myOperation->getValue("ATTR2", NULL);
    if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

    if(myTransaction->execute( NdbTransaction::Commit ) == -1)
if (i == 3) {
  std::cout << "Detected that deleted tuple doesn't exist!\n";
} else {
  APIERROR(myTransaction->getNdbError());
}

    if (i != 3) {
```

```
  printf(" %2d    %2d\n", i, myRecAttr->u_32_value());
      }
      myNdb->closeTransaction(myTransaction);
    }
  }

  /**************
   * Drop table *
   **************/
  if (mysql_query(&mysql, "DROP TABLE MYTABLENAME"))
    MYSQLERROR(mysql);

  delete myNdb;
  delete cluster_connection;

  ndb_end(0);
  return 0;
}
```

6.6. NDB API Event Handling Example

This example demonstrates NDB API event handling.

The source code for this program may be found in the MySQL 5.1 source tree, in the file `storage/ndb/ndbapi-examples/ndbapi_event/ndbapi_event.cpp`.

```
/*
 *  ndbapi_event.cpp: Illustrates event handling in the NDB API.
 */
#include <NdbApi.hpp>

// Used for cout
#include <stdio.h>
#include <iostream>
#include <unistd.h>
#ifdef VM_TRACE
#include <my_global.h>
#endif
#ifndef assert
#include <assert.h>
#endif


/**
 * Assume that there is a table which is being updated by
 * another process (e.g. flexBench -l 0 -stdtables).
 * We want to monitor what happens with column values.
 *
 * Or using the mysql client:
 *
 * shell> mysql -u root
 * mysql> create database TEST_DB;
 * mysql> use TEST_DB;
 * mysql> create table t0
 *        (c0 int, c1 int, c2 char(4), c3 char(4), c4 text,
 *        primary key(c0, c2)) engine ndb charset latin1;
 *
 * In another window start ndbapi_event, wait until properly started

   insert into t0 values (1, 2, 'a', 'b', null);
   insert into t0 values (3, 4, 'c', 'd', null);
   update t0 set c3 = 'e' where c0 = 1 and c2 = 'a'; -- use pk
   update t0 set c3 = 'f'; -- use scan
   update t0 set c3 = 'F'; -- use scan update to 'same'
   update t0 set c2 = 'g' where c0 = 1; -- update pk part
   update t0 set c2 = 'G' where c0 = 1; -- update pk part to 'same'
   update t0 set c0 = 5, c2 = 'H' where c0 = 3; -- update full PK
   delete from t0;

   insert ...; update ...; -- see events w/ same pk merged (if -m option)
   delete ...; insert ...; -- there are 5 combinations ID IU DI UD UU
   update ...; update ...;

   -- text requires -m flag
   set @a = repeat('a',256); -- inline size
   set @b = repeat('b',2000); -- part size
```

```
   set @c = repeat('c',2000*30); -- 30 parts

   -- update the text field using combinations of @a, @b, @c ...

 * you should see the data popping up in the example window
 *
 */

#define APIERROR(error) \
  { std::cout << "Error in " << __FILE__ << ", line:" << __LINE__ << ", code:" \
              << error.code << ", msg: " << error.message << "." << std::endl; \
    exit(-1); }

int myCreateEvent(Ndb* myNdb,
                  const char *eventName,
                  const char *eventTableName,
                  const char **eventColumnName,
                  const int noEventColumnName,
                  bool merge_events);

int main(int argc, char** argv)
{
  if (argc < 3)
  {
    std::cout << "Arguments are <connect_string cluster> <timeout> [m(merge events)|d(debug)].\n";
    exit(-1);
  }
  const char *connectstring = argv[1];
  int timeout = atoi(argv[2]);
  ndb_init();
  bool merge_events = argc > 3 && strchr(argv[3], 'm') != 0;
#ifdef VM_TRACE
  bool dbug = argc > 3 && strchr(argv[3], 'd') != 0;
  if (dbug) DBUG_PUSH("d:t:");
  if (dbug) putenv("API_SIGNAL_LOG=-");
#endif

  Ndb_cluster_connection *cluster_connection=
    new Ndb_cluster_connection(connectstring); // Object representing the cluster

  int r= cluster_connection->connect(5 /* retries               */,
                                     3 /* delay between retries */,
                                     1 /* verbose               */);
  if (r > 0)
  {
    std::cout
      << "Cluster connect failed, possibly resolved with more retries.\n";
    exit(-1);
  }
  else if (r < 0)
  {
    std::cout
      << "Cluster connect failed.\n";
    exit(-1);
  }

  if (cluster_connection->wait_until_ready(30,30))
  {
    std::cout << "Cluster was not ready within 30 secs." << std::endl;
    exit(-1);
  }

  Ndb* myNdb= new Ndb(cluster_connection,
          "TEST_DB");  // Object representing the database

  if (myNdb->init() == -1) APIERROR(myNdb->getNdbError());

  const char *eventName= "CHNG_IN_t0";
  const char *eventTableName= "t0";
  const int noEventColumnName= 5;
  const char *eventColumnName[noEventColumnName]=
    {"c0",
     "c1",
     "c2",
     "c3",
     "c4"
    };

  // Create events
  myCreateEvent(myNdb,
```

```
  eventName,
  eventTableName,
  eventColumnName,
  noEventColumnName,
              merge_events);

// Normal values and blobs are unfortunately handled differently..
typedef union { NdbRecAttr* ra; NdbBlob* bh; } RA_BH;

int i, j, k, l;
j = 0;
while (j < timeout) {

  // Start "transaction" for handling events
  NdbEventOperation* op;
  printf("create EventOperation\n");
  if ((op = myNdb->createEventOperation(eventName)) == NULL)
    APIERROR(myNdb->getNdbError());
  op->mergeEvents(merge_events);

  printf("get values\n");
  RA_BH recAttr[noEventColumnName];
  RA_BH recAttrPre[noEventColumnName];
  // primary keys should always be a part of the result
  for (i = 0; i < noEventColumnName; i++) {
    if (i < 4) {
      recAttr[i].ra    = op->getValue(eventColumnName[i]);
      recAttrPre[i].ra = op->getPreValue(eventColumnName[i]);
    } else if (merge_events) {
      recAttr[i].bh    = op->getBlobHandle(eventColumnName[i]);
      recAttrPre[i].bh = op->getPreBlobHandle(eventColumnName[i]);
    }
  }

  // set up the callbacks
  printf("execute\n");
  // This starts changes to "start flowing"
  if (op->execute())
    APIERROR(op->getNdbError());

  NdbEventOperation* the_op = op;

  i= 0;
  while (i < timeout) {
    // printf("now waiting for event...\n");
    int r = myNdb->pollEvents(1000); // wait for event or 1000 ms
    if (r > 0) {
// printf("got data! %d\n", r);
while ((op= myNdb->nextEvent())) {
        assert(the_op == op);
  i++;
  switch (op->getEventType()) {
  case NdbDictionary::Event::TE_INSERT:
    printf("%u INSERT", i);
    break;
  case NdbDictionary::Event::TE_DELETE:
    printf("%u DELETE", i);
    break;
  case NdbDictionary::Event::TE_UPDATE:
    printf("%u UPDATE", i);
    break;
  default:
    abort(); // should not happen
  }
        printf(" gci=%d\n", (int)op->getGCI());
        for (k = 0; k <= 1; k++) {
          printf(k == 0 ? "post: " : "pre : ");
          for (l = 0; l < noEventColumnName; l++) {
            if (l < 4) {
              NdbRecAttr* ra = k == 0 ? recAttr[l].ra : recAttrPre[l].ra;
              if (ra->isNULL() >= 0) { // we have a value
                if (ra->isNULL() == 0) { // we have a non-null value
                  if (l < 2)
                    printf("%-5u", ra->u_32_value());
                  else
                    printf("%-5.4s", ra->aRef());
                } else
                  printf("%-5s", "NULL");
              } else
                printf("%-5s", "-"); // no value
```

```
              } else if (merge_events) {
                int isNull;
                NdbBlob* bh = k == 0 ? recAttr[l].bh : recAttrPre[l].bh;
                bh->getDefined(isNull);
                if (isNull >= 0) { // we have a value
                  if (! isNull) { // we have a non-null value
                    Uint64 length = 0;
                    bh->getLength(length);
                    // read into buffer
                    unsigned char* buf = new unsigned char [length];
                    memset(buf, 'X', length);
                    Uint32 n = length;
                    bh->readData(buf, n); // n is in/out
                    assert(n == length);
                    // pretty-print
                    bool first = true;
                    Uint32 i = 0;
                    while (i < n) {
                      unsigned char c = buf[i++];
                      Uint32 m = 1;
                      while (i < n && buf[i] == c)
                        i++, m++;
                      if (! first)
                        printf("+");
                      printf("%u%c", m, c);
                      first = false;
                    }
                    printf("[%u]", n);
                    delete [] buf;
                  } else
                    printf("%-5s", "NULL");
                } else
                  printf("%-5s", "-"); // no value
              }
            }
            printf("\n");
          }
  }
      } else
  printf("timed out (%i)\n", timeout);
    }
    // don't want to listen to events anymore
    if (myNdb->dropEventOperation(the_op)) APIERROR(myNdb->getNdbError());
    the_op = 0;

    j++;
  }

  {
    NdbDictionary::Dictionary *myDict = myNdb->getDictionary();
    if (!myDict) APIERROR(myNdb->getNdbError());
    // remove event from database
    if (myDict->dropEvent(eventName)) APIERROR(myDict->getNdbError());
  }

  delete myNdb;
  delete cluster_connection;
  ndb_end(0);
  return 0;
}

int myCreateEvent(Ndb* myNdb,
      const char *eventName,
      const char *eventTableName,
      const char **eventColumnNames,
      const int noEventColumnNames,
                  bool merge_events)
{
  NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  if (!myDict) APIERROR(myNdb->getNdbError());

  const NdbDictionary::Table *table= myDict->getTable(eventTableName);
  if (!table) APIERROR(myDict->getNdbError());

  NdbDictionary::Event myEvent(eventName, *table);
  myEvent.addTableEvent(NdbDictionary::Event::TE_ALL);
  //  myEvent.addTableEvent(NdbDictionary::Event::TE_INSERT);
  //  myEvent.addTableEvent(NdbDictionary::Event::TE_UPDATE);
  //  myEvent.addTableEvent(NdbDictionary::Event::TE_DELETE);
```

```
  myEvent.addEventColumns(noEventColumnNames, eventColumnNames);
  myEvent.mergeEvents(merge_events);

  // Add event to database
  if (myDict->createEvent(myEvent) == 0)
    myEvent.print();
  else if (myDict->getNdbError().classification ==
     NdbError::SchemaObjectExists) {
    printf("Event creation failed, event exists\n");
    printf("dropping Event...\n");
    if (myDict->dropEvent(eventName)) APIERROR(myDict->getNdbError());
    // try again
    // Add event to database
    if ( myDict->createEvent(myEvent)) APIERROR(myDict->getNdbError());
  } else
    APIERROR(myDict->getNdbError());

  return 0;
}
```

6.7. Event Handling with Multiple Clusters

This example illustrates the handling log events using the MGM API on multiple clusters in a single application.

The source code for this program may be found in the MySQL 5.1 source tree, in the file `storage/ndb/ndbapi-examples/mgmapi_logevent_dual/mgmapi_logevent_dual.cpp`.

```
#include <mysql.h>
#include <ndbapi/NdbApi.hpp>
#include <mgmapi.h>
#include <stdio.h>

/*
 * export LD_LIBRARY_PATH=../../../libmysql_r/.libs:../../../ndb/src/.libs
 */

#define MGMERROR(h) \
{ \
  fprintf(stderr, "code: %d msg: %s\n", \
          ndb_mgm_get_latest_error(h), \
          ndb_mgm_get_latest_error_msg(h)); \
  exit(-1); \
}

#define LOGEVENTERROR(h) \
{ \
  fprintf(stderr, "code: %d msg: %s\n", \
          ndb_logevent_get_latest_error(h), \
          ndb_logevent_get_latest_error_msg(h)); \
  exit(-1); \
}

int main(int argc, char** argv)
{
  NdbMgmHandle h1,h2;
  NdbLogEventHandle le1,le2;
  int filter[] = { 15, NDB_MGM_EVENT_CATEGORY_BACKUP,
       15, NDB_MGM_EVENT_CATEGORY_CONNECTION,
       15, NDB_MGM_EVENT_CATEGORY_NODE_RESTART,
       15, NDB_MGM_EVENT_CATEGORY_STARTUP,
       15, NDB_MGM_EVENT_CATEGORY_ERROR,
       0 };
  struct ndb_logevent event1, event2;

  if (argc < 3)
  {
    printf("Arguments are <connect_string cluster 1> <connect_string cluster 2> [<iterations>].\n");
    exit(-1);
  }
  const char *connectstring1 = argv[1];
  const char *connectstring2 = argv[2];
  int iterations = -1;
  if (argc > 3)
    iterations = atoi(argv[3]);
```

```
ndb_init();

h1= ndb_mgm_create_handle();
h2= ndb_mgm_create_handle();
if ( h1 == 0 || h2 == 0 )
{
  printf("Unable to create handle\n");
  exit(-1);
}
if (ndb_mgm_set_connectstring(h1, connectstring1) == -1 ||
    ndb_mgm_set_connectstring(h2, connectstring1))
{
  printf("Unable to set connectstring\n");
  exit(-1);
}
if (ndb_mgm_connect(h1,0,0,0)) MGMERROR(h1);
if (ndb_mgm_connect(h2,0,0,0)) MGMERROR(h2);

if ((le1= ndb_mgm_create_logevent_handle(h1, filter)) == 0) MGMERROR(h1);
if ((le2= ndb_mgm_create_logevent_handle(h1, filter)) == 0) MGMERROR(h2);

while (iterations-- != 0)
{
  int timeout= 1000;
  int r1= ndb_logevent_get_next(le1,&event1,timeout);
  if (r1 == 0)
    printf("No event within %d milliseconds\n", timeout);
  else if (r1 < 0)
    LOGEVENTERROR(le1)
  else
  {
    switch (event1.type) {
    case NDB_LE_BackupStarted:
printf("Node %d: BackupStarted\n", event1.source_nodeid);
printf("  Starting node ID: %d\n", event1.BackupStarted.starting_node);
printf("  Backup ID: %d\n", event1.BackupStarted.backup_id);
break;
    case NDB_LE_BackupCompleted:
printf("Node %d: BackupCompleted\n", event1.source_nodeid);
printf("  Backup ID: %d\n", event1.BackupStarted.backup_id);
break;
    case NDB_LE_BackupAborted:
printf("Node %d: BackupAborted\n", event1.source_nodeid);
break;
    case NDB_LE_BackupFailedToStart:
printf("Node %d: BackupFailedToStart\n", event1.source_nodeid);
break;

    case NDB_LE_NodeFailCompleted:
printf("Node %d: NodeFailCompleted\n", event1.source_nodeid);
break;
    case NDB_LE_ArbitResult:
printf("Node %d: ArbitResult\n", event1.source_nodeid);
printf("  code %d, arbit_node %d\n",
       event1.ArbitResult.code & 0xffff,
       event1.ArbitResult.arbit_node);
break;
    case NDB_LE_DeadDueToHeartbeat:
printf("Node %d: DeadDueToHeartbeat\n", event1.source_nodeid);
printf("  node %d\n", event1.DeadDueToHeartbeat.node);
break;

    case NDB_LE_Connected:
printf("Node %d: Connected\n", event1.source_nodeid);
printf("  node %d\n", event1.Connected.node);
break;
    case NDB_LE_Disconnected:
printf("Node %d: Disconnected\n", event1.source_nodeid);
printf("  node %d\n", event1.Disconnected.node);
break;
    case NDB_LE_NDBStartCompleted:
printf("Node %d: StartCompleted\n", event1.source_nodeid);
printf("  version %d.%d.%d\n",
       event1.NDBStartCompleted.version >> 16 & 0xff,
       event1.NDBStartCompleted.version >> 8 & 0xff,
       event1.NDBStartCompleted.version >> 0 & 0xff);
break;
    case NDB_LE_ArbitState:
printf("Node %d: ArbitState\n", event1.source_nodeid);
printf("  code %d, arbit_node %d\n",
```

```
       event1.ArbitState.code & 0xffff,
       event1.ArbitResult.arbit_node);
break;

    default:
break;
    }
  }

  int r2= ndb_logevent_get_next(le1,&event2,timeout);
  if (r2 == 0)
    printf("No event within %d milliseconds\n", timeout);
  else if (r2 < 0)
    LOGEVENTERROR(le2)
  else
  {
    switch (event2.type) {
    case NDB_LE_BackupStarted:
printf("Node %d: BackupStarted\n", event2.source_nodeid);
printf("  Starting node ID: %d\n", event2.BackupStarted.starting_node);
printf("  Backup ID: %d\n", event2.BackupStarted.backup_id);
break;
    case NDB_LE_BackupCompleted:
printf("Node %d: BackupCompleted\n", event2.source_nodeid);
printf("  Backup ID: %d\n", event2.BackupStarted.backup_id);
break;
    case NDB_LE_BackupAborted:
printf("Node %d: BackupAborted\n", event2.source_nodeid);
break;
    case NDB_LE_BackupFailedToStart:
printf("Node %d: BackupFailedToStart\n", event2.source_nodeid);
break;

    case NDB_LE_NodeFailCompleted:
printf("Node %d: NodeFailCompleted\n", event2.source_nodeid);
break;
    case NDB_LE_ArbitResult:
printf("Node %d: ArbitResult\n", event2.source_nodeid);
printf("  code %d, arbit_node %d\n",
       event2.ArbitResult.code & 0xffff,
       event2.ArbitResult.arbit_node);
break;
    case NDB_LE_DeadDueToHeartbeat:
printf("Node %d: DeadDueToHeartbeat\n", event2.source_nodeid);
printf("  node %d\n", event2.DeadDueToHeartbeat.node);
break;

    case NDB_LE_Connected:
printf("Node %d: Connected\n", event2.source_nodeid);
printf("  node %d\n", event2.Connected.node);
break;
    case NDB_LE_Disconnected:
printf("Node %d: Disconnected\n", event2.source_nodeid);
printf("  node %d\n", event2.Disconnected.node);
break;
    case NDB_LE_NDBStartCompleted:
printf("Node %d: StartCompleted\n", event2.source_nodeid);
printf("  version %d.%d.%d\n",
       event2.NDBStartCompleted.version >> 16 & 0xff,
       event2.NDBStartCompleted.version >> 8 & 0xff,
       event2.NDBStartCompleted.version >> 0 & 0xff);
break;
    case NDB_LE_ArbitState:
printf("Node %d: ArbitState\n", event2.source_nodeid);
printf("  code %d, arbit_node %d\n",
       event2.ArbitState.code & 0xffff,
       event2.ArbitResult.arbit_node);
break;

    default:
break;
    }
  }
}

ndb_mgm_destroy_logevent_handle(&le1);
ndb_mgm_destroy_logevent_handle(&le2);
ndb_mgm_destroy_handle(&h1);
ndb_mgm_destroy_handle(&h2);
ndb_end(0);
```

```
  return 0;
}
```

Index

E

F

G

I

L

M

N

O

P

R

S

T

U

W