Embedded InnoDB 1.0 (Early Adopter Release) User’s Guide and Reference

Embedded InnoDB 1.0 (Early Adopter Release) User’s Guide and Reference

This is the User’s Guide and Reference for the Embedded InnoDB 1.0, generated on February 25, 2010 (rev 6762).
Copyright © 2009, 2010 Innobase Oy. All rights reserved.
Copyright © 2009, 2010 Oracle. All rights reserved.

Abstract

Embedded InnoDB provides database capabilities that you can embed in an application, without any database installation or configuration.

Provided that you reproduce the above copyright notice, this list of conditions and the following disclaimer on all copies of this documentation (the “Documentation”) for Embedded InnoDB (the “Program”): (a) permission is hereby given to make a reasonable number of complete copies of the Documentation in formats of your choice, solely for your use with the Program, and (b) you may copy and distribute unmodified and complete versions of the content of this Documentation in format(s) of your choice, but only in conjunction with your otherwise-authorized distribution of the Program.

THIS DOCUMENTATION IS PROVIDED BY INNOBASE OY AND ITS CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL INNOBASE OY OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

Chapter 1. Concepts and Architecture for Embedded InnoDB

Welcome to Embedded InnoDB, a general-purpose embedded database engine that is capable of providing a wealth of data management services. It is designed from the ground up for high-throughput applications requiring in-process, bullet-proof management of mission-critical data. Embedded InnoDB can gracefully scale from managing a few bytes to terabytes of data. For the most part, Embedded InnoDB is limited only by your system's available physical resources.

1.1. Overview of Embedded InnoDB

You use Embedded InnoDB through a series of programming APIs that give you the ability to read and write your data, manage your database(s), and perform other more advanced activities such as managing transactions.

Because Embedded InnoDB is an embedded database engine, it is extremely fast. You compile and link it into your application in the same way as you would any third-party library. This means that Embedded InnoDB runs in the same process space as does your application, allowing you to avoid the high cost of interprocess communications incurred by stand-alone database servers.

Note

Before going any further, it is important to mention that Embedded InnoDB is not a relational database (although you could use it to build a relational database). Out of the box, Embedded InnoDB does not provide higher-level features such as triggers, or a high-level query language such as SQL. Instead, Embedded InnoDB provides just those APIs required to store and retrieve your data as efficiently as possible.

Embedded InnoDB lets you embed database processing within an application, without the need for an actual installation of MySQL or another database. The database operations are performed by an engine derived from the InnoDB plugin. You can perform database operations such as queries returning result sets, insert, update, and delete logic on database structures such as tables and columns, and transaction management. You call C or C++ functions to perform these operations and process the results, rather than submitting SQL statements.

1.1.1. Comparison with Database Application Development

In comparison to other compact or embedded database systems you might be familiar with:

- You work with tables, columns, and indexes.
- You can treat data as distinct types such as `VARCHAR`, `INTEGER`, and so on, or as raw bytes (similar to a `BLOB`).
- All your application logic goes through API calls, not a SQL interface.
- Your application requires zero installation and configuration other than providing your own executable program. The library can be linked statically or dynamically into your application. All configuration parameters have default values that apply if you do not set them explicitly. All necessary files are created if they do not already exist, inside the application's current working directory unless otherwise specified. Currently, no environment variables are needed or used.
- Your database work is done inside ACID-compliant transactions, with commit / rollback / savepoint capability.

  Because DDL operations such as creating tables cause changes in the data dictionary, you must enclose DDL operations within transactions. The data dictionary must be locked in exclusive mode when creating tables using the API function `ib_schema_lock_exclusive()`. Although for safety you must enclose DDL operations within a transaction, these operations cannot be rolled back.

- You retrieve query results by iterating through cursors. The cursor moves in sequence through a set of index values; you must step through the cursor and test the column values for each result, to determine when to stop reading.
- You manipulate column values through structures called *tuples*. Depending on the type of operation, these structures represent different groups of indexed columns, or all the columns of a table.
- You have control over low-level details such as transaction isolation levels and lock modes for tables and cursors.
- Your application should use a pre-planned, stable data model. You cannot change column definitions after creating a table.
- You can quickly look up data using a key value or range of values, by referencing an index on any column. To perform more complex kinds of queries using multiple conditions or tables, your application must perform some of the logic and optimizations normally applied by a SQL parser.
- All data is written to and read from disk. A certain amount of buffering and caching is handled by the InnoDB engine, with recovery if something goes wrong before or during write operations.
- Indexes are based on B-tree data structures. They can be unique (to represent a primary key) or can index values that contain duplicates. Indexes do represent null values, in contrast to some database systems that leave nulls out of indexes.
- Each database is represented by its own directory.
- By default, all tables are stored in one data file. When the `file_per_table` option is set, each table is stored in a separate data file.
- Every table has a primary key. The primary key is referred to as a clustered index, because the table data is stored within the index itself, based on the primary key order. (Similar to an Oracle index-organized table.) If your application does not create a primary key, InnoDB creates one based on the row ID.
- Indexes other than the primary key are referred to as secondary indexes. They have the flexibility to allow duplicates and overlook nulls in the indexed columns.
- The definition of each table or index is referred to as its schema. You use a sequence of API calls to set up a table schema or index schema, and another API call to create the table or index. (As opposed to a `CREATE TABLE` or `CREATE INDEX` statement in SQL, which performs those operations in a single step.)

Examples are given throughout this book that are designed to illustrate API usage. These examples build upon each other; for brevity, the later examples omit some of the repetitive initialization code such as starting a transaction. The sample programs, including all the required initialization, are available in the Embedded InnoDB software distribution. You can find them in:

```
share/innodb-release-version/examples
```

1.1.2. Comparison with InnoDB Plugin

Being familiar with the InnoDB storage engine, and in particular the latest InnoDB technology available through the InnoDB Plugin, can give you a head start in developing Embedded InnoDB applications. For information about the Plugin or to download it, see:

```
http://www.innodb.com/products/innodb_plugin/
```

For InnoDB terminology that applies to Embedded InnoDB development, see Embedded InnoDB Glossary.

The version number of the Embedded InnoDB product reflects the version number of the underlying InnoDB Plugin. For example, Embedded InnoDB version 1.0.3.5325 was based on InnoDB Plugin version 1.0.3.

These features in particular are inherited from the InnoDB Plugin:

- Index creation is much faster than historically has been the case in InnoDB.
- Table data can be compressed on disk, for tables that use the new “Barracuda” file format.
- Data from multiple tables can be stored together in a single file (the “system tablespace”), or in a separate file for each table, or a combination of those approaches.

Some InnoDB Plugin features are not available:

- Foreign key relationships are not supported.
- Auto-increment columns are not available.
- Semi-consistent read is not available.
- Descending indexes are not available.

1.1.3. Comparison with MySQL

If you like, you can develop and deploy Embedded InnoDB applications without using MySQL at all.

You might find it helpful to be familiar with the capabilities of MySQL and the InnoDB storage engine to help design your application logic, tune your data-manipulation code, and debug any problems. For example, if you have existing MySQL table definitions, you can see how the types are mapped to Embedded InnoDB constant values, as explained in Section 6.2.4, “Column Types”.

For even faster prototyping and debugging, you can transfer data directories back and forth between a MySQL installation and an Embedded InnoDB application. For example, you could create a set of InnoDB tables using SQL in MySQL, then perform all the data manipulation through an Embedded InnoDB program. Or you could set up data through an Embedded InnoDB program, then verify the data by running queries from MySQL. For details about this process, see Chapter 10, *Transferring Databases between MySQL and Embedded InnoDB*.

Features from MySQL that require higher-level interfaces are not available in Embedded InnoDB programs, for example:

- The `INFORMATION_SCHEMA` tables.
- `.FRM` files.
- Partitioned tables.
- Replication.
- Auto-commit.

1.2. Overview of Tables and Table Data

InnoDB databases contain tables. Tables have the familiar structure of rows, columns, and datatypes for the columns.

Tables contain records. (You can think of records as rows. We refer to records only to be precise, because some of the DML operations use structures that are a little different than traditional table rows.)

The column values for each record are represented in memory by a tuple. When you insert a record, you fill a set of column values into a tuple. When you update a record, you read the set of current column values into a tuple, modify one or more columns, then store that tuple back into the same record. When you retrieve data from a table, the columns come back through a tuple.

For fuller definitions of these terms and the relationships between them, see the Glossary.

1.2.1. Overview of the Data Dictionary

Embedded InnoDB, being based on the underlying InnoDB storage engine, keeps track of tables, indexes, and table columns. This information is stored in a set of system tables called the data dictionary. Embedded InnoDB applications are not prevented from writing to these tables as in regular MySQL usage; however, you must treat these tables as read-only.

The system tables are:

- `SYS_TABLES` - Contains all the tables
- `SYS_COLUMNS` - Contains all the columns for all the tables
- `SYS_INDEXES` - Contains all the indexes defined on all the tables
- `SYS_FIELDS` - Contains all the columns for all the indexes

For efficiency in setting up multiple data structures, Embedded InnoDB does not wrap each table or index creation operation within a transaction. You must wrap these operations within calls to lock and unlock the data dictionary, and to start and commit transactions. You will see examples of the proper sequences of function calls later on.

Each separate program using the Embedded InnoDB library must use its own data dictionary and data files.

1.2.2. Overview of Data Files

Data files are stored by default underneath the current working directory of the application. You can specify a different location by calling the configuration APIs.

The system tablespace, which holds the tables of the data dictionary and by default all the tables used by your application, is a single large file. If you change the configuration settings to use the file-per-table option, each table created afterwards is stored in a separate file.

1.3. Overview of Indexes

A table can have one or more indexes defined on it.

InnoDB requires that every table must have a primary key defined on it, which we refer to as the clustered index. InnoDB automatically creates a clustered index you do not specify one. The table records are physically organized to be in the order specified by this index. In fact, all the table data is stored in the index itself. (Thus, you should choose primary key columns that have relatively stable values, to avoid records moving around to maintain the index order.)

You use the clustered index when performing write operations, when accessing columns that are not covered by other indexes, and typically for your most important queries.

All other indexes are referred to as secondary indexes. These indexes include a subset of the table columns (the indexed columns, plus the primary key columns), making them more suitable for data that is frequently inserted, updated, and deleted. You can retrieve the values for indexed columns directly from these indexes.

When you use a secondary index to find records, but need to retrieve columns that are not part of the secondary index: set the “need access to clustered index” flag in the secondary index cursor (by calling `ib_cursor_set_cluster_access()` on the secondary index cursor), find the relevant entry in the secondary index, then use the secondary index cursor to read the complete record with a clustered index read tuple.

For either kind of index, the index key can be a single column, or a composite key derived from multiple columns. When you use a composite index, the first column in the index is the most important from a query perspective. The other indexed columns are primarily to help retrieve results in order by those columns.

For fuller definitions of these terms and the relationships between them, see the Glossary.

1.4. Overview of Queries and WHERE Clauses

Note

Pay close attention to this section! The details of how you position and move cursors, and translate SQL `WHERE` clauses into your own application logic, are probably the most challenging concepts to grasp when coming from a SQL background with a full-fledged database.

High-end databases process complex SQL queries and produce a result set, representing exactly those rows that match all the query conditions. Embedded databases typically allow for simpler queries, such as looking up a single item by searching for a unique key.

Embedded InnoDB occupies a middle ground. When you position a cursor, you supply a single comparison operator that works on a single column. Embedded InnoDB puts the cursor at the appropriate starting place within the table. You can read forward or backward, examining other columns as you go. By coding your own application logic, you can emulate more complicated queries. Coding queries at such a low level avoids some of the usual SQL parsing overhead, and allows you to choose your own optimization techniques.

For fuller definitions of these terms and the relationships between them, see the Glossary.

For the information needed to use these APIs, after learning the background information in this section, see:

- Section 4.3.5, “Emulating a WHERE Clause”
- Section 6.7.7, “API Functions - Tuples”
- Appendix B, *Code Examples Supplied with Embedded InnoDB*

1.4.1. Overview of Cursors

Cursors provide a mechanism to iterate over the records in a table. Using cursors, you can read, write, and delete database records. You perform all cursor operations by passing a handle (an opaque type) to the relevant functions. As each cursor is created, it is attached to a transaction. All changes to the database using the cursor are part of that same transaction; you must make sure that you are finished using the cursor before the transaction ends. Once a transaction is committed or rolled back, the cursor handle is considered invalid. All subsequent results from using the stale cursor handle are undefined. You receive a cursor handle by opening an existing table or index, using the ID or the name of the applicable table or index. (The relevant API calls are `ib_cursor_open_table`, `ib_cursor_open_table_using_id`, `ib_cursor_open_index_using_name`, and `ib_cursor_open_index_using_id`.)

You can reuse a cursor by calling `ib_cursor_reset` on the cursor handle and attaching it to a running transaction. You close a cursor by calling `ib_cursor_close`.

1.4.2. Overview of Tuples

Tuples are memory structures that represent different sets of columns, You use them for all kinds of DML operations. In a DML operation, they represent the columns of a record being inserted or updated. In a query, they represent columns from a record pointed to be a cursor.

There are different kinds of tuples derived from the clustered index or a secondary index. There is also a distinction between tuples used for simply searching for a record, and for reading back the columns of the record. In all, you work with four different kinds of tuples based on the above combinations.

1.4.2.1. The “Search” Tuple

A search tuple is a fast, compact structure that only contains the columns needed to perform a search. You use this kind of tuple to perform queries where the answer requires only the columns from a single index.

1.4.2.2. The “Read” and “Read/Write” Tuples

A read tuple is a larger structure containing extra columns so that you can get back values not just from indexed columns. When inserting or updating row you use a “read” tuple.

When this tuple comes from the clustered index (that is, the primary key), it contains all the columns of the table.

When this tuple comes from a secondary index, in addition to the columns from the index, it also contains the primary key columns. If the query can be satisfied with the columns from the secondary index and the clustered index, this tuple is all you need. If you need more columns that are not covered by those two indexes, you can set the “clustered index access flag” before searching, and use a clustered index read tuple to read the columns from the cluster index using the secondary index cursor that was used for searching.

1.5. Overview of Transactions

All database operations need to be covered by transactions. You start by acquiring a transaction handle by calling the `ib_trx_begin()` function and passing an isolation level for the transaction.

```
ib_trx_t   ib_trx = ib_trx_begin(IB_TRX_SERIALIZABLE);
```

When a transaction is committed, all changes made within that transaction are made permanent. For example:

```
err = ib_trx_commit(ib_trx);
```

When a transaction is rolled back, all changes made within that transaction are discarded. For example:

```
err = ib_trx_rollback(ib_trx);
```

Once a transaction is committed or rolled back, the cursor handle is considered invalid. All subsequent results from using the stale cursor handle are undefined.

There is one special case to consider with transactions. When the InnoDB deadlock monitor selects a victim (an active transaction) to roll back, the selected transaction is rolled back by InnoDB. You cannot commit or roll back this transaction using the above commit/rollback functions. The handle can either be freed or reused to start a new transaction. The function calls for this special case are:

```
ib_trx_release(ib_trx)
```

and

```
ib_trx_start(ib_trx)
```

To query the transaction state you can use:

```
ib_trx_state_t trx_state = ib_trx_state(ib_trx)
```

and examine the state returned by this function.

For fuller definitions of these terms and the relationships between them, see the Glossary.

1.5.1. How Isolation Levels Affect Transactions

Each transaction has a property called the *isolation level* that controls how strictly the InnoDB engine protects different transactions from “seeing” uncommitted changes made by other transactions. For maximum correctness and reliability, you should typically choose the `IB_REPEATABLE_READ` isolation level. For maximum performance in advanced applications, where you can ensure that this level of strictness is not required, you might choose a different isolation level, as explained in Section 5.3, “Setting the Transaction Isolation Level”.

1.5.2. Overview of Savepoints

Savepoints help to implement nested transactions. They can be used to provide scope to operations on tables that are part of a larger transaction. For example, scheduling a trip in a reservation system might involve booking several different flights; if a desired flight is unavailable, you might roll back the changes involved in booking that one leg, without rolling back the earlier flights that were successfully booked. Embedded InnoDB provides these functions for savepoints:

- `ib_savepoint_take`

  Creates a named savepoint. If the transaction is not yet started, starts it. If there is already a savepoint of the same name, this call erases that old savepoint and replaces it with a new one. Savepoints are deleted in a transaction commit or rollback.

- `ib_savepoint_release`

  Releases only the named savepoint. Any savepoints that were set after this savepoint are left as-is.

- `ib_savepoint_rollback`

  Rolls back a transaction back to a named savepoint. Modifications after the savepoint are undone, but Embedded InnoDB does **not** release the corresponding locks that are stored in memory. If a lock is “implicit”, that is, a new inserted row holds a lock where the lock information is carried by the transaction ID stored in the row, these locks are naturally released in the rollback. Any savepoints that were set after this savepoint are deleted. If name equals `NULL` (the C NULL value), then all the savepoints are rolled back.

1.6. Overview of Security

Any data encryption or protection for the data files must be provided by your application. There is no security provided by Embedded InnoDB and no encryption. It does not have any notion of authentication or authorization.

Chapter 2. Getting Started with Embedded InnoDB

2.1. Downloading Embedded InnoDB

Download Embedded InnoDB (source, or binaries for supported platforms) by visiting the Embedded InnoDB download page:

```
http://www.innodb.com/products/embedded-innodb/download/
```

2.2. Prerequisites for Embedded InnoDB

Ensure that your development systems have the appropriate operating systems, libraries, and development tools needed to build Embedded InnoDB applications.

2.2.1. Prerequisites for Linux and UNIX Systems

- You must have the GNU Compiler Collection and GNU `autotools` package installed.
- By default, you must have root privileges so that you can install the shared libraries in system directories such as `/usr/local/lib`. You can also change the shared library location as part of the build process, to store the library in a directory that you own.

2.2.2. Prerequisites for Windows Systems

- To build from source, you must have the `cmake` command, available as part of the CMake open source package from:

  ```
  http://www.cmake.org/
  ```

- You must have the Microsoft Visual Studio product. (See the `README` file for the build options corresponding to different levels of the Microsoft Visual Studio product.)
- To build a Embedded InnoDB library that includes the table compression feature, you must have the open source `zlib` library on your system. This library is included by default on most Linux and UNIX systems; Windows users can download it from:

  ```
  http://www.zlib.net
  ```

2.3. Installing Embedded InnoDB

To install Embedded InnoDB:

- Untar or unzip the distribution to the directory of your choice.
- If you downloaded the source distribution, build the product binaries. For information on building Embedded InnoDB, see `README` in the top-level directory in the source distribution.

In particular, if you want to create a smaller shared library or remove the dependency on the `zlib`, you can turn off the InnoDB compression feature by including the option `--disable-compression` when running the `configure` command. You can also turn off the XA support by including the option `--disable-xa` when running the `configure` command.

- Set up your application build environment to link in the Embedded InnoDB library, whose file name starts with `libinnodb`. On Linux and UNIX systems, include the directory name in the value of the `$LIBPATH` variable. On Windows systems, put the library in a directory that is listed in the `%PATH%` variable.

Consult the Embedded InnoDB sample programs to see the outline and common operations for an Embedded InnoDB application. On Linux and UNIX systems, these sample programs are installed under `/usr/local/share/embedded_innodb-1.0/examples`. On Windows systems, these sample programs are installed under `embedded_innodb-1.0\examples`. See Appendix B, *Code Examples Supplied with Embedded InnoDB* for details about each one.

2.4. Starting the InnoDB Database Engine

Each time your program that uses the Embedded InnoDB libraries starts, it must start the InnoDB engine; the startup process creates I/O and background threads. Once control returns from the `ib_startup` function, it is safe to use InnoDB for normal database operations.

When InnoDB is run for the first time with a particular data directory, it creates the system tablespaces and the log files.

When InnoDB is started on a previous instance of a database, InnoDB may perform recovery first, depending on whether it was shut down cleanly or not.

```
ib_err_t err;
ib_u64_t version;

/* Check the API version. */
version = ib_api_version();

/* The format of version is:
   | 16 bits future use | 16 bits current | 16 bits revision | 16 bits age | */

/* Initialize the memory sub-system. */
ib_init();

/* Call the ib_cfg_*() functions to setup the directory etc. */

/* Create system files if this is the first time
or do recovery if starting an existing instance. */
err = ib_startup("barracuda");
/* File format "barracuda" supports all
the currently available table formats. */

if (err == DB_SUCCESS) {
 printf("InnoDB started!\n");
} else {
 printf("Error starting up InnoDB: %s\n", ib_strerror(err));
}

err = ib_shutdown(IB_SHUTDOWN_NORMAL);
```

```
if (err == DB_SUCCESS) {
 printf("InnoDB shutdown succeed!\n");
} else {
 printf("InnoDB shutdown failed: err %s\n", ib_strerror(err));
}
```

For further information, see Section 6.7.1, “API Functions - Startup/Shutdown”.

2.5. Configuration

InnoDB operates under the control of a set of configuration variables, for example, to turn particular features on and off, or to specify the locations of data files. Embedded InnoDB uses API calls to modify these settings.

The Embedded InnoDB configuration variables are listed and described in Chapter 8, *Embedded InnoDB Configuration Variables*.

The values, scope, and workings of all the InnoDB configuration options are documented in the MySQL manual [http://dev.mysql.com/doc/refman/5.1/en/innodb-configuration.html].

There are three types of configuration variables: integer, text and boolean. All three types can be set using either a generic function call:

```
ib_cfg_set(const char*, ...)
```

or by using the macros listed below. We recommend that you use the macros, for readability without sacrificing performance.

For information on the data types used with the configuration APIs, see Section 6.2.2, “Configuration Variable Types”. For information on the configuration APIs themselves, see Section 6.7.4, “API Functions - Configuration”.

2.5.1. Setting Integer Variables

In this example, we set the InnoDB buffer pool size to 128 megabytes.

```
ib_err_t err;

err = ib_cfg_set_int("buffer_pool_size", 134217728)

if (err != DB_SUCCESS) {
 fprintf(stderr, "Error setting InnoDB buffer pool size: %s\n",
  ib_strerror(err));
}
```

2.5.2. Setting Boolean Values

In this example we enable the file-per-table setting:

```
err = ib_cfg_set_bool_on("file_per_table");
/* Check whether we were successful in enabling the setting or not. */
```

In this example we disable the file per table setting:

```
err = ib_cfg_set_bool_off("file_per_table");
/* Check whether we were successful in disabling the setting or not. */
```

2.5.3. Setting Text Values

In this example we set the InnoDB data file path:

```
err = ib_cfg_set_text("data_file_path", "ibdata1:32M:autoextend");
/* Check whether we were successful in setting the data file path. */
```

2.6. Application Usage

In between the startup and shutdown operations, your application performs its data management and data processing. Those operations are the focus of the remainder of this book:

- Chapter 3, *Performing CREATE and DROP Operations (DDL) with Embedded InnoDB*
- Chapter 4, *Performing SELECT, INSERT, UPDATE, and DELETE (DML) Operations with Embedded InnoDB*
- Chapter 5, *Managing Transactions with Embedded InnoDB*
- Chapter 6, *C API Reference for Embedded InnoDB*

2.7. Shutting Down the InnoDB Database Engine

Once your program is finished with the InnoDB engine, shut it down cleanly using the `ib_shutdown` function, specifying one of the constants listed in Section 6.3.1, “Shutdown Modes”. There must be no active transactions when shutdown is initiated. All transactions must either be committed or rolled back. It is always good practice to shut down the engine once you are done with it; shutting down cleanly avoids any recovery processing the next time you start the engine.

2.7.1. Potential Shutdown Issues

- Sometimes shutdown may take longer than expected, because data may be buffered during `INSERT` operations. The buffered data is written to disk before the engine shuts down. The engine also may need to complete outstanding `DELETE` operations where the records are marked for removal from the table, but not removed immediately; this is known as the “purge” step.

Chapter 3. Performing CREATE and DROP Operations (DDL) with Embedded InnoDB

This section discusses how to perform the equivalent of SQL `CREATE` and `DROP` operations with the Embedded InnoDB APIs. `ALTER` operations are currently not supported.

This section refers to tables, indexes, and other database terminology that may have special nuances within Embedded InnoDB; familiarize yourself with these nuances by reading Chapter 1, *Concepts and Architecture for Embedded InnoDB* first, and consult Embedded InnoDB Glossary for definitions of any unfamiliar terms.

Throughout this section, we refer to functions and constants that are defined later, in Chapter 6, *C API Reference for Embedded InnoDB*, and especially in Section 6.7.5, “API Functions - DDL”.

3.1. Managing Databases

InnoDB maps a database to a file system directory. To create a table in a particular database, use the following naming scheme: “*`database_name/table_name`*”. (Always pass this fully qualified form of the name to APIs that require a table name.)

If the **file_per_table** configuration variable is set, InnoDB creates the underlying directory, in a location relative to the **data_home_dir** configuration setting.

3.1.1. Creating a Database

If **file_per_table** configuration variable is set, InnoDB uses one data file per table. Otherwise, all user tables and indexes are stored in one *data* tablespace. The underlying directory must exist, or it can be created by the `ib_database_create` function:

```
ib_err_t err = ib_database_create("test");
```

This name must be a simple database name, not a path.

3.1.2. Dropping a database

Dropping a database drops all tables in the database and then deletes the underlying directory.

```
ib_err_t err = ib_database_drop("test");
```

3.2. Managing Tables

The primary kind of object that you work with in a database is the table.

3.2.1. Creating a Table

Creating a table involves a sequence of API calls to set up the table and its related objects:

- The data structures for the table itself.
- Each column.
- The clustered index, representing the primary key.

When all the structures are set up and connected together, a final API call actually creates the table.

For example, imagine that you want to create a table with the following structure:

```
CREATE TABLE T(c1 VARCHAR(32), c2 VARCHAR(32), c3 INT UNSIGNED, PRIMARY KEY (c1, c2));
```

The steps involved are:

- Create a handle for the table schema (its column structure) by calling the function `ib_table_schema_create()`.
- Add column type information to the table schema by calling the function `ib_table_schema_add_col()`. Repeat for each column in the table.
- Add an index schema to the table instance by calling function `ib_table_schema_add_index()`.
- Define each index column by calling function `ib_index_schema_add_col()`.
- Set the index type to cluster, using the function `ib_index_schema_set_clustered()`. Every table must have a clustered index that represents its primary key.
- Request access to the data dictionary, by starting a transaction (`ib_trx_begin()`) and requesting a lock (`ib_schema_lock_exclusive()`).
- Create the table in InnoDB's data dictionary by calling function `ib_table_create()`.
- Commit the transaction by calling function `ib_trx_commit()`. (The commit function also releases the lock on the data dictionary.)
- Destroy the table schema handle by calling function `ib_table_schema_delete()`, to avoid memory leaks.

Note

Error checking has been omitted from the example to make it easier to follow the code. The API functions can return the error codes that are defined in Section 6.6, "Error and Status Codes".

```
ib_trx_t        ib_trx;
ib_id_t         table_id = 0;
ib_tbl_sch_t    ib_tbl_sch = NULL;
ib_idx_sch_t    ib_idx_sch = NULL;
char            table_name[IB_MAX_TABLE_NAME_LEN];

snprintf(table_name, sizeof(table_name), "%s/%s", database_name, name);

/* Pass a table page size of 0, ie., use default page size. */
ib_table_schema_create(table_name, &ib_tbl_sch, IB_TBL_COMPACT, 0);

/* The fifth argument is currently not used. */
ib_table_schema_add_col(ib_tbl_sch, "c1", IB_VARCHAR, IB_COL_NONE, 0, 32);
ib_table_schema_add_col(ib_tbl_sch, "c2", IB_VARCHAR, IB_COL_NONE, 0, 32);
ib_table_schema_add_col(ib_tbl_sch, "c3", IB_INT, IB_COL_UNSIGNED, 0, 4);

/* Index schema handle is "owned" by the table schema handle in this
case and will be deleted when the table schema handle is deleted. */

ib_table_schema_add_index(ib_tbl_sch, "PRIMARY_KEY", &ib_idx_sch);
```

```
/* Set prefix length to 0. */
ib_index_schema_add_col(ib_idx_sch, "c1", 0);

/* Set prefix length to 0. */
ib_index_schema_add_col(ib_idx_sch, "c2", 0);

ib_index_schema_set_clustered(ib_idx_sch);

/* Create the transaction that will cover data dictionary update. */
ib_trx = ib_trx_begin(IB_TRX_SERIALIZABLE);

/* Lock the data dictionary in exclusive mode */
ib_schema_lock_exclusive(ib_trx);

/* Create the actual table from the schema. The table id of the new
table will be returned in table_id. */
ib_table_create(ib_trx, ib_tbl_sch, &table_id);

/* Commit the transaction */
ib_trx_commit(ib_trx);

ib_table_schema_delete(ib_tbl_sch);
```

3.2.1.1. Mapping MySQL Types to Embedded InnoDB Values

Whether you are adapting existing SQL code into an Embedded InnoDB program, writing a program that works with a existing MySQL tables, or approaching the application design from a database background, you must be able to map from MySQL column datatypes to the corresponding Embedded InnoDB data structures and values:

- When creating a table, you use the constant values from Section 6.2.4, “Column Types” to define the type of each column, through the `ib_table_schema_add_col()` function.
- For every column that you read or write, you create a data structure of type `ib_col_meta_t`, as described in Section 6.2.6, “Column Metadata”, and assign a type value to the `ib_col_type_t` field, using one of the constant values from Section 6.2.4, “Column Types”.

3.2.2. Opening a Table or Index

Before you can search, read, or write records, you need to open either a table or an index. Because every table has a primary key (either one you specified, or a default one supplied by the Embedded InnoDB engine), “opening a table” is shorthand for opening the table’s primary key index, also known as the clustered index.

The open operation creates a cursor handle that is then used to position, search, read, write, and delete records. All table and index operations require one of these cursor handles. (Subsequent sections explain the considerations for exactly which index to use, depending on the columns involved in the operation.)

To open a table, you specify its name. You must have an active transaction. The cursor is attached to this transaction, to prevent any further read, write, or retrieve operations using the cursor after the transaction is committed or rolled back.

```
ib_trx_t   ib_trx;
ib_crsr_t  ib_crsr;
```

```
ib_trx = ib_trx_begin(IB_TRX_REPEATABLE_READ);
ib_cursor_open_table(table_name, ib_trx, &ib_crsr);
```

Opening an index is very similar to opening a table. Instead of the table name, you specify the index name:

```
/* Open the secondary index "SEC_1" which is defined on the
table to which the cursor is attached. */
err = ib_cursor_open_index_using_name(crsr, "SEC_1", &index_crsr);
```

Or you can specify the index ID:

```
ib_trx_t   ib_trx;
ib_crsr_t  ib_crsr;

ib_trx = ib_trx_begin(IB_TRX_REPEATABLE_READ);
ib_cursor_open_index_using_id(index_id, ib_trx, &ib_crsr);
```

Note

To keep the sample code succinct, later examples may start at a point where you are expected to already have a transaction `ib_trx` open and a cursor `crsr` attached to it.

3.2.3. Closing a Table or Index

While performing operations on a table or index, you pass a cursor handle to each function. When finished with the operations, close the cursor by calling the `ib_cursor_close()` function. Once the cursor is closed, its memory is freed and it is unusable. There is no garbage collection of open cursors when transactions are committed or rolled back; you must explicitly close each cursor.

```
ib_cursor_close(ib_crsr);
```

You can call the function `ib_cursor_reset` to use the same cursor handle again. (Remember to attach the cursor to a running transaction after calling the reset function.)

3.2.4. Truncating a Table

When you want to keep a table, but delete all the data in the table, you can *truncate* the table by calling the `ib_cursor_truncate` or `ib_table_truncate` function. This operation is faster and more convenient than deleting all the records. If the operation is successful it will close the cursor and set it to NULL.

```
ib_cursor_truncate(&ib_crsr, &table_id);
```

```
ib_table_truncate(table_name, &table_id);
```

3.3. Managing Indexes

InnoDB supports two types of indexes:

Clustered index	Every table must have one and only one clustered index, equivalent to a primary key. The column with the clustered index cannot have any null or duplicate values, so that it can be used to uniquely identify every record in the table. If your code does not create one, InnoDB creates one automatically based on the system `DB_ROWID` column.

Secondary index	There can be many secondary indexes. Columns with secondary indexes can contain nulls and duplicate values. Unique secondary indexes cannot contain duplicate values.

3.3.1. Creating an Index

There are two ways to create an index within InnoDB:

1. Add indexes to the table schema when a new table is being created.
2. Create an index schema (a representation of the index columns) and call `ib_index_create`. This option is used to add secondary indexes to an existing table.

3.3.1.1. Recreating a Clustered Index

If you recreate the clustered index, for example to use your own definition instead of the system generated one, or to change the primary key, InnoDB creates a new table with the new clustered index. If any secondary indexes are defined on the original table, InnoDB creates new versions of those indexes on the new table. This operation results in new table and index IDs, since the old versions are dropped once the operation is complete. Once the old table is dropped, the new table is renamed to take its place.

3.3.2. Opening an Index

This operation is covered in Section 3.2.2, “Opening a Table or Index”.

Chapter 4. Performing SELECT, INSERT, UPDATE, and DELETE (DML) Operations with Embedded InnoDB

This section discusses how to perform the equivalent of SQL `SELECT`, `INSERT`, `UPDATE`, and `DELETE` operations with the Embedded InnoDB APIs.

This section refers to tables, indexes, and other database terminology that may have special nuances within Embedded InnoDB; familiarize yourself with these nuances by reading Chapter 1, *Concepts and Architecture for Embedded InnoDB* first, and consult Embedded InnoDB Glossary for definitions of any unfamiliar terms. This section assumes that the required tables, indexes, and so on already exist; see Chapter 3, *Performing CREATE and DROP Operations (DDL) with Embedded InnoDB* for instructions on creating these kinds of objects.

Throughout this section, we refer to functions and constants that are defined later, in Chapter 6, *C API Reference for Embedded InnoDB*, and especially in Section 6.7.6, “API Functions - DDL and DML Support”.

4.1. Mapping MySQL Types to Embedded InnoDB Values

Whether you are adapting existing SQL code into an Embedded InnoDB program, writing a program that works with a existing MySQL tables, or approaching the application design from a database background, you must be able to map from MySQL column datatypes to the corresponding Embedded InnoDB data structures and values:

- When creating a table, you use the constant values from Section 6.2.4, “Column Types” to define the type of each column, through the `ib_table_schema_add_col()` function.
- For every column that you read or write, you create a data structure of type `ib_col_meta_t`, as described in Section 6.2.6, “Column Metadata”, and assign a type value to the `ib_col_type_t` field, using one of the constant values from Section 6.2.4, “Column Types”.

4.2. Using Tuples

The tuple is an in-memory data structure that represents the columns from a table record, or the key values for an index record. Whenever you call functions to perform the equivalents of SQL `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements, you use tuples to specify the column values you are looking for, retrieve current column values, or store a new set of column values.

4.2.1. Creating a Tuple

Currently, there are two basic types of tuples:

- Tuple to read table rows.
- Tuple to search index keys.

The operations are a little different depending on whether you are working through a clustered index or a secondary index. Thus, you might create and work with each of these kinds of tuples:

- Tuple to read or write a row, from or to a clustered index.

- Tuple to read a row from a secondary index.
- Tuple to search a clustered index.
- Tuple to search a secondary index.

```
        ib_tpl_t   ib_tpl;
```

When doing substantial data processing, you will probably work with several tuples at a time. For example, to update a record, you read the original values into one tuple, then write updated values back to the table using another tuple.

```
/* Create a tuple for searching a clustered index. */
ib_tpl = ib_clust_search_tuple_create(crsr);

/* Create a tuple for searching a secondary index. */
ib_tpl = ib_sec_search_tuple_create(crsr);

/* Create a tuple for reading/writing a row from/to a clustered index. */
ib_tpl = ib_clust_read_tuple_create(crsr);

/* Create a tuple for reading a row from a secondary index. */
ib_tpl = ib_sec_read_tuple_create(crsr);
```

4.2.2. Deleting a Tuple

There is a single call to delete all four tuple types from above:

```
ib_tuple_delete(ib_tpl);
```

4.2.3. Column Metadata in a Tuple

All the metadata about a column is contained in each tuple. You can read the values of columns in the tuples by using either of these techniques, either copying the values or accessing the contents via a pointer:

- Reading typed values, using functions such as `ib_tuple_read_u8` to read an unsigned 8-byte integer. You ask explicitly for a data value of a particular type. This technique makes for reliable, readable code, but involves some extra overhead.
- Reading raw bytes, using functions such as `ib_col_copy_value` (copy data into a temporary variable) or `ib_col_get_value` (return a pointer to the data). This technique gives maximum performance, but you must handle the length calculations, endianness issues, and pointer manipulation yourself.

4.2.4. Scanning and Positioning

Whenever you would use a query or any kind of `WHERE` clause in SQL, with Embedded InnoDB you will position a cursor and scan forward or backward through the records.

4.2.4.1. Getting the Cursor Handle

All table and index operations are via cursor handles that are obtained using the steps outlined in the section Section 3.2.2, “Opening a Table or Index”. The cursor can be positioned by searching for a key, or by calling one of `ib_cursor_first` or `ib_cursor_last` functions to position the cursor at the first record or the last record in the table or index.

4.2.4.2. Full Table Scan

To iterate over *all* the records in a table, open the table or index to get a cursor handle, then use code like the following.

```
ib_err_t err;
ib_tpl_t ib_tpl;

/* Assuming we are iterating over the table, we would create
a tuple to read the data like so: */

ib_tpl = ib_clust_read_tuple_create(ib_crsr);
if (ib_tpl == NULL) {
 /* Handle out of memory error. */
} else {
 err = ib_cursor_first(ib_crsr);

 while (err == DB_SUCCESS) {
  /* Read the record and process it. */
  err = ib_cursor_read_row(ib_crsr, ib_tpl);

  /* Check for errors from the read ignored for brevity. */
  /* Do something with the tuple here. */

  err = ib_cursor_next(ib_crsr);
  /* Check for errors from the next ignored for brevity. */

  …
  /* Remember to reset the tuple that was used to
  read the record from the cursor. */
  ib_tpl = ib_tuple_clear(ib_tpl);
 }

 ib_tuple_delete(ib_tpl);
}
```

To process the records in a particular order, choose a column with that order, define an index using that column as the first column, and then open a cursor on that index. (Usually, the primary key for a table is defined using the most logical or common sort order for query results. That way, you can open the table as opposed to a specific index, and get results back in primary key order.)

4.2.4.3. Searching for Records

You search tables and indexes using the cursor interface functions. The search key can contain all the columns of the key, or a subset of the key columns. The columns must be set starting from the first column. The steps for searching for a record are as follows:

1. Create a key tuple handle by calling the function `ib_key_search_tuple_create()`.
2. Set the columns in the tuple handle by calling the function `ib_col_set_value()` one or more times.
3. Position the cursor by calling `ib_cursor_moveto()`.

Example code:

```
int       res;
```

```
    ib_tpl_t  key;
    ib_err_t  err;

    key = ib_sec_search_tuple_create(crsr);
    ib_col_set_value(key, 0, "a", strlen("a"));
    ib_col_set_value(key, 1, "x", strlen("x"));

    /* res will contain the result from comparing key and the user
    record that the cursor is positioned on. The result is undefined
    if the function returns DB_END_OF_INDEX or error. */
    err = ib_cursor_moveto(crsr, key, IB_CUR_GE, &res);

    ...
    ib_tuple_delete(key);
```

4.2.5. Reading a Column Value

Once the cursor is positioned on a particular record, you can read the columns from that record. What you do after reading the columns depends on the type of operation:

- In a query with a `WHERE` clause, you read the columns, check whether you could stop processing records yet, and do the appropriate output or manipulation of the column values.
- For a `DELETE` with a `WHERE` clause, you read the columns and decide whether or not to delete that particular record.
- For an `UPDATE` with a `WHERE` clause, you read the columns and decide whether or not to update that particular record. If the record should be updated, you make a new data structure (tuple) with a copy of the original column values, update the appropriate columns, and write the new record back to the table.

Here is an example to read values from a tuple, that also handles `NULL` values in columns. It assumes that you have already created a tuple holding a set of column values.

```
int             i;
int             n_cols = ib_tuple_get_n_cols(tpl);

for (i = 0; i < n_cols; ++i) {
 ib_ulint_t      data_len;
 ib_col_meta_t   ib_col_meta;

 /* Get the meta data for the ith column. */
 data_len = ib_col_get_meta(tpl, i, &ib_col_meta);

 /* Skip system columns. */
 if (ib_col_meta.type == IB_SYS) {
  continue;
 /* Nothing to print. Check for NULL values. */
 } else if (data_len == IB_SQL_NULL) {
  fprintf(stream, "|");
  continue;
 } else {
  switch (ib_col_meta.type) {
  case IB_INT: {
   /* You can either read the raw bytes and do the
   conversion yourself by twiddling the bits or use
```

```
         the typed interface of the InnoDB API to read the
         int values.

         The general strategy will be to determine the INT
         size from the column meta data size (type_len)
         along with the sign flag. Then call the appropriate
         API function to read the int type. */
         ...

         switch (ib_col_meta.type_len) {
         case 1:
         case 2:
         ...
         case 4: {
          ib_u32_t v;

          ib_tuple_read_i32(tpl, i, &v);
          fprintf(stream, "%d\n", v);
         }
         ...
         case 8:
         ...
         default:
          /* Data dictionary is corrupt! */
          assert(IB_FALSE);
         }
         break;
        }
        case IB_FLOAT: {
         float   v;

         ib_tuple_read_float(tpl, i, &v);
         fprintf(stream, "%f", v);
         break;
        }
        case IB_DOUBLE: {
         double  v;

         ib_tuple_read_double(tpl, i, &v);
         fprintf(stream, "%lf", v);
         break;
        }
        case IB_CHAR:
        case IB_BLOB:
        case IB_DECIMAL:
        case IB_VARCHAR: {
         const char*     ptr;

         /* Get access to the raw data via a pointer. */
         ptr = ib_col_get_value(tpl, i);
         fprintf(stream, "%.*s", (int) data_len, ptr);
         break;
        }
        default:
```

```
        assert(IB_FALSE);
        break;
      }
    }
    fprintf(stream, "|");
  }

  fprintf(stream, "\n");
```

4.2.6. Setting a Column Value

The API function for setting a column value is `ib_col_set_value()`. You specify a target tuple, the column number to set, a pointer to the data bytes, and the length of the raw data bytes. InnoDB does no type checking when setting the value of a column.

The column values are only stored in a table when you call a function to perform an insert or update using this tuple.

```
char*   c = "some value";

ib_col_set_value(tpl, col_no, c, strlen(c));
```

4.2.6.1. Setting a Column Value to SQL NULL

When a new tuple is created, the column values are all set to `IB_SQL_NULL`. Thus, for `INSERT` operations, you only need to fill in those columns that should be non-`NULL`.

To explicitly set the value of a column in a tuple to `IB_SQL_NULL`, for example during an `UPDATE` operation, call `ib_col_set_value` and set the *`data_len`* parameter to `IB_SQL_NULL`.

```
ib_col_set_value(tpl, col_no, NULL, IB_SQL_NULL);
```

4.2.7. Copying the Columns from One Tuple to Another

You can copy the contents of one tuple to another, as long as the tuple types are the same type (search or read), and are defined on the same table or index.

```
ib_tpl_t  src_tpl;
ib_tpl_t  dst_tpl;

Create the two tuples
…
Read or set column values of src tpl from somewhere
…

/* Copy the old contents to the new tuple. */
err = ib_tuple_copy(dst_tpl, src_tpl);
```

4.2.8. Checking if a Column Value is SQL NULL

There are two ways to test for SQL `NULL` values: one using the column metadata, and the other by accessing the value directly. The recommended method is to check the column metadata. The destination tuple is first “cleared” and then the data is copied.

```
ib_col_meta_t   ib_col_meta;

/* Get the meta data for the ith column. */
data_len = ib_col_get_meta(tpl, i, &ib_col_meta);

/* Skip system columns. */
if (ib_col_meta.type == IB_SYS) {
 continue;
/* Nothing to print. Check for NULL values. */
} else if (data_len == IB_SQL_NULL) {
 /* Do something. */
}
```

The function `ib_col_get_value` returns `NULL` if the column value is `IB_SQL_NULL`. The function `ib_col_copy_value` returns `IB_SQL_NULL` if the column value is `NULL` .

4.3. Working with Records

Remember, Embedded InnoDB works with table data at a low level, so we refer to records to distinguish them from table rows, although conceptually those notions are similar.

4.3.1. Reading a Record

To read a record from the clustered index, create a cursor handle for that table, then position the cursor on the record to read.

- Create cursor handle.
- Create a tuple into which to read the row data, using this function:

  `ib_clust_read_tuple_create()`
- Position the cursor on the record to read. For a full table scan, you first call `ib_cursor_first()` or `ib_cursor_last()`, and then `ib_cursor_next()` or `ib_cursor_prev()` for each subsequent record. To process a subset of rows, you first call `ib_cursor_moveto()`, and again `ib_cursor_next()` or `ib_cursor_prev()` for each subsequent record.
- Read the tuple by calling the function `ib_cursor_read_row()`.
- Delete the tuple handle when finished.
- Close cursor when finished.

Example code:

```
ib_err_t        err;
ib_tpl_t        tpl;

tpl = ib_clust_read_tuple_create(crsr);

/* Scan the entire table and print all rows. */
err = ib_cursor_first(crsr);

while (err == DB_SUCCESS) {
 err = ib_cursor_read_row(crsr, tpl);

 /* Possible handle locking and timeout errors too
 in multi-threaded applications. */
 if (err == DB_RECORD_NOT_FOUND || err == DB_END_OF_INDEX) {
  break;
 }

 …
 /* Do something with the tuple */
 …

 err = ib_cursor_next(crsr);

 /* Possible handle locking and timeout errors too
 in multi-threaded applications. */
 if (err == DB_RECORD_NOT_FOUND || err == DB_END_OF_INDEX) {
  break;
 }

 tpl = ib_tuple_clear(tpl);
}

ib_tuple_delete(tpl);
```

4.3.2. Inserting a Record

To insert a record, perform the following steps:

- Create a cursor handle by calling function `ib_cursor_open_table()` or `ib_cursor_open_table_using_id()`.
- Create a tuple handle by calling function `ib_clust_read_tuple_create()`. (Since the insert can affect all the columns of the table, the operation requires a tuple that contains all those columns.)
- Set the column values in the tuple by calling function `ib_col_set_value()`. Any columns you do not set, remain as SQL `NULL` values by default. For string values, you specify the numeric index of the column, a pointer to the bytes of the data value, and the length of the data. For numeric values, you pass the numeric index of the column and the value itself. The values are copied into the tuple.
- Insert the record into the table by calling function `ib_cursor_insert_row()`.
- Delete the tuple handle instance by calling function `ib_tuple_delete()`.

Example code:

```
ib_err_t        err;
ib_tpl_t        tpl;

tpl = ib_clust_read_tuple_create(crsr);

ib_col_set_value(tpl, 0, "a", strlen("a"));
ib_col_set_value(tpl, 1, "b", strlen("b"));
ib_tuple_write_i32(tpl, 2, 100);
```

```
err = ib_cursor_insert_row(crsr, tpl);

/* Check err and handle potential errors here ... */

ib_tuple_delete(tpl);
```

4.3.3. Updating a Record

To update a record, perform the following steps:

- Create a cursor handle by calling function `ib_cursor_open_table()` or `ib_cursor_open_table_using_id()`.
- Position the cursor on the record to update by calling function `ib_cursor_moveto()`, or any of the `ib_cursor_first()`, `ib_cursor_next()`, `ib_cursor_last()`, `ib_cursor_prev()` functions.
- Create a tuple handle to read the old values by calling function `ib_clust_read_tuple_create()`.
- Create a tuple handle to write the new values by calling function `ib_clust_read_tuple_create()`. (The same kind of tuple data structure is used for both reading and writing.)
- Read the values from the row into old tuple by calling function `ib_cursor_read_row()`.
- Copy the value from the old tuple to the new tuple by calling function `ib_tuple_copy()`. (You must pass both the old and new tuples as parameters when you call the function to do the update.)
- Set the column values in the new tuple by calling function `ib_col_set_value()` one or more times.
- Write the updated record back into the table by calling function `ib_table_update_row()`.
- Delete the old and new tuple handles by calling function `ib_tuple_delete()` for each one.

Example code, from the sample program `ib_test1.c`:

```
char*           c1;
int             c1_len;
int             c3_len;
ib_col_meta_t   col_meta;
ib_tpl_t        old_tpl = NULL;
ib_tpl_t        new_tpl = NULL;

/* Create the tuple instance that we will use to update the
table. old_tpl is used for reading the existing row and
new_tpl will contain the update row data. */

old_tpl = ib_clust_read_tuple_create(crsr);
new_tpl = ib_clust_read_tuple_create(crsr);

ib_cursor_read_row(crsr, old_tpl);

/* Get the first column value. c1 will be NULL if
the column value is SQL_NULL. */
c1 = ib_col_get_value(old_tpl, 0);
```

```
    c1_len = ib_col_get_meta(old_tpl, 0, &col_meta);

    /* Copy the old contents to the new tuple. */
    ib_tuple_copy(new_tpl, old_tpl);

    /* Set the new value of column c3 in the new tuple. */
    ib_tuple_write_i32(new_tpl, 2, 10);

    err = ib_cursor_update_row(crsr, old_tpl, new_tpl);
    /* Check err status etc. */

    ib_tuple_delete(old_tpl);
    ib_tuple_delete(new_tpl);
```

4.3.4. Deleting a Record

To delete a record, perform the following steps:

- Obtain a cursor handle to a table.
- Position the cursor on the record to delete.
- Delete the row by calling function `ib_table_delete_row()`.

Below is an example of searching for the record to delete, from the sample program `ib_test1.c`:

```
    ib_err_t        err;
    int             res = ~0;
    ib_tpl_t        key = NULL;

    /* Create a tuple for searching an index. */
    key_tpl = ib_sec_search_tuple_create(crsr);

    /* Set the value to delete. */
    ib_col_set_value(key, 0, "b", strlen("b"));
    ib_col_set_value(key, 1, "z", strlen("z"));

    /* Search for the key using the clustered index (PK) */
    err = ib_cursor_moveto(crsr, key, IB_CUR_GE, &res);

    /* Must be positioned on the record to delete, since
    we've specified an exact match. */
    /* res will tell us whether there was an exact
    match found or not, res == 0 means cursor positioned
    on a record that matches exactly. */

    ib_tuple_delete(key);

    /* InnoDB handles the updating of all secondary indexes. */
    if (res == 0) {
     err = ib_cursor_delete_row(crsr);
    }
```

```
/* Check err status etc. */
```

4.3.5. Emulating a WHERE Clause

To emulate a `WHERE` clause, which lets you operate on multiple rows with a single `INSERT`, `UPDATE`, or `DELETE` statement, use the techniques from Section 4.2.4, “Scanning and Positioning” to loop through all the relevant records from your result set, and perform the appropriate operation on each one as demonstrated in the preceding sections.

Chapter 5. Managing Transactions with Embedded InnoDB

All DDL and DML operations must be covered by transactions.

This section refers to transactions, isolation levels, savepoints, and other database terminology that may have special nuances within Embedded InnoDB; familiarize yourself with these nuances first, by reading:

- Chapter 1, *Concepts and Architecture for Embedded InnoDB*.
- Embedded InnoDB Glossary.

For instructions for performing DDL and DML operations, see Chapter 3, *Performing CREATE and DROP Operations (DDL) with Embedded InnoDB* and Chapter 4, *Performing SELECT, INSERT, UPDATE, and DELETE (DML) Operations with Embedded InnoDB*.

Throughout this section, we refer to functions and constants that are defined later, in Chapter 6, *C API Reference for Embedded InnoDB*, and especially in Section 6.7.3, “API Functions - Transactions”.

5.1. Creating a Transaction Handle

You first create a transaction handle of type `ib_trx_t` . You do this by calling the `ib_trx_begin()` API function and specifying the transaction isolation level for that transaction.

If the transaction is not started, or has already been committed or rolled back, using the transaction handle for other operations produces undefined results (which could include a crash).

```
ib_trx_t  ib_trx;

ib_trx = ib_trx_begin(IB_REPEATABLE_READ);

...
/* Do some operations on tables */
...

ib_trx_commit(ib_trx);
```

5.2. Committing or Rolling Back a Transaction

Committing the transaction with the `ib_trx_commit()` function makes permanent any changes to tables, and releases any locks you have acquired. To roll back the changes instead, call the `ib_trx_rollback()` function, which undoes any changes made to tables, and also releases all locks.

Although you must wrap DDL operations such as `CREATE` and `DROP` inside a transaction also, the changes that these operations make to the system tables cannot be rolled back.

5.3. Setting the Transaction Isolation Level

Transaction isolation levels can only be set when a new transaction is started with `ib_trx_begin()`.

`IB_READ_UNCOMMITTED`	Dirty read: non-locking `SELECT` operations are performed so that we do not look at a possible earlier version of a record; thus they are not “consistent” reads under this isolation level; otherwise like level `IB_REPEATABLE_READ`.
`IB_READ_COMMITTED`	Somewhat Oracle-like isolation, where a transaction can see data committed by other transactions since the start of the current transaction. `UPDATE` and `DELETE` operations with `WHERE` clauses specifying a range of values, prevent other transactions from affecting rows within the same range. Other transactions are even prevented from inserting new rows with values that would fall within the same range. `SELECT ... FOR UPDATE` and `... LOCK IN SHARE MODE` use a conservative locking strategy that does not prevent other transactions from inserting rows with particular values. Each consistent read reads its own snapshot.
`IB_REPEATABLE_READ`	All consistent reads in the same transaction read the same snapshot; full next-key locking is used in locking reads to block insertions into gaps.
`IB_SERIALIZABLE`	All plain `SELECT` operations are converted to `LOCK IN SHARE MODE` reads.

5.4. Nested Transactions

Nested transactions can be emulated using savepoints. By dividing up the work into separate units, you can take savepoints of the state of the current transaction at various stages that you choose. Savepoints cannot be committed, but can be rolled back. Rolling back to a savepoint also rolls back any savepoints taken after that savepoint. Savepoints have tags (or names) and can be referred to by these tags. InnoDB simply treats the tag as a sequence of raw bytes and attaches no other special meaning to it. Savepoints are deleted in a transaction commit or rollback.

5.4.1. Taking Savepoints

To take a savepoint, you must already have an active transaction. If there is already a savepoint of the same name, this call erases that old savepoint and replaces it with a new one.

```
ib_trx_t  ib_trx;

ib_trx = ib_trx_begin(IB_REPEATABLE_READ);
...
/* Make some changes. */
ib_savepoint_take(ib_trx, savepoint_tag, savepoint_tag_len);
```

5.4.2. Deleting a Savepoint

Releases only the named savepoint. Any savepoints that were set after this savepoint are left as-is. This does not undo any changes but only deletes the internal housekeeping data of the savepoint. If the savepoint does not exist, the function returns `DB_NO_SAVEPOINT`.

```
 ib_savepoint_release(ib_trx, savepoint_tag, savepoint_tag_len);
```

5.4.3. Rollback to Savepoint

You can either roll back to a specific named savepoint to undo the changes made after a certain point, or roll back all the savepoints to undo all changes made in that transaction. The savepoints are deleted once they are rolled back. Modifications after the savepoint are undone, but InnoDB does **not** release the corresponding locks that are stored in memory. If a lock is “implicit”, that is, a new inserted row holds a lock where the lock information is carried by the transaction ID stored in the row, these locks are naturally released in the rollback. Any savepoints that were set after this savepoint are deleted. If name equals NULL (that is, a C null pointer), all the savepoints are rolled back.

For example, the following call rolls back all the savepoints and deletes them.

```
ib_savepoint_rollback(ib_trx, NULL, 0);
```

and this call rolls back the named savepoint and all the savepoints taken after the named savepoint and deletes them.

```
 ib_savepoint_rollback(ib_trx, savepoint_tag, savepoint_tag_len);
```

Chapter 6. C API Reference for Embedded InnoDB

6.1. Constants

The following constants are used by the Embedded InnoDB API.

Constant name	Description
`IB_TRUE`	Boolean true.
`IB_FALSE`	Boolean false.
`IB_SQL_NULL`	Special data length value that signifies a NULL column value.
`IB_N_SYS_COLS`	The number of system columns in a key or row. These are columns that are used internally by InnoDB, such as `DB_TRX_ID`.

6.2. Data Types

The data structures and type definitions used by Embedded InnoDB.

6.2.1. Base Types

Type Name	Description
`ib_err_t`	The type used for all the error and status codes.
`ib_byte_t`	The type of a byte on the target platform.
`ib_id_t`	Table and index IDs.
`ib_bool_t`	Boolean type; can be `IB_TRUE` or `IB_FALSE`.
`ib_ulint_t`	The type of an unsigned long integer on the target platform.
`ib_opaque_t`	Opaque type, whose details are hidden.
`ib_charset_t`	Not currently used.
`ib_i8_t`	8-bit signed integer.
`ib_u8_t`	8-bit unsigned integer.
`ib_i16_t`	16-bit signed integer.
`ib_u16_t`	16-bit unsigned integer.
`ib_i32_t`	32-bit signed integer.
`ib_u32_t`	32-bit unsigned integer.
`ib_i64_t`	64-bit signed integer.
`ib_u64_t`	64-bit unsigned integer.

6.2.2. Configuration Variable Types

InnoDB supports various configuration variables. The types of these variables are listed below:

Type Name	Description
`IB_CFG_IBOOL`	The configuration parameter is of type `ibool`.
`IB_CFG_ULINT`	The configuration parameter is of type `ulint`.
`IB_CFG_ULONG`	The configuration parameter is of type `ulong`.
`IB_CFG_TEXT`	The configuration parameter is of type `char*`.
`IB_CFG_CB`	The configuration parameter is a callback parameter.

6.2.3. Table Format types

InnoDB supports various table format types. You must specify the appropriate type when creating the table. All the supported types are listed below:

Table Type	Description
`IB_TBL_REDUNDANT`	Redundant row format.
`IB_TBL_COMPACT`	Compact row format.
`IB_TBL_DYNAMIC`	Dynamic row format. BLOB prefixes are not stored in the clustered index.
`IB_TBL_COMPRESSED`	Compressed row format. Similar to dynamic format, but with pages compressed.

6.2.4. Column Types

The column type `ib_col_type_t` can have the following possible values. Typically, you read columns one by one out of a tuple, and use a `switch` construct to decide how to process each column based on its type.

Type Name	Description
`IB_VARCHAR`	Character varying length.
`IB_CHAR`	Fixed-length character string.
`IB_BINARY`	Fixed length binary, similar to IB_CHAR but the column is not padded to the right.
`IB_BINARY`	Variable length binary.
`IB_VARCHAR_ANYCHARSET`	Variable-length column using any character set.
`IB_CHAR_ANYCHARSET`	Fixed-length column using any character set.
`IB_BLOB`	Binary large object, or a TEXT type.
`IB_INT`	Integer: can be any size 1 to 8 bytes.
`IB_SYS`	System column (read-only). For example, InnoDB adds some system columns to each clustered index, for housekeeping purposes. When stepping through the columns of a result set, ignore any column with this type.
`IB_FLOAT`	C `float`.
`IB_DOUBLE`	C `double`.
`IB_DECIMAL`	Decimal stored as an ASCII string.

6.2.4.1. Mapping MySQL Types to Embedded InnoDB

Here is the reverse mapping, showing how to convert from any MySQL column type to the equivalent `ib_col_type_t` value. In particular, note:

- To create a column with an unsigned type, you pass the `IB_COL_UNSIGNED` value as a separate parameter from the type itself when setting up the column metadata.
- Only the `Latin1` character set is supported. UTF-8 encoding is not currently supported. (Of course, you can store any bytes you want in the table; this restriction applies to comparison operations that happen internally in the Embedded InnoDB APIs.) The `Latin1` character set maps to the Embedded InnoDB types `IB_CHAR` and `IB_VARCHAR`, while other character sets map to the types `IB_CHAR_ANYCHARSET` and `IB_VARCHAR_ANYCHARSET`.
- Some MySQL types map to different Embedded InnoDB types depending on the precision of the type.
- Some different MySQL types map to the same Embedded InnoDB type. For example, the MySQL integer types all map to `IB_INT`, and when setting up the column metadata you pass different values for the length parameter to account for the different MySQL types.
- Some MySQL attributes, such as `ZEROFILL`, are for presentation only and are not represented or significant in the Embedded InnoDB metadata.
- Certain attributes that are not recognized or supported by Embedded InnoDB are passed to your code in the `IB_CUSTOM1`, `2`, and `3` fields if you open a table that was created with MySQL. (If you are interacting with MySQL data at that level, you should already be an expert in the MySQL source code.)

MySQL Type	**Value for ib_col_type_t**
`BIGINT`	`IB_INT`
`BINARY(M)`	`IB_CHAR` or `IB_CHAR_ANYCHARSET`
`BIT`	`IB_CHAR` or `IB_CHAR_ANYCHARSET`
`BLOB`	`IB_BLOB`
`BOOL, BOOLEAN`	`IB_INT`
`CHAR`	`IB_CHAR` or `IB_CHAR_ANYCHARSET`
`CHAR BYTE` (an alias for the BINARY data type)	`IB_CHAR` or `IB_CHAR_ANYCHARSET`
`DATE`	`IB_INT`
`DATETIME`	`IB_INT`
`DECIMAL`	`IB_VARCHAR`
`DEC`	`IB_VARCHAR`
`DOUBLE PRECISION, REAL`	`IB_DOUBLE`
`DOUBLE`	`IB_DOUBLE`
`ENUM('value1','value2',...)`	`IB_VARCHAR` or `IB_VARCHAR_ANYCHARSET`
`FIXED`	`IB_FLOAT`
`FLOAT(p)`	`IB_FLOAT` or `IB_DOUBLE`
`FLOAT`	`IB_FLOAT` or `IB_DOUBLE`
`INTEGER`	`IB_INT`
`INT`	`IB_INT`

MySQL Type	Value for ib_col_type_t
LONGBLOB	IB_BLOB
LONGTEXT	IB_VARCHAR or IB_VARCHAR_ANYCHARSET
MEDIUMBLOB	IB_BLOB
MEDIUMINT	IB_INT
MEDIUMTEXT	IB_VARCHAR or IB_VARCHAR_ANYCHARSET
NUMERIC	IB_FLOAT
SERIAL	IB_INT (unsigned)
SET('value1','value2',...)	IB_VARCHAR or IB_VARCHAR_ANYCHARSET
SMALLINT	IB_INT
TEXT	IB_VARCHAR or IB_VARCHAR_ANYCHARSET
TIME	IB_INT
TIMESTAMP	IB_INT
TINYBLOB	IB_BLOB
TINYINT	IB_INT
TINYTEXT	IB_VARCHAR or IB_VARCHAR_ANYCHARSET
VARBINARY	IB_VARCHAR or IB_VARCHAR_ANYCHARSET
VARCHAR	IB_VARCHAR or IB_VARCHAR_ANYCHARSET
YEAR	IB_INT

6.2.5. Column Attributes

The column type `ib_col_attr_t` can have the following attributes:

Attribute Type	Description
IB_COL_NONE	No special attributes.
IB_COL_NOT_NULL	Column cannot be NULL.
IB_COL_UNSIGNED	Column is `IB_INT` and unsigned.
IB_COL_CUSTOM1	Custom precision type. You can use this and the following custom fields as bit flags to store your own information about column attributes. Embedded InnoDB does not use these fields itself; InnoDB ignores any values they contain.
IB_COL_CUSTOM2	Custom precision type.
IB_COL_CUSTOM3	Custom precision type.

6.2.6. Column Metadata

Every column definition is represented by the data structure `ib_col_meta_t`:

```
typedef struct ib_col_meta_t {
  ib_col_type_t   type;        /* Type of the column */
  ib_col_attr_t   attr;        /* Column attributes */
  ib_ulint_t      type_len;    /* Length of type in bytes, can
                               be IB_SQL_NULL */
  ib_u16_t        client_type; /* 16 bits of data relevant only to the
                               client. InnoDB ignores this attribute */
  ib_charset_t*   charset;     /* Column charset */
} ib_col_meta_t;
```

6.3. Modes

6.3.1. Shutdown Modes

There are several modes for shutdown, that control what actions InnoDB should perform when a shutdown of the storage engine is requested.

Shutdown Mode	**Description**
`IB_SHUTDOWN_NORMAL`	Normal shutdown. Do insert buffer merge and purge before complete shutdown. Depending on the state of the server, this can take several minutes.
`IB_SHUTDOWN_NO_IBUFMERGE_PURGE`	Do not do a purge and index buffer merge at shutdown. Note that InnoDB will do the insert buffer merge and purge on the next start-up.
`IB_SHUTDOWN_NO_BUFPOOL_FLUSH`	Same as `IB_SHUTDOWN_NO_IBUFMERGE_PURGE`, and in addition do not even flush the buffer pool to data files. No committed transactions are lost. Note that this is the equivalent of crashing the server.

6.3.2. Lock Modes

InnoDB supports the following lock modes. You specify the degree to which you need exclusive access to a resource (for example, while writing) or shared access (for example, while reading). Embedded InnoDB controls access to the appropriate tables, rows, and so on based on the locks you acquire, and returns error codes if a deadlock or timeout condition occurs. (Thus, it is important to always check return codes.)

Some lock modes can only be used to lock tables; for example, the lock modes `IB_LOCK_IS` and `IB_LOCK_IX`.

Table type	**Description**
`IB_LOCK_IS`	Intention shared (can only be used for locking tables).
`IB_LOCK_IX`	Intention exclusive (can only be used for locking tables).
`IB_LOCK_S`	Shared.
`IB_LOCK_X`	Exclusive.

6.3.3. Cursor Search Modes

The cursor search modes represent conditional tests that you would specify in a SQL `WHERE` clause, for example to position the cursor on a row that is greater or less than some value. The search mode setting is used by `ib_cursor_moveto()` to position the cursor when searching for a key.

Embedded InnoDB currently supports the following search modes:

Search Mode	Description
`IB_CUR_G`	This search mode lets you implement the equivalent of a SQL comparison `WHERE` *`column`* `>` *`value`*. The cursor is positioned on a record that is greater than the search key. If no record has a value that is greater than the search key, the cursor is positioned on the last record in the table. You can also use this search mode to search in descending order. Position the cursor using IB_CUR_G, then call `ib_cursor_prev()` to move through the records in descending order.
`IB_CUR_GE`	This search mode lets you implement the equivalent of a SQL comparison `WHERE` *`column`* `>=` *`value`*. If the search key is found, the cursor is positioned on the first row that is equal to the search key. If the search key is not found, the cursor is positioned on the first record that is greater than the search key. Because there is no search mode that requires an exact match, use this search mode also to implement the equivalent of a SQL comparison `WHERE` *`column`* `=` *`value`*. Position the cursor using `IB_CUR_GE`, then keep reading and processing records until the key value is different from the search key.
`IB_CUR_L`	This search mode lets you implement the equivalent of a SQL comparison `WHERE` *`column`* `<` *`value`*. The cursor is positioned on a record that is less than the search key. If no record has a value that is less than the search key, the cursor is positioned on the first record in the table.
`IB_CUR_LE`	This search mode lets you implement the equivalent of a SQL comparison `WHERE` *`column`* `<=` *`value`*. If the key is found, the cursor is positioned on the first row that is equal to the search key. If the search key is not found, the cursor is positioned on the first record that is less than the search key. If no record has a value that is less than the search key, the cursor is positioned on the first record in the table.

6.3.4. Cursor Match Modes

These cursor match modes affect the state of the cursor. The match modes control these things: the positioning of the cursor; whether or not its state needs to be made persistent; whether the next N records need to be fetched into the row cache, on the assumption that the program will do cursor next/prev operations. Storing the cursor state is an expensive operation and should be minimized and for that reason. Embedded InnoDB fetches the next N records to avoid unnecessary persistent cursor operations. Embedded InnoDB currently supports the following match modes:

Match Mode	Description
`IB_CLOSEST_MATCH`	Closest match possible. Note that if you do a search with a full key value from a unique index, Embedded InnoDB does not store the cursor position and the program must not call `ib_cursor_next()` or `ib_cursor_prev()` on the cursor.
`IB_EXACT_MATCH`	Search using a complete value. If an exact match is not found, InnoDB returns `DB_RECORD_NOT_FOUND`. `IB_EXACT_MATCH` does not store the cursor state and does not fetch the next N records to the row cache, making it ideal for single-record lookups.
`IB_EXACT_PREFIX`	Search using a key prefix which must match to rows: the prefix may contain an incomplete field (the last field in prefix may be just a prefix of a fixed length column).

6.4. Transactions

6.4.1. Transaction Isolation Levels

You can specify the following values when starting a transaction. Each value represents an isolation level. The values are listed here in descending order of frequency of use in typical applications. (In fact, the only design decision needed for most applications is whether to use `IB_REPEATABLE_READ`, which is the default for regular MySQL and InnoDB applications, or switch to `IB_READ_COMMITTED`.

Isolation Level	**Description**
`IB_REPEATABLE_READ`	The repeatable read isolation level. All consistent reads in the same transaction read the same snapshot. Full next-key locking is used in locking reads to block insertions into gaps. This is the default isolation level for the InnoDB storage engine when running under the MySQL database.
`IB_READ_COMMITTED`	The read committed isolation level, a somewhat Oracle-like mode. It represents a tradeoff between concurrency and reliability that leans more towards concurrency; a transaction can see data that was changed by another transaction and committed, rather than strictly seeing the data as it existed at the transaction start time. One difference from Oracle: in range `UPDATE` and `DELETE` operations, the engine blocks the insertion of “phantom rows” with next-key locks. The API equivalents of `SELECT FOR UPDATE` and `LOCK IN SHARE MODE` only lock the index records, **not** the gaps before them, and thus allow free inserting. Each consistent read gets data from its own snapshot.
`IB_READ_UNCOMMITTED`	The read uncommitted isolation level, which allows dirty reads. `SELECT` operations do not acquire locks, but also do not reconstruct the original data for rows that are being changed by other transactions, even changes that have not been committed yet. Otherwise, this isolation level is like level `IB_REPEATABLE_READ`. Because this mode leans so heavily towards concurrency at the expense of repeatable results, it is rarely used.
`IB_SERIALIZABLE`	The serializable read isolation level. All plain `SELECT` operations are converted to `LOCK IN SHARE MODE` reads. Because this mode leans so heavily towards deterministic results at the expense of concurrency, it is rarely used.

6.4.2. Transaction States

A transaction can be in one of the following states, which you can check by using the `ib_trx_state()` function. The InnoDB deadlock monitor can roll back a transaction, and you should be prepared for this, especially where there is high contention.

Transaction State	**Description**
`IB_TRX_NOT_STARTED`	The transaction has not been started yet.
`IB_TRX_ACTIVE`	The transaction is currently active and needs to be either committed or rolled back.
`IB_TRX_COMMITTED_IN_MEMORY`	Not committed to disk yet.
`IB_TRX_PREPARED`	Support for 2-phase commit “XA” protocol.

6.5. Handles (opaque types)

These handle types are returned by various create and open calls. You do not manipulate their contents directly. You just pass the returned handles as parameters to subsequent function calls.

Handle Type	Description
`ib_tpl_t`	Handle to a tuple, used to read and write tuples from a table.
`ib_trx_t`	Transaction handle.
`ib_crsrl_t`	Handle to an InnoDB cursor.
`ib_sch_t`	Handle to a table schema instance, used to create tables.
`ib_idx_sch_t`	Handle to an index schema instance, used to create indexes on a table.

6.6. Error and Status Codes

The API functions can return the following status/error codes.

Status/Error Code	Description
DB_SUCCESS	All API functions that complete successfully return this error code.
DB_ERROR	Generic failure code.
DB_OUT_OF_MEMORY	InnoDB cannot allocate memory.
DB_OUT_OF_FILE_SPACE	No more disk space on the filesystem.
DB_LOCK_WAIT	Another transaction has a request on the table in an incompatible mode: this transaction may have to wait.
DB_DEADLOCK	A lock request will result in a deadlock error. Deadlock has already happened and the transaction has been rolled back by the deadlock monitor. The transaction handle should be freed with `ib_trx_release()`.
DB_DUPLICATE_KEY	An insert or update operation violates a unique constraint.
DB_QUE_THR_SUSPENDED	Lock request was ignored; do not enqueue a lock request if the query thread should be stopped anyway.
DB_MISSING_HISTORY	Required history data has been deleted, due to lack of space in rollback segment.
DB_TABLE_NOT_FOUND	Table could not be found in the data dictionary.
DB_MUST_GET_MORE_FILE_SPACE	The database must be stopped and restarted with more file space.
DB_TABLE_IS_BEING_USED	Attempt to create a table and the table already exists in the data dictionary.
DB_TOO_BIG_RECORD	A record being inserted or updated would not fit on a compressed page, or it would become bigger than 1/2 the free space in an uncompressed page frame. This error can only happen when inserting or updating table data; InnoDB handle updates to index pages automatically.
DB_LOCK_WAIT_TIMEOUT	Lock wait lasted too long. The transaction has been rolled back by the dead lock monitor. Free the transaction handle by calling the function `ib_trx_release()`.
DB_NO_REFERENCED_ROW	Referenced key value not found for a foreign key in an insert or update of a row. (Not currently implemented.)

Status/Error Code	**Description**
DB_ROW_IS_REFERENCED	Cannot delete or update a row because it contains a key value that is referenced. (Not currently implemented.)
DB_CANNOT_ADD_CONSTRAINT	Adding a foreign key constraint to a table failed. (Not currently implemented.)
DB_CORRUPTION	Data structure corruption noticed.
DB_COL_APPEARS_TWICE_IN_INDEX	InnoDB cannot handle an index where the same column appears twice.
DB_CANNOT_DROP_CONSTRAINT	Dropping a foreign key constraint from a table failed. (Not currently implemented.)
DB_NO_SAVEPOINT	No savepoint exists with the given name.
DB_TABLESPACE_ALREADY_EXISTS	Cannot create a new single-table tablespace because a file of the same name already exists.
DB_TABLESPACE_DELETED	Tablespace does not exist or is being dropped right now.
DB_LOCK_TABLE_FULL	Lock structs have exhausted the buffer pool. (For big transactions, InnoDB stores the lock structs in the buffer pool.)
DB_FOREIGN_DUPLICATE_KEY	Foreign key constraints activated, but the operation would lead to a duplicate key in some table. (Not currently implemented.)
DB_TOO_MANY_CONCURRENT_TRXS	When InnoDB runs out of the preconfigured undo slots; this can only happen when there are too many concurrent transactions.
DB_UNSUPPORTED	When InnoDB sees any construct or feature that it cannot recognize or work with; for example, a table created by a later version of the engine.
DB_PRIMARY_KEY_IS_NULL	A column in the primary key was found to be `NULL`.

6.6.1. Warnings

Warning Code	**Description**
DB_RECORD_NOT_FOUND	A record cannot be found in a table or index during a search.
DB_END_OF_INDEX	A record cannot be found in a table or index during a search, and the search ends up at the end of the index.
DB_SCHEMA_ERROR	Error encountered when validating a table or index schema.
DB_DATA_MISMATCH	Attempt to set the value of a column in a tuple from a different type. For example, trying to write an `INT` value to a `CHAR` column.
DB_SCHEMA_NOT_LOCKED	An API function expects the schema to be locked in exclusive mode, but it is not.
DB_NOT_FOUND	Generic error code for “Not found” type of errors.
DB_READONLY	Generic error code for “Read only” type of errors.
DB_INVALID_INPUT	Generic error code for “Invalid input” type of errors.

6.7. API Function Reference

The Embedded InnoDB API is divided into several groups:

- Section 6.7.1, “API Functions - Startup/Shutdown”
- Section 6.7.2, “API Functions - Cursors”

- Section 6.7.3, “API Functions - Transactions”
- Section 6.7.4, “API Functions - Configuration”
- Section 6.7.5, “API Functions - DDL”
- Section 6.7.6, “API Functions - DDL and DML Support”
- Section 6.7.7, “API Functions - Tuples”

To see these APIs in action, look in the sample programs listed in Appendix B, *Code Examples Supplied with Embedded InnoDB*.

6.7.1. API Functions - Startup/Shutdown

These APIs are typically the first and last functions that you call within a Embedded InnoDB program. (You might also call some of the functions in Section 6.7.4, “API Functions - Configuration” prior to starting the database engine.) To see these APIs in action, look in the sample programs listed in Appendix B, *Code Examples Supplied with Embedded InnoDB*.

6.7.1.1. ib_api_version

Synopsis

```
    ib_u64_t ib_api_version(void);
```

This function returns a `ib_u64_t` value representing the version number of the API. It is primarily useful at the beginning of your program, before calling any other Embedded InnoDB function, to choose a version-specific code path. The version number format is in the lower three byes and can be extracted as follows:

```
int release, revision, age;

release = version >> 32;
revision = (version >> 16) & 0xffff;
age = version & 0xffff
```

6.7.1.2. ib_init

Synopsis

```
    ib_err_t ib_init(void);
```

This function must be called before calling any other Embedded InnoDB function except ib_api_version() and the configuration functions. It initializes the memory sub-system.

6.7.1.3. ib_startup

Synopsis

```
    ib_err_t ib_startup(const char* format);
```

Starts up the engine, creates the system tables if they do not exist, and does recovery if the system was not shut down cleanly. This function starts up the background I/O threads and sets up the buffer pool and other memory structures. The only functions that you can call before startup are these configuration functions: ib_cfg_set*(), ib_cfg_get*() and ib_cfg_var_get_type().

If you do not set up the configuration parameters, InnoDB uses pre-configured default values. In particular, it uses the current working directory to create the system tablespace and the redo log files.

Parameters:	`in: format`	is the file format name
Returns:	DB_SUCCESS or error code	

6.7.1.4. ib_shutdown

Synopsis

```
ib_err_t ib_shutdown(ib_shutdown_t flag);
```

Shuts down the InnoDB instance and frees up all the memory. There must not be any active transactions. It also shuts down the background I/O threads. InnoDB supports several modes of shutdown.

Parameters:	`in: flag`	is the shutdown mode
Returns:	DB_SUCCESS or error code	

6.7.2. API Functions - Cursors

To see these APIs in action, look in the sample programs listed in Appendix B, *Code Examples Supplied with Embedded InnoDB.*

6.7.2.1. ib_cursor_open_table

Synopsis

```
ib_err_t ib_cursor_open_table(const char* name, ib_trx_t ib_trx,
                              ib_crsr_t* ib_crsr);
```

Open a table using the table name. Use this function to create a cursor over the table. The transaction instance can be `NULL`; in this case, you must call ib_cursor_attach_trx() to attach the cursor to an active transaction before using the cursor.

Parameters:	`in: name`	is the table name to open
	`in: ib_trx`	An active transaction; can be `NULL`
	`out: ib_crsr`	is the newly created cursor instance
Returns:	DB_SUCCESS or error code	

6.7.2.2. ib_cursor_open_table_using_id

Synopsis

```
ib_err_t ib_cursor_open_table_using_id(
         ib_id_t        table_id,
         ib_trx_t       ib_trx,
         ib_crsr_t*     ib_crsr);
```

Open a table using the table numeric ID. Use this function to create a cursor over the table. The transaction instance can be `NULL`; in this case, you must call ib_cursor_attach_trx()

Parameters:	`in: table_id`	is the table id to open
	`in: ib_trx`	an active transaction, can be `NULL`
	`out: ib_crsr`	is the newly created cursor instance
Returns:	DB_SUCCESS or error code	

6.7.2.3. ib_cursor_open_index_using_name

Synopsis

```
ib_err_t ib_cursor_open_index_using_name(
         ib_crsr_t       ib_open_crsr,
         const char*     index_name,
         ib_crsr_t*      ib_crsr);
```

Open an index using the secondary index name. Use this function to create a cursor over a secondary index. `ib_open_crsr` should be an open table cursor.

Parameters:	`in: ib_open_crsr`	is an open table cursor
	`in: index_name`	name of the index to use
	`out: ib_crsr`	is the newly created cursor instance
Returns:	DB_SUCCESS or error code	

6.7.2.4. ib_cursor_open_index_using_id

Synopsis

```
ib_err_t ib_cursor_open_index_using_id(
         ib_id_t         index_id,
         ib_trx_t        ib_trx,
         ib_crsr_t*      ib_crsr);
```

Open an index using the index numeric ID. Use this function to create a cursor over the index. The transaction instance can be `NULL`; in this case, you must call ib_cursor_attach_trx()

Parameters:	`in: index_id`	the index id is a 64 bit number and should be encoded as follows: [table id:32bits | index id:32 bits]
	`in: ib_trx`	transaction instance, can be `NULL`
	`out: ib_crsr`	is the newly created cursor instance
Returns:	DB_SUCCESS or error code	

6.7.2.5. ib_cursor_first

Synopsis

```
ib_err_t ib_cursor_first(ib_crsr_t ib_crsr);
```

Position the cursor on the first record in the index or table.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor
Returns:	DB_SUCCESS or error code	

6.7.2.6. ib_cursor_next

Synopsis

```
ib_err_t ib_cursor_next(ib_crsr_t ib_crsr);
```

Move the cursor to the next user record.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor
Returns:	DB_SUCCESS or error code	

6.7.2.7. ib_cursor_last

Synopsis

```
ib_err_t ib_cursor_last(ib_crsr_t ib_crsr);
```

Position the cursor on the last table or secondary index record.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor
Returns:	DB_SUCCESS or error code	

6.7.2.8. ib_cursor_prev

Synopsis

```
ib_err_t ib_cursor_prev(ib_crsr_t ib_crsr);
```

Move the cursor to the previous table or secondary index record.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor
Returns:	DB_SUCCESS or error code	

6.7.2.9. ib_cursor_attach_trx

Synopsis

```
ib_err_t ib_cursor_attach_trx(ib_crsr_t ib_crsr, ib_trx_t ib_trx);
```

Set the current active transaction and attaches the cursor to the transaction. It is an error to attach a cursor to a transaction when the cursor is already attached to another transaction.

Parameters:	`in: ib_crsr`	An open table or secondary index cursor that is not attached to any transaction.
	`in: ib_trx`	an active transaction instance, cannot be `NULL`
Returns:	DB_SUCCESS or error code	

6.7.2.10. ib_cursor_set_cluster_access

Synopsis

```
ib_err_t ib_cursor_set_cluster_access(ib_crsr_t ib_crsr);
```

Prepare the cursor so it can access the clustered index via the secondary index. The InnoDB engine acquires the necessary locks. Use this function when you are using the secondary index for positioning or lookup, and want to subsequently access fields that are not included in that secondary index.

Parameters:	`in: ib_crsr`	is an open secondary index cursor

6.7.2.11. ib_cursor_is_positioned

Synopsis

```
ib_err_t ib_cursor_is_positioned(ib_crsr_t ib_crsr);
```

Check if an InnoDB table or secondary index cursor is positioned on a record in the table.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor
Returns:	IB_TRUE or IB_FALSE	

6.7.2.12. ib_cursor_set_lock_mode

Synopsis

```
ib_err_t ib_cursor_set_lock_mode(ib_crsr_t ib_crsr, ib_lck_mode_t ib_lck_mode);
```

Set the lock mode of the cursor. The lock mode can be one of IB_LOCK_X or IB_LOCK_S. You must call this function when you are doing the equivalent of `SELECT FOR UPDATE`, even if the application is single-threaded.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor
	`in: ib_lck_mode`	one of IB_LOCK_S or IB_LOCK_X
Returns:	DB_SUCCESS or error code	

6.7.2.13. ib_cursor_set_match_mode

Synopsis

```
void ib_cursor_set_match_mode(ib_crsr_t ib_crsr, ib_match_mode_t match_mode);
```

Set the match mode for the cursor, used by ib_cursor_moveto. The values are explained in Section 6.3.4, “Cursor Match Modes”.

Parameters:	`in: ib_crsr`	is an open table cursor
	`in: match_mode`	the match mode to set

6.7.2.14. ib_cursor_set_simple_select

Synopsis

```
void ib_cursor_simple_select(ib_crsr_t ib_crsr);
```

Set the simple select flag, a hint that means that your code does not intend to modify any of the records that the cursor traverses. (That is, the cursor represents a simple `SELECT` statement rather than `SELECT ... FOR UPDATE`.)

When this flag is set and the transaction isolation level is either IB_TRX_READ_UNCOMMITTED or IB_TRX_READ_COMMITTED and the cursor lock mode is set to something other than IB_LOCK_NONE, Embedded InnoDB does not set gap locks.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor

6.7.2.15. ib_cursor_stmt_begin

Synopsis

```
void ib_cursor_stmt_begin(ib_crsr_t ib_crsr);
```

Set the statement begin flag. When you use the READ COMMITTED isolation level, each “statement” has its own snapshot to perform consistent reads. Since Embedded InnoDB programs do not use actual SQL statements, this function lets you specify the point to generate each snapshot.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor

6.7.2.16. ib_cursor_lock

Synopsis

```
ib_err_t ib_cursor_lock(ib_crsr_t ib_crsr, ib_lck_mode_t ib_lck_mode);
```

Lock the table that the cursor is iterating over. This function locks the table even if the cursor is a secondary index cursor.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor
	`in: ib_lck_mode`	one of IB_LOCK_IS or IB_LOCK_IX
Returns:	DB_SUCCESS or error code	

6.7.2.17. ib_cursor_reset

Synopsis

```
ib_err_t ib_cursor_reset(ib_crsr_t ib_crsr);
```

Reset the cursor. The cursor state is reset, and memory allocated for old versions of records is reclaimed. Memory for internal data structures is freed. The cursor is no longer associated with any transaction after successfully returning from this function.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor
Returns:	DB_SUCCESS or error code	

6.7.2.18. ib_cursor_close

Synopsis

```
ib_err_t ib_cursor_close(ib_crsr_t ib_crsr);
```

Close the cursor. The cursor handle is invalid after calling this function.

Parameters:	`in: ib_crsr`	is an open table cursor
Returns:	DB_SUCCESS or error code	

6.7.2.19. ib_cursor_insert_row

Synopsis

```
ib_err_t ib_cursor_insert_row(ib_crsr_t ib_crsr, const ib_tpl_t ib_tpl);
```

Insert a row into a table.

Parameters:	`in: ib_crsr`	is an open table cursor
	`in: ib_tpl`	tuple to insert into a table
Returns:	DB_SUCCESS or error code	

6.7.2.20. ib_cursor_update_row

Synopsis

```
ib_err_t ib_cursor_update_row(
        ib_crsr_t       ib_crsr,
        const ib_tpl_t  ib_old_tpl,
        const ib_tpl_t  ib_new_tpl);
```

Update a row in the table. The cursor should be positioned on the record to be updated. You must fill in all columns for the tuples representing both the new and old rows. The Embedded InnoDB engine determines which values are unchanged and thus do not need to be written out.

Parameters:	`in: ib_crsr`	is an open table cursor
	`in: ib_old_tpl`	The column values of the old record on which the cursor is positioned.
	`in: ib_new_tpl`	The new values that need to be updated.
Returns:	DB_SUCCESS or error code	

6.7.2.21. ib_cursor_delete_row

Synopsis

```
ib_err_t ib_cursor_delete_row(ib_crsr_t ib_crsr);
```

Delete the current row from a table. Current row is where the cursor is positioned.

Parameters:	`in: ib_crsr`	is an open table cursor
Returns:	DB_SUCCESS or error code	

6.7.2.22. ib_cursor_read_row

Synopsis

```
ib_err_t ib_cursor_read_row(ib_crsr_t ib_crsr, ib_tpl_t ib_tpl);
```

Read the current row from a table or a secondary index. Current row is where the cursor is positioned. There are two types of read operations: reading the key columns, and reading all the columns. This function can be called with either a table cursor or a secondary index cursor. The columns that are actually read depends on the tuple type. See also ib_sec_search_tuple_create, ib_sec_read_tuple_create, ib_clust_search_tuple_create and ib_clust_read_tuple_create for creating different tuple types.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor
	`in/out: ib_tpl`	Tuple to read the values into
Returns:	DB_SUCCESS or error code	

6.7.2.23. ib_cursor_moveto

Synopsis

```
ib_err_t ib_cursor_moveto(
        ib_crsr_t       ib_crsr,
        ib_tpl_t        ib_tpl,
        ib_srch_mode_t  ib_srch_mode,
  int*            result);
```

Search for a key in the clustered index (that is, the full table) or via the secondary index.

Parameters:	`in: ib_crsr`	is an open table or secondary index cursor
	`in: ib_tpl`	Tuple to search for. It must be created by either the ib_sec_search_tuple_create or ib_clust_search_tuple_create function.
	`in: ib_srch_mode`	This determines where the cursor will be positioned on return depending on the last match. See also search modes.
	`out: result`	result is -1, 0 or 1 depending on tuple equal or greater than the current row
Returns:	DB_SUCCESS or error code	

6.7.3. API Functions - Transactions

These functions deal with transactions. For usage instructions, see Chapter 5, *Managing Transactions with Embedded InnoDB*. To see these APIs in action, look in the sample programs listed in Appendix B, *Code Examples Supplied with Embedded InnoDB*.

6.7.3.1. ib_trx_begin

Synopsis

```
ib_trx_t ib_trx_begin(ib_trx_level_t ib_trx_level);
```

Begin a new transaction, put the transaction in the active state, and return a handle to the transaction.

Parameters:	`in: ib_trx_level`	The transaction isolation level
Returns:	a valid transaction handle, or `NULL` on failure	

```
ib_trx_t    ib_trx;
ib_err_t    ib_err;
ib_crsr_t   ib_crsr;

ib_trx = ib_trx_begin(IB_TRX_REPEATABLE_READ);
ib_err = ib_cursor_open_table("db/t", ib_trx, &ib_crsr);
... Do some DML operations ...
ib_err = ib_cursor_close(ib_crsr);
ib_trx_commit(ib_trx);
```

6.7.3.2. ib_trx_commit

Synopsis

```
ib_err_t ib_trx_commit(ib_trx_t ib_trx);
```

Commits a transaction. Also releases any schema latches, and frees the transaction handle.

Parameters:	`in: ib_trx`	is an active transaction handle
Returns:	DB_SUCCESS or error code	

6.7.3.3. ib_trx_state

Synopsis

```
ib_trx_state_t ib_trx_state(ib_trx_t ib_trx);
```

Query the transaction's state. This function can be used to check for the state of the transaction in case it has been rolled back by the InnoDB deadlock detector. Note that when a transaction is selected as a victim for rollback, InnoDB will always return an appropriate error code indicating this. See also DB_DEADLOCK, DB_LOCK_TABLE_FULL and DB_LOCK_WAIT_TIMEOUT

Parameters:	`in: ib_trx`	handle to a transaction instance
Returns:	Transaction state	

6.7.3.4. ib_trx_start

Synopsis

```
ib_err_t ib_trx_start(ib_trx_t ib_trx, ib_trx_level_t ib_trx_level);
```

Restart a transaction that has been rolled back. This special function exists for the case when the deadlock detector has rolled back a transaction. While the transaction has been rolled back, the handle is still valid and can be reused by calling this function. If you do not want to reuse the transaction handle, you can free the handle by calling ib_trx_release().

Parameters:	`in: ib_trx`	is a handle to a transaction that is in state `IB_TRX_NOT_STARTED`
	`in: ib_trx_level`	The transaction isolation level for the new transaction.
Returns:	DB_SUCCESS or error code	

6.7.3.5. ib_trx_release

Synopsis

```
ib_err_t ib_trx_release(ib_trx_t ib_trx);
```

Release the transaction handle. If a transaction was rolled back by Embedded InnoDB because of a lock wait time-out or deadlock, the rolled back transaction cannot be committed or rolled back using ib_trx_commit or ib_trx_rollback. The transaction must be released using this function, or else restarted by ib_trx_start. The transaction must be in state IB_TRX_NOT_STARTED; use the ib_trx_state function to query the state of a transaction handle.

Parameters:	`in: ib_trx`	is a handle to a transaction that is in state `IB_TRX_NOT_STARTED`
Returns:	DB_SUCCESS or error code	

```
ib_trx_t    ib_trx;
ib_err_t    ib_err;

ib_trx = ib_trx_begin(IB_TRX_REPEATABLE_READ);
... Do some DML operations ...
/* If the transaction is rolled back by InnoDB then the
```

```
        transaction handle ib_trx is still valid and has to be
        destroyed using ib_trx_release() explicitly. */
        if (ib_err == DB_DEADLOCK || ib_err == DB_LOCK_WAIT_TIMEOUT) {
                ib_err = ib_trx_release(ib_trx);
                /* Check return value. */
        } else {
                /* Assuming everything went OK. */
                ib_trx_commit(ib_trx);
        }
```

6.7.3.6. ib_trx_rollback

Synopsis

```
ib_err_t ib_trx_rollback(ib_trx_t ib_trx);
```

Roll back a transaction. This function also release any locks acquired by cursors attached to the transaction, and frees (releases) the transaction handle.

Parameters:	`in: ib_trx`	is a handle to a transaction that is in state `IB_TRX_ACTIVE`.
Returns:	DB_SUCCESS or error code	

6.7.3.7. ib_savepoint_take

Synopsis

```
ib_err_t ib_savepoint_take(ib_trx_t ib_trx, const void* name,
                           ib_ulint_t name_len);
```

Creates a named savepoint. If the transaction is not yet started, starts it. If there is already a savepoint of the same name, this call erases that old savepoint and replaces it with a new one. Savepoints are deleted in a transaction commit or rollback.

Parameters:	`in: ib_trx`	is a handle to a transaction that is in state `IB_TRX_ACTIVE`.
	`in: name`	Pointer to an arbitrary string, that identifies the savepoint. InnoDB does not attach any significance to this string. The NUL byte is not significant.
	`in: name_len`	Length of the name in bytes.

6.7.3.8. ib_savepoint_release

Synopsis

```
ib_err_t ib_savepoint_release(
         ib_trx_t        ib_trx,
         const void*     name,
          ib_ulint_t      name_len);
```

Releases only the named savepoint. Any savepoints that were set after this savepoint are left as-is.

Parameters:	`in: ib_trx`	is a handle to a transaction that is in state `IB_TRX_NOT_STARTED`
	`in: name`	Savepoint name that was used in `ib_savepoint_take()`
	`in: name_len`	length of name in bytes
Returns:	DB_SUCCESS or error code	

6.7.3.9. ib_savepoint_rollback

Synopsis

```
ib_err_t ib_savepoint_rollback(
         ib_trx_t        ib_trx,
         const char*     name,
         ib_ulint_t      name_len);
```

Rolls back a transaction back to a named savepoint. Modifications after the savepoint are undone, but InnoDB **does not** release the corresponding locks that are stored in memory. If a lock is implicit, that is, a new inserted row holds a lock where the lock information is carried by the transaction ID stored in the row, these locks are naturally released in the rollback. Any savepoints that were set after this savepoint are deleted.

Parameters:	`in: ib_trx`	is a handle to a transaction that is in state `IB_TRX_NOT_STARTED`
	`in: name`	The name of the savepoint used in `ib_savepoint_take()`; can be `NULL`
	`in: name_len`	Length of the savepoint name. If name is `NULL`, this parameter is ignored.
Returns:	DB_SUCCESS or error code	

6.7.4. API Functions - Configuration

These APIs deal with the configuration variables listed in Chapter 8, *Embedded InnoDB Configuration Variables*. For usage information, see Section 2.5, “Configuration”. To see these APIs in action, look in the sample programs listed in Appendix B, *Code Examples Supplied with Embedded InnoDB*.

6.7.4.1. ib_cfg_var_get_type

Synopsis

```
ib_err_t ib_cfg_var_get_type(const char* name, ib_cfg_var_type_t* type);
```

Get the type of an InnoDB configuration variable.

Parameters:	`in: name`	name of a configuration variable
	`out: type`	The type of the variable.
Returns:	DB_SUCCESS or error code	

6.7.4.2. ib_cfg_set_int

Synopsis

```
ib_err_t ib_cfg_set_int(const char* name, ib_ulint_t value);
```

Set an integer configuration variable value.

Parameters:	`in: name`	name of configuration variable.
	`in: value`	the value to set
Returns:	DB_SUCCESS or error code	

6.7.4.3. ib_cfg_set_text

Synopsis

```
ib_err_t ib_cfg_set_text(const char* name, const char* value);
```

Set a text configuration variable value.

Parameters:	`in: name`	name of configuration variable.
	`in: value`	the value to set
Returns:	DB_SUCCESS or error code	

6.7.4.4. ib_cfg_set_bool_on

Synopsis

```
ib_err_t ib_cfg_set_bool_on(const char* name);
```

Set a Boolean configuration variable to ON.

Parameters:	`in: name`	name of configuration variable.
Returns:	DB_SUCCESS or error code	

6.7.4.5. ib_cfg_set_bool_off

Synopsis

```
ib_err_t ib_cfg_set_bool_off(const char* name);
```

Unset a Boolean configuration variable.

Parameters:	`in: name`	name of configuration variable.
Returns:	DB_SUCCESS or error code	

6.7.4.6. ib_cfg_set_callback

Synopsis

```
ib_err_t ib_cfg_set_callback(const char* name, ib_cb_t func);
```

Set a generic callback function.

Parameters:	`in: name`	name of configuration variable.
	`in: func`	pointer to the callback function
Returns:	DB_SUCCESS or error code	

6.7.4.7. ib_cfg_get

Synopsis

```
ib_err_t ib_cfg_get(const char* name, void* value);
```

Retrieve the value of a configuration variable. Returns a generic pointer to the value, which must be typecast according to the type of the variable, using the types listed in Section 6.2.2, “Configuration Variable Types”.

Parameters:	`in: name`	configuration variable name
	`in: value`	Pointer to a variable that matches the type of the configuration parameter. The InnoDB engine has no way of telling whether your program has set the correct type. Use ib_cfg_var_get_type to determine the type of the configuration variable.
Returns:	DB_SUCCESS or error code	

6.7.4.8. ib_cfg_get_all

Synopsis

```
ib_err_t ib_cfg_get_all(const char*** names, ib_u32_t* names_num);
```

Get a list of the names of all configuration variables. The caller is responsible for free(3)ing the returned array of strings when it is not needed anymore and for not modifying the individual strings. The type of each individual variable can be retrieved with ib_cfg_var_get_type() and its value with ib_cfg_get()

Parameters:	`out: names`	pointer to array of strings
	`out: names_num`	number of strings returned
Returns:	DB_SUCCESS or error code	

6.7.5. API Functions - DDL

These APIs deal with the create and drop operations, which are categorized as Data Definition Language, known as DDL. For usage information, see Chapter 3, *Performing CREATE and DROP Operations (DDL) with Embedded InnoDB*. To see these APIs in action, look in the sample programs listed in Appendix B, *Code Examples Supplied with Embedded InnoDB*.

6.7.5.1. ib_table_schema_create

Synopsis

```
ib_err_t ib_table_schema_create(
         const char*     name,
         ib_table_sch_t* ib_tbl_sch,
         ib_tbl_fmt_t    ib_tbl_fmt,
         ib_ulint_t      page_size);
```

Create a table schema handle, which is used for creating InnoDB tables.

Parameters:	`in: name`	The table name for which to create the schema. Currently, Embedded InnoDB requires the name of the database to be encoded in the name, for example: `test_db/table`.
	`out: ib_tbl_sch_t`	The handle for the schema.
	`in: ib_tbl_fmt`	The table format
	`in: page_size`	For compressed tables, the page size can be one of 0, 1, 2, 4, 8 or 16. Specifying a size of 0 for compressed tables selects the system default page size. For regular tables, this value is ignored.
Returns:	DB_SUCCESS or error code	

6.7.5.2. ib_table_schema_delete

Synopsis

```
ib_err_t ib_table_schema_delete(ib_tbl_sch_t ib_tbl_sch);
```

Deletes the table schema handle, once you have created the table.

Parameters:	`in: ib_tbl_sch`	is a handle to a table schema

6.7.5.3. ib_table_schema_add_col

Synopsis

```
ib_err_t ib_table_schema_add_col(
   ib_tbl_sch_t    ib_tbl_sch,
   const char*     name,
   ib_col_type_t   ib_col_type,
   ib_col_attr_t   ib_col_attr,
   ib_u16_t        client_type,
   ib_ulint_t      len);
```

Add a column to a table schema.

Parameters:	`in: ib_tbl_sch`	the table schema handle
	`in: name`	is the name of the column.
	`in: ib_col_type`	is the type of the column
	`in: ib_col_attr`	is the column attribute
	`in: client_type`	currently ignored, can be 0.
	`in: len`	is the length of the column. For BLOB types, the value should be 0.
Returns:	DB_SUCCESS or error code	

6.7.5.4. ib_table_schema_add_index

Synopsis

```
ib_err_t ib_table_schema_add_index(
         ib_tbl_sch_t    ib_tbl_sch,
         const char*     name,
         ib_idx_sch_t*   ib_idx_sch);
```

Add an index to the table schema. Use this to create the clustered index, and optionally one or more secondary indexes, when creating a table. The index schema is owned by the table schema instance, and is freed when the table schema instance is freed.

Strictly speaking, you do not need to create a clustered index (primary key) yourself, because the InnoDB engine constructs one if necessary using the row ID. However, for tables of any size or importance, you should typically define the primary key to optimize the most critical queries on the table.

Parameters:	`in: ib_tbl_sch`	is a handle to a table schema

	`in: name`	is the name of the index.
	`out: ib_idx_sch`	is the index schema handle
Returns:	DB_SUCCESS or error code	

6.7.5.5. ib_index_schema_create

Synopsis

```
ib_err_t ib_index_schema_create(
         ib_trx_t        ib_usr_trx,
         const char*     name,
         const char*     table_name,
         ib_idx_sch_t*   ib_idx_sch);
```

Create an index schema handle.

Parameters:	`in: ib_usr_trx`	is a handle to an active transaction.
	`in: name`	is the name of the index to create
	`in: table_name`	is the name of the parent table
	`out: ib_idx_sch`	is the index schema handle
Returns:	DB_SUCCESS or error code	

6.7.5.6. ib_index_schema_set_clustered

Synopsis

```
ib_err_t ib_index_schema_set_clustered(ib_idx_sch_t ib_idx_sch);
```

Set the index type to clustered. The default is secondary. The index is also tagged as a unique index.

Parameters:	`in: ib_idx_sch`	is a handle to an index schema
Returns:	DB_SUCCESS or error code	

6.7.5.7. ib_index_schema_set_unique

Synopsis

```
ib_err_t ib_index_set_unique(ib_idx_sch_t ib_idx_sch);
```

Tag the index as a unique index. Must be done before the index is created.

Parameters:	`in: ib_idx_sch`	is the index schema handle
Returns:	DB_SUCCESS or error code	

6.7.5.8. ib_index_schema_delete

Synopsis

```
ib_err_t ib_index_schema_delete(
         ib_idx_sch_t    ib_idx_sch);
```

Deletes an index schema handle created with `ib_index_schema_create`, after you have created the index.

Parameters:	`in: ib_idx_sch`	is the index schema handle to delete
Returns:	DB_SUCCESS or error code	

6.7.5.9. ib_index_schema_add_col

Synopsis

```
ib_err_t ib_index_schema_add_col(
         ib_idx_sch_t    ib_idx_sch,
         const char*     name,
         ib_ulint_t      prefix_len);
```

Specify a column name that is going to be part of the index key.

The prefix length only applies for `BLOB` or long `VARCHAR` values. It represents the number of bytes to examine to determine if two values are equal or how they should be ordered. For typical columns, leave this value as zero.

Parameters:	`in: ib_idx_sch`	is a handle to an index schema
	`in: name`	is the name of the column to include in the index key.
	`in: prefix_len`	is the length of the prefix if it is a prefix key, otherwise 0.
Returns:	DB_SUCCESS or error code	

6.7.5.10. ib_table_create

Synopsis

```
ib_err_t ib_table_create(ib_trx_t ib_trx, ib_tbl_sch_t ib_tbl_sch, ib_id_t* id);
```

Create a table using the schema definition. The metadata for the table is stored in the InnoDB data dictionary. The table itself can be stored in the system tablespace, or in a separate tablespace file if the file-per-table option is enabled.

If the table exists in the database, this function returns `DB_TABLE_IS_BEING_USED` and `id` contains that ID of the table.

Parameters:	`in: ib_trx_t`	is the transaction that is used to cover the table creation. The transaction must be holding the schema lock in exclusive mode; see ib_schema_lock_exclusive().
	`in: ib_tbl_sch`	is a handle to a table schema
	`out: id`	is the table id
Returns:	DB_SUCCESS or error code	

6.7.5.11. ib_index_create

Synopsis

```
ib_err_t ib_index_create(ib_idx_sch_t ib_idx_sch, ib_id_t id);
```

Create a secondary index using a user-defined index schema.

Parameters:	`in: ib_idx_sch`	is a handle to an index schema
	`out: id`	is the index ID; the index ID has the parent table ID encoded in the high 32 bits.
Returns:	DB_SUCCESS or error code	

6.7.5.12. ib_table_drop

Synopsis

```
ib_err_t ib_table_drop(ib_trx_t trx, const char* name);
```

Drop a table by table name. If the table is in a separate tablespace file because of the file-per-table setting, the tablespace file is deleted. Your must acquire an exclusive lock over the schema before attempting this operation.

Parameters:	`in: trx`	is the covering transaction.
	`in: name`	is the name of the table to drop, qualified with the database name, for example: `test_db/table`.
Returns:	DB_SUCCESS or error code	

6.7.5.13. ib_index_drop

Synopsis

```
ib_err_t ib_index_drop(ib_trx_ti ib_trx, ib_id_t index_id);
```

Drop a secondary index by ID. You must acquire an exclusive lock over the schema before attempting this operation.

Parameters:	`in: ib_trx`	is the covering transaction.
	`in: index_id`	is the ID of the index to drop
Returns:	DB_SUCCESS or error code	

6.7.5.14. ib_table_lock

Synopsis

```
ib_err_t ib_table_lock(
        ib_trx_t        ib_trx,
        ib_id_t         table_id,
        ib_lck_mode_t   ib_lck_mode);
```

Lock a table in IB_LOCK_IX (exclusive) or IB_LOCK_IS (shared) mode. Use a shared lock if your transaction performs only read operations on the table. Use an exclusive lock if your transaction performs read and write operations on the table.

Parameters:	`in: ib_trx`	is an active transaction
	`in: table_id`	is id of the table to lock.
	`out: ib_lck_mode`	is the lock mode
Returns:	DB_SUCCESS or error code	

6.7.5.15. ib_table_rename

Synopsis

```
ib_err_t ib_table_rename(
        ib_trx_t        ib_trx,
        const char*     old_name,
        const char*     new_name);
```

Rename a table.

You specify fully qualified names (of the format `database/table`) for both the old and the new name. The Embedded InnoDB API does not have the idea of “connecting” to a database and using that as the default for all operations. The Embedded InnoDB API also does not use the “backtick” notation for specifying table names that match reserved words or contain special punctuation characters.

Parameters:	`in: ib_trx`	is an active transaction
	`in: old_name`	is the fully qualified name of a table, for example: `test_db/table`.
	`in: new_name`	is the fully qualified name of a table, for example: `test_db/table`.

Returns: DB_SUCCESS or error code

6.7.5.16. ib_database_create

Synopsis

```
ib_bool_t ib_database_create(const char* dbname);
```

Create a database and its associated directory, if it does not exist. The database is created relative to the `data_home_dir` configuration setting. If the `file_per_table` configuration variable is not set, this function is a no-op (because no directory is needed) but still returns `IB_TRUE`.

Parameters:	`in: dbname`	is the database name; it should not contain any "`/`" or "`\`" characters, or any character that is not allowed in a filename or path name.

Returns: IB_TRUE on success and IB_FALSE on any failure.

6.7.5.17. ib_database_drop

Synopsis

```
ib_err_t ib_database_drop(const char* dbname);
```

Drop a database, all the tables it contains, and its associated directory. Use with care! If the directory is not empty after the tables are dropped (that is, if you store other files in the directory), it is not removed.

Parameters:	`in: dbname`	is database name to be dropped

Returns: DB_SUCCESS or error code

6.7.6. API Functions - DDL and DML Support

To see these APIs in action, look in the sample programs listed in Appendix B, *Code Examples Supplied with Embedded InnoDB*.

6.7.6.1. ib_status_get_i64

Synopsis

```
ib_err_t ib_status_get_i64(const char* name, ib_i64_t* dst);
```

Get the value of an INT status variable. Use this function to access variables that represent the internal state of InnoDB.

Parameters:	`in: name`	is a name of an internal status variable of type integer.
	`in: dst`	points to the integer where the value is copied.

Returns:	DB_SUCCESS; DB_DATA_MISMATCH if found but type is not INT; or DB_NOT_FOUND if no match is found.

6.7.6.2. ib_set_client_compare

Synopsis

```
void ib_set_client_compare(ib_client_cmp_t client_cmp_func);
```

Set the callback function for comparing the column keys. This function can only be used to override comparison of BLOBs and user-defined column types (currently not implemented). For details of parameters and types for the callback function, see `ib_client_cmp_t` in `innodb.h`.

Parameters:	`in: client_cmp_func`	is a client column compare function

6.7.6.3. ib_schema_tables_iterate

Synopsis

```
ib_err_t ib_schema_tables_iterate(
         ib_trx_t        ib_trx,
         ib_schema_visitor_table_all_t visitor,
         void*           arg);
```

This function iterates over all the tables in the InnoDB data dictionary, including the system tables. It aborts if the `visitor` function returns a non-zero value. It calls the function:

`visitor.tables(arg, const char* tablename, int tablename_len)`

Parameters:	`in: ib_trx`	is an active transaction
	`in: visitor`	is the callback function.
	`out: arg`	is the argument for the callback function
Returns:	DB_SUCCESS or error code	

6.7.6.4. ib_table_schema_visit

Synopsis

```
ib_err_t ib_table_schema_visit(
         ib_trx_t        ib_trx,
         const char*     name,
         const ib_schema_visitor_t*  visitor,
         void*           arg);
```

Read the schema (column and index definitions) for a table, using the visitor pattern. It makes the following sequence of calls:

```
visitor->table()
visitor->table_col() for each user column
visitor->index() for each user index
visitor->index_col() for each column in user index
```

It stops if any of the above functions returns a non-zero value. The caller must have an exclusive lock on the InnoDB data dictionary.

Parameters:	`in: ib_trx`	is an active transaction instance
	`in: name`	is the name of the table.
	`out: visitor`	is the callback function
	`out: arg`	is the argument for the callback function
Returns:	DB_SUCCESS or error code	

6.7.6.5. ib_schema_lock_exclusive

Synopsis

```
ib_err_t ib_schema_lock_exclusive(ib_trx_t ib_trx);
```

Locks the InnoDB data dictionary in exclusive mode. Call this function before making changes to the data dictionary such as creating or dropping tables or indexes. (For InnoDB experts: this operation acquires the dictionary mutex.)

Parameters:	`in: ib_trx`	is an active transaction handle
Returns:	DB_SUCCESS or error code	

6.7.6.6. ib_schema_lock_shared

Synopsis

```
ib_err_t ib_schema_lock_shared(ib_trx_t ib_trx);
```

Locks the InnoDB data dictionary in shared mode. Use while performing read-only operations on the data dictionary, such as traversing the list of tables.

Parameters:	`in: ib_trx`	is an active transaction handle
Returns:	DB_SUCCESS or error code	

6.7.6.7. ib_schema_lock_is_exclusive

Synopsis

```
ib_bool_t ib_schema_lock_is_exclusive(const ib_trx_t ib_trx);
```

Checks whether the transaction holds an exclusive lock on the data dictionary.

Parameters:	`in: ib_trx`	is a transaction handle
Returns:	IB_TRUE if the transaction holds an exclusive lock on the data dictionary.	

6.7.6.8. ib_schema_lock_is_shared

Synopsis

```
ib_bool_t ib_schema_lock_is_shared(const ib_trx_t ib_trx);
```

Checks whether the transaction holds a shared lock on the data dictionary.

Parameters:	`in: ib_trx`	is a transaction handle
Returns:	IB_TRUE if the transaction holds a shared lock on the data dictionary.	

6.7.6.9. ib_schema_unlock

Synopsis

```
ib_err_t ib_schema_unlock(ib_trx_t ib_trx);
```

Releases all locks on the data dictionary.

Note

Currently, when InnoDB rolls back a transaction, it does not release the special data dictionary lock acquired by the `ib_schema_lock_exclusive()` or `ib_schema_lock_shared()` functions. Your application must do this explicitly.

Parameters:	`in: ib_trx`	is an active transaction handle that owns the data dictionary lock.
Returns:	DB_SUCCESS or error code	

6.7.6.10. ib_cursor_truncate

Synopsis

```
ib_err_t ib_cursor_truncate(ib_crsr_t* ib_crsr, ib_id_t* id);
```

Truncate a table. Works by creating a new table space and dropping the old one. This operation is faster than stepping through all the records of the table and deleting each one. Any indexes on the table are recreated, and are initially empty.

Parameters:	`out: ib_crsr`	is a pointer to an open table cursor, on success it will be set to NULL
	`out: id`	is the new table id
Returns:	DB_SUCCESS or error code	

6.7.6.11. ib_table_truncate

Synopsis

```
ib_err_t ib_table_truncate(const const* name, ib_id_t* table_id);
```

Truncate a table. Works by creating a new table space and dropping the old one. This operation is faster than stepping through all the records of the table and deleting each one. Any indexes on the table are recreated, and are initially empty.

Parameters:	`in: name`	is a fully qualified table name, for example: `test_db/table`.
	`out: table_id`	is the new table id
Returns:	DB_SUCCESS or error code	

6.7.6.12. ib_table_get_id

Synopsis

```
ib_err_t ib_table_get_id(const char* table_name, ib_id_t* table_id);
```

Look up the ID of a table using the table name. The table name is case-sensitive.

Parameters:	`in: table_name`	is a fully qualified table name.
	`out: table_id`	is the table id if found.
Returns:	DB_SUCCESS or error code	

6.7.6.13. ib_index_get_id

Synopsis

```
ib_err_t ib_index_get_id(
        const char*     table_name,
        const char*     index_name,
        ib_id_t*        index_id);
```

Look up the ID of an index.

Parameters:	`in: table_name`	is the name of the table that contains the index.
	`in: index_name`	is the index to look for.
	`out: index_id`	is the index id.
Returns:	DB_SUCCESS or error code	

6.7.6.14. ib_logger_set

Synopsis

```
void ib_logger_set(ib_msg_log_t ib_msg_log, ib_msg_stream_t ib_msg_stream);
```

Set the function that logs InnoDB informational and error messages. Currently, this function must be a drop-in replacement for `fprintf()`. (Issue `man 3 fprintf` to see the function prototype.)

Parameters:	`in: ib_msg_log`	is the message logging function
	`in: ib_msg_stream`	is really a callback argument relevant only to the ib_msg_log function.

6.7.6.15. ib_strerror

Synopsis

```
const char* ib_strerror(ib_err_t db_errno);
```

Convert an error number to a human-readable text message. The returned string is static, and should not be freed or modified.

Parameters:	`in: db_errno`	is the InnoDB error code, one of the values from Section 6.6, “Error and Status Codes”.
Returns:	String representation of the error code	

6.7.7. API Functions - Tuples

These functions work with tuples. To understand how tuples relate to queries and other operations using a `WHERE` clause, see Section 1.4, “Overview of Queries and WHERE Clauses”. For usage information, see Section 4.3.5, “Emulating a WHERE Clause”. To see these APIs in action, look in the sample programs listed in Appendix B, *Code Examples Supplied with Embedded InnoDB*.

6.7.7.1. ib_sec_read_tuple_create

Synopsis

```
ib_tpl_t ib_sec_read_tuple_create(ib_crsr_t ib_crsr);
```

Create a tuple for reading a row from the secondary index. This tuple contains the columns from the secondary index key, and the primary key columns for the table.

Parameters:	`in: ib_crsr`	is a cursor opened on a secondary index.

Returns: A tuple instance that can be used to read secondary index key columns. The columns include the cluster key columns also.

6.7.7.2. ib_sec_search_tuple_create

Synopsis

```
ib_tpl_t ib_sec_search_tuple_create(ib_crsr_t ib_crsr);
```

Create a tuple for searching secondary indexes. This tuple contains only the columns for the secondary index key.

Parameters:	`in: ib_crsr`	is a cursor opened on a secondary index.

Returns: A tuple instance that can be used to search the secondary index. It only contains columns that make up the secondary index key.

6.7.7.3. ib_clust_read_tuple_create

Synopsis

```
ib_tpl_t ib_clust_read_tuple_create(ib_tpl_t ib_tpl);
```

Create a tuple for reading and writing a row in the clustered index. This tuple contains all the columns for the table. (It even includes the system-defined columns that are normally invisible to user code.)

Parameters:	`in: ib_crsr`	is a cursor opened on a table

Returns: A tuple that can be used for reading and writing the columns in table.

6.7.7.4. ib_clust_search_tuple_create

Synopsis

```
ib_tpl_t ib_clust_search_tuple_create(ib_crsr_t ib_crsr);
```

Create a tuple for searching the clustered index. The tuple contains only those columns that make up the primary key.

Parameters:	`in: ib_crsr`	is a cursor opened on a table

Returns: A tuple that can be used for searching the primary key.

6.7.7.5. ib_tuple_get_cluster_key

Synopsis

```
ib_err_t ib_tuple_get_cluster_key(
         ib_crsr_t       ib_crsr,
         ib_tpl_t*       ib_dst_tpl,
         const ib_tpl_t  ib_src_tpl);
```

Create a tuple containing the primary key columns, for use in searching the clustered index. The primary key column values are extracted from the secondary index key tuple. (If your query can be satisfied by examining the primary key columns, rather than performing a further lookup, you can examine the secondary index key tuple directly.)

Parameters:	`in: ib_crsr`	is a secondary index cursor.
	`out: ib_dst_tpl`	is a tuple that can be used to search the clustered index.
	`in: ib_src_tpl`	is a secondary index read tuple.

Returns: DB_SUCCESS or error code

6.7.7.6. ib_tuple_delete

Synopsis

```
void ib_tuple_delete(ib_tpl_t ib_tpl);
```

Delete a tuple handle, after you are finished using the tuple.

Parameters:	`in: ib_tpl`	is the tuple handle to delete.

6.7.7.7. ib_tuple_get_n_cols

Synopsis

```
ib_ulint_t ib_tuple_get_n_cols(const ib_tpl_t ib_tpl);
```

Determine the total number of columns in the tuple.

Parameters:	`in: ib_tpl`	is the tuple instance to query.

Returns: The total number of columns in the tuple, including system columns.

6.7.7.8. ib_tuple_get_n_user_cols

Synopsis

```
ib_ulint_t ib_tuple_get_n_user_cols(const ib_tpl_t ib_tpl);
```

Determine the total number of user-defined columns in the tuple definition.

Parameters:	`in: ib_tpl`	is the tuple instance to query.
Returns:	The number of user-defined columns only, not counting the system columns.	

6.7.7.9. ib_tuple_copy

Synopsis

```
ib_err_t ib_tuple_copy(ib_tpl_t ib_dst_tpl, const ib_tpl_t ib_src_tpl);
```

Copy contents of one tuple to another.

Parameters:	`in: ib_dst_tpl`	is the destination tuple.
	`in: ib_src_tpl`	is the source tuple.
Returns:	DB_SUCCESS or error code	

6.7.7.10. ib_tuple_clear

Synopsis

```
ib_tpl_t ib_tuple_clear(ib_tpl_t ib_tpl);
```

Resets the tuple so that it can be used to insert/update/search another row. All columns are set to SQL `NULL` values.

Parameters:	`in: ib_tpl`	is the tuple to clear, the tuple will be freed
Returns:	an instance of a new tuple that is the same type as the tuple that was passed in	

6.7.7.11. ib_col_get_meta

Synopsis

```
ib_ulint_t ib_col_get_meta(
        ib_tpl_t       ib_tpl,
        ib_ulint_t     col_no,
        ib_col_meta_t  ib_col_meta);
```

Get a column’s type, length and attributes from the tuple.

Parameters:	`in: ib_tpl`	is an instance of a tuple.
	`in: col_no`	is column ordinal value.
	`out: ib_col_meta`	is where the column meta data will be copied.
Returns:	Length of the data in the column, can be IB_SQL_NULL	

6.7.7.12. ib_col_get_value

Synopsis

```
const void* ib_col_get_value(ib_tpl_t ib_tpl, ib_ulint_t col_no);
```

Get the value of a column within a tuple. The `get_` functions are for known-length items such as integers, while the `copy_` functions are for variable-length items such as strings.

Parameters:	`in: ib_tpl`	is the tuple to read from.
	`in: col_no`	is the ordinal value of the column.
Returns:	a pointer to the actual raw data; returns `NULL` if the column value is `IB_SQL_NULL`	

6.7.7.13. ib_col_copy_value

Synopsis

```
ib_ulint_t ib_col_copy_value(
        ib_tpl_t        ib_tpl,
        ib_ulint_t      col_no,
        void*           dst,
        ib_ulint_t      max_len);
```

Copy the value from the tuple to a user buffer. The `get_` functions are for known-length items such as integers, while the `copy_` functions are for variable-length items such as strings.

Parameters:	`in: ib_tpl`	is the tuple to copy from.
	`in: col_no`	is the ordinal value of the column.
	`out: dst`	is the buffer to copy to.
	`out: max_len`	is the maximum number of bytes that can be copied to `dst`.
Returns:	The number of bytes copied or `IB_SQL_NULL`	

6.7.7.14. ib_col_get_len

Synopsis

```
ib_ulint_t ib_col_get_len(ib_tpl_t ib_tpl, ib_ulint_t col_no);
```

Get the size of the data in bytes in a column of the tuple. The `get_` functions are for known-length items such as integers, while the `copy_` functions are for variable-length items such as strings.

Parameters:	`in: ib_tpl`	is the tuple to read from.
	`in: col_no`	is the column ordinal number.
Returns:	The number of bytes or `IB_SQL_NULL`	

6.7.7.15. ib_col_set_value

Synopsis

```
ib_err_t ib_col_set_value(
         ib_tpl_t        ib_tpl,
         ib_ulint_t      col_no,
         const void*     src,
         ib_ulint_t      len);
```

Set the value of a column in the tuple. The number of bytes copied is the minimum of column length and length specified by the `len` parameter.

Parameters:	`in: ib_tpl`	is the tuple to write to.
	`in: col_no`	is the column ordinal value.
	`in: src`	is the pointer to the source data.
	`in: len`	is the maximum number of bytes to copy from `src`.
Returns:	DB_SUCCESS or error code	

6.7.7.16. Functions to read numeric columns

Synopsis

```
ib_err_t ib_tuple_read_i8(ib_tpl_t ib_tpl, int col_no, ib_i8_t* val);
ib_err_t ib_tuple_read_u8(ib_tpl_t ib_tpl, int col_no, ib_i8_t* val);
ib_err_t ib_tuple_read_i16(ib_tpl_t ib_tpl, int col_no, ib_i16_t* val);
ib_err_t ib_tuple_read_u16(ib_tpl_t ib_tpl, int col_no, ib_u16_t* val);
ib_err_t ib_tuple_read_i32(ib_tpl_t ib_tpl, int col_no, ib_i32_t* val);
ib_err_t ib_tuple_read_u32(ib_tpl_t ib_tpl, int col_no, ib_i32_t* val);
ib_err_t ib_tuple_read_i64(ib_tpl_t ib_tpl, int col_no, ib_i64_t* val);
ib_err_t ib_tuple_read_u64(ib_tpl_t ib_tpl, int col_no, ib_u64_t* val);
ib_err_t ib_tuple_read_float(ib_tpl_t ib_tpl, int col_no, float* val);
```

```
    ib_err_t ib_tuple_read_double(ib_tpl_t ib_tpl, int col_no, double* val);
```

These getter functions provide a typed interface for reading values from numerical columns.

Parameters:	`in: ib_tpl`	is valid InnoDB tuple instance
	`in: col_no`	is the ordinal value of the column
	`out: val`	is where the value read will be written
Returns:	DB_SUCCESS or error code	

6.7.7.17. Functions to write numeric columns

Synopsis

```
    ib_err_t ib_tuple_write_i8(ib_tpl_t ib_tpl, int col_no, ib_i8_t val);
    ib_err_t ib_tuple_write_u8(ib_tpl_t ib_tpl, int col_no, ib_i8_t val);
    ib_err_t ib_tuple_write_i16(ib_tpl_t ib_tpl, int col_no, ib_i16_t val);
    ib_err_t ib_tuple_write_u16(ib_tpl_t ib_tpl, int col_no, ib_u16_t val);
    ib_err_t ib_tuple_write_i32(ib_tpl_t ib_tpl, int col_no, ib_i32_t val);
    ib_err_t ib_tuple_write_u32(ib_tpl_t ib_tpl, int col_no, ib_i32_t val);
    ib_err_t ib_tuple_write_i64(ib_tpl_t ib_tpl, int col_no, ib_i64_t val);
    ib_err_t ib_tuple_write_u64(ib_tpl_t ib_tpl, int col_no, ib_u64_t val);
    ib_err_t ib_tuple_write_float(ib_tpl_t ib_tpl, int col_no, float val);
    ib_err_t ib_tuple_write_double(ib_tpl_t ib_tpl, int col_no, double val);
```

These setter functions provide a typed interface for writing values to numeric columns.

Parameters:	`in: ib_tpl`	is a valid InnoDB tuple instance
	`in: col_no`	is the ordinal value of the column
	`in: val`	is the value to write to the column
Returns:	DB_SUCCESS or error code	

Chapter 7. Embedded InnoDB Limits

Embedded InnoDB provides support for managing everything from very small databases that fit entirely in memory, to extremely large databases holding millions of records and terabytes of data.

Embedded InnoDB databases store data in a binary format that is portable across platforms, even of differing endian-ness. Integers are always stored in big-endian format; that is why you must call functions to read integer values rather than getting the values directly. Be aware, however, that portability aside, some performance issues can crop up in the event that you are using little-endian architecture.

Also, Embedded InnoDB databases and data structures are designed for concurrent access: they are thread-safe, and they share well across multiple processes. However, to allow multiple processes to share databases and the cache, Embedded InnoDB makes use of mechanisms that do not work well on network-shared drives (NFS or Windows networks shares, for example). For this reason, you cannot place your Embedded InnoDB databases and environments on network-mounted drives.

- Maximum number of databases -- No limit
- Maximum number of tables -- 4 billion
- Maximum size of a table -- 32 terabytes
- Maximum size of a tablespace -- 64 terabytes
- Maximum size of a row -- Depends on the row format. For non-compressed row format, the maximum row size is 8000 bytes if stored on the same page, or n * 4 GB if the row contains n BLOBs or other long columns. Compressed row format has different limits. The maximum size for both row formats also depends on the page size.
- Maximum number of columns in a table -- 1000
- Maximum number of columns in a clustered index -- 1000
- Maximum number of columns in a secondary index -- 1000
- Maximum size of the key columns in an index -- 4 kilobytes
- Maximum number of secondary indexes on a table -- 1000
- Maximum size of variable length columns -- No limit
- Maximum number of concurrent transactions -- No limit on readonly transactions. Concurrent writes are limited to 1023.
- Maximum number of active savepoints -- No limit
- Maximum number of open tables -- No limit

Chapter 8. Embedded InnoDB Configuration Variables

Embedded InnoDB programs can configure the database engine using several variables. Some variables can only be set before startup and cannot be changed once InnoDB has been started up. Some variables can be set at any time. For the API functions that work with configuration variables, see Section 6.7.4, “API Functions - Configuration”.

Variable name	Type	Class	Description
adaptive_hash_index	Boolean	Pre-Startup	Set to OFF to disable the Adaptive Hash Index. Default is ON.
adaptive_flushing	Boolean	Pre-Startup	Set to OFF to disable the adaptive flushing. Default is ON.
additional_mem_pool_size	Integer	Pre-Startup	Embedded InnoDB uses a separate memory pool for house keeping and its data dictionary. The default setting is 4MB. If Embedded InnoDB needs more memory it will allocate using malloc().
autoextend_increment	Integer	Pre-Startup	Embedded InnoDB will extend the tablespace by this many pages when it needs to increase the size of the table space. The default setting is 8 pages.
buffer_pool_size	Integer	Pre-Startup	Size of the buffer pool in mega bytes. Default is 8MB.
checksums	Boolean	Pre-Startup	Set to OFF if you want to disable page checksums. Default is ON.

data_file_path	Text	Pre-Startup	The names and sizes of the Embedded InnoDB tablespaces. The syntax for this variable in pseudo-BNF:
			
			`size := INTEGER`
			`spec := filename:size[K|M|G]`
			`spec := spec[;spec]`
			`spec := spec[:autoextend[:max:size[K|M|G]]]`
			
			An example:
			
			`ibdata1:1G;ibdata2:64M:autoextend`
			
			Note that a Windows path may contain a drive name and a ':' e.g.,
			
			`C:\ibdata\ibdata1:1G`

data_home_dir	Text	Pre-Startup	Directory where the system files will be created. All database directories will also be created relative to this path. Default is “./”.Note: The path must end in a “/” or “\” depending on the platform.
doublewrite	Boolean	Pre-Startup	Controls whether InnoDB will use the double write buffer. Default is ON
file_format	Text	Anytime	The highest format that InnoDB supports. This value is stamped on the InnoDB system tablespace.
file_io_threads	Integer	Pre-Startup	The number of IO threads that Embedded InnoDB will create for IO operations. The default setting is 4 threads.

Variable name	Type	Class	Description
file_per_table	Boolean	Pre-Startup	Controls whether InnoDB will store user tables in one table space or create a .ibd file per table. Default is ON.
flush_log_at_trx_commit	Integer	Pre-Startup	This variable can be set to 0, 1 or 2. • 0 - Force sync of log contents to disk once every second. • 1 - Force sync of log contents to disk at transaction commit. • 2 - Write log contents to disk at transaction commit but do not force sync. Default setting is 1.
flush_method	Text	Pre-Startup	Permitted values are: fsync, O_DIRECT or O_DSYNC on Unices. On Windows there is only one possible setting: async_unbuffered. • fsync - Open files without any special modes and use fsync() to sync all files. • O_DIRECT - Open files using O_DIRECT on Solaris Embedded InnoDB will use directio(). All files are synced using fsync(). • O_DSYNC - Open only the log files using O_SYNC mode, data files are synced using fsync(). The default setting on Unices is: fsync.
force_recovery	Integer	Pre-Startup	Permitted values are 1, 2, 3, 4, 5 or 6. • 1 - Force recovery of tables that have corrupt pages. Use this option to dump tables that have some pages that are corrupt by attempting to skip the corrupt pages. • 2 - Disable the master thread, this will disable the purge operation. • 3 - Do not rollback incomplete transactions during recovery • 4 - Disable insert buffer merge. • 5 - Disable read of UNDO logs, uncommitted transactions are treated as commited. • 6 - Disable the redo log application during recovery
io_capacity	Integer	Pre-Startup	The number of IO operations that the server can do. The default value is 200, the minimum permissible is 100. You can set this to a higher value if your hardware can handle higer IOPS.
lock_wait_timeout	Integer	Pre-Startup	The time in seconds a transaction will wait for a lock. The default valus is 60 seconds.
log_buffer_size	Integer	Pre-Startup	The size of the buffer for storing redo log entries. The default value is 384KB.

Variable name	Type	Class	Description
log_file_size	Integer	Pre-Startup	Size of the REDO log files in mega bytes. Default is 16MB.
log_files_in_group	Integer	Pre-Startup	The number of log files in a group. Embedded InnoDB writes to the log files in a circular fashion. Default value is 2.
log_group_home_dir	Text	Pre-Startup	Path to the directory where the log files will be create. The default value is "./".
max_dirty_pages_pct	Integer	Pre-Startup	The master thread tries to keep the ratio of modified pages in the buffer pool to all database pages in the buffer pool smaller than this number. But it is not guaranteed that the value stays below that during a time of heavy update/insert activity. The default value is 75.
max_purge_lag	Integer	Pre-Startup	For controlling the delay of DML statements if purge is lagging. The default value is 0 (infinite).
lru_old_blocks_pct	Integer	Anytime	This parameter is for the advanced user, change from the default setting only if you know what you are doing. It sets the point in the LRU list from where all pages are classified as "old". It is expressed as a percent. Default value is 37.
lru_block_access_recency	Integer	Anytime	This parameter is for the advanced user, change from the default setting only if you know what you are doing. It sets the threshold in milliseconds between accesses to a block at which it will be made "young". The default value is 0 (disabled).
open_files	Integer	Pre-Startup	If you have set "file_per_table" to ON, then this is useful in setting the number of file descriptors that Embedded InnoDB will keep open. Default value is 300.
read_io_threads	Integer	Pre-Startup	Number of IO threads dedicated to reading. Default value is 4 threads
write_io_threads	Integer	Pre-Startup	Number of IO threads dedicated to writing. Default value is 4 threads
pre_rollback_hook	Callback	Disabled	Future use.
print_verbose_log	Boolean	Pre-Startup	Disable if you want Embedded InnoDB to reduce the number of messages it writes to the logger. Default is ON.
rollback_on_timeout	Integer	Pre-Startup	If set then we rollback the transaction on DB_LOCK_WAIT_TIMEOUT error. Default is ON
status_file	Text	Anytime	Enable this if you want Embedded InnoDB to write the output of the status monitor to the logger. Default is OFF.
sync_spin_loops	Integer	Pre-Startup	The number of times to spin while waiting for a latch. Default value is 30.
use_sys_malloc	Boolean	Pre-Startup	Enable if you want Embedded InnoDB to use the system malloc() and free(). Default is OFF.
version	Text	Anytime	Get the Embedded InnoDB version info. A read-only variable.

Chapter 9. Embedded InnoDB Status variables

Embedded InnoDB allows users to query the state of the database using the `ib_status_get_i64()` function. For an example, see the ib_status.c sample program.

Variable name	Description
read_req_pending	Number of read requests that are pending.
write_req_pending	Number of write requests that are pending.
fsync_req_pending	Number of fsync requests that are pending.
write_req_done	Number of write requests that have been completed.
read_req_done	Number of read requests that have been completed.
fsync_req_done	Number of fsync requests that have been completed.
bytes_total_written	Total number of bytes written.
bytes_total_read	Total number of bytes read.
buffer_pool_current_size	Size of the buffer pool in bytes.
buffer_pool_data_pages	Number of data pages in the buffer pool.
buffer_pool_dirty_pages	Number of dirty pages in the buffer pool.
buffer_pool_misc_pages	Number of miscellaneous pages in the buffer pool.
buffer_pool_free_pages	Number of free pages in the buffer pool.
buffer_pool_read_reqs	Number of read requests issued for pages from the buffer pool.
buffer_pool_reads	Number of buffer pool reads that led to the reading of a disk page.
buffer_pool_waited_for_free	The number of times when we had to wait for a free page in the buffer pool. This can happen when the buffer pool is full and InnoDB needs to do a flush to disk, in order to be able to read or create a page.
buffer_pool_pages_flushed	Number of pages that have been written from the buffer pool to the disk.
buffer_pool_write_reqs	Number of write requests issued by the buffer pool.
buffer_pool_total_pages	Number of buffer pages created in the buffer pool with no read.
buffer_pool_pages_read	Number of read operations.
buffer_pool_pages_written	Number of write operations.
double_write_pages_written	Number of pages that have been flushed to the double write buffer.
double_write_invoked	Number of times the double write buffer was flushed.
log_buffer_slot_waits	Number of the log buffer had to flush to disk due to lack of space.
log_write_reqs	Number of log write requests.
log_write_flush_count	Number of physical writes to the log file.
log_bytes_written	Number of bytes written to the log file.
log_fsync_req_done	Number of fsyncs() done on the log file.
log_write_req_pending	Number of pending writes on the log file.
log_fsync_req_pending	Number of fsync() requests pending on the log file.

Variable name	**Description**
lock_row_waits	Number of times transactions had to wait for a locks.
lock_row_waiting	Number of transactions waiting for locks.
lock_total_wait_time_in_secs	Lock wait time in seconds.
lock_wait_time_avg_in_secs	Average lock wait time in seconds
lock_max_wait_time_in_secs	Maximum time in seconds that a transaction has waited on a lock.
row_total_read	Number of rows accessed or read.
row_total_inserted	Number of rows inserted.
row_total_updated	Number of rows updated.
row_total_deleted	Number of rows deleted.
read_ahead_sequential	Number of times the linear read ahead was invoked (not currently implemented)
page_size	InnoDB page size.
have_atomic_builtins	Non-zero if InnoDB was compiled with atomic ops.

Chapter 10. Transferring Databases between MySQL and Embedded InnoDB

In Section 1.1.3, “ Comparison with MySQL ”, you can learn why you might take a MySQL database and manipulate its InnoDB tables with an Embedded InnoDB program, or vice versa. Here are the details for that process.

10.1. Transferring Databases from MySQL to Embedded InnoDB

This technique is useful for using SQL statements in MySQL to set up tables and indexes, that you can then process with an Embedded InnoDB application, for maximum speed with all data manipulation operations. You can also verify compatibility of your application with any data types and data values used in a full MySQL application.

Follow these steps to transfer the data for all the databases within a data directory from MySQL to Embedded InnoDB:

1. All the tables that you will be able to access through Embedded InnoDB must be InnoDB tables, either created with the `ENGINE=INNODB` clause or with InnoDB as the default storage engine. (To verify which storage engine is used for each table, issue the command `show table status` in the `mysql` command processor.) If you intend to use Embedded InnoDB to access a table that uses the MyISAM or other storage engine, you can turn it into an InnoDB table with the command:

   ```
   alter table table_name engine=innodb;
   ```

   or clone it to an InnoDB table with the command:

   ```
   create table new_table engine=innodb as select * from old_table;
   ```

2. The InnoDB tables can either be part of the system tablespace, or in their own `.ibd` files (that is, created under the `innodb_file_per_table` option).
3. Shut down the MySQL database server that is accessing the data directory.
4. If you want to permanently transfer the data for use by an Embedded InnoDB application, recursively copy the MySQL data directory to a new location. If you only want to use the data in a Embedded InnoDB application temporarily, you can run the program and access the data in the original MySQL data directory, then restart the MySQL server after the program ends.
5. In your application, skip the table creation step and proceed directly to opening the table.
6. Read, insert, or update columns using the Embedded InnoDB constants to decode the column type information, using the mapping information from Section 6.2.4, “Column Types”.

10.2. Transferring Databases from Embedded InnoDB to MySQL

This technique is useful for running SQL statements in MySQL for one-time setup or periodic verification of the data used in an Embedded InnoDB application. For example, you might auto-generate a large set of queries to serve as a test suite, more easily than writing all the same queries with the Embedded InnoDB API. You can also this technique to pull data produced in a performance-critical environment such as real-time data capture, into MySQL for later analysis through existing reports.

Follow these steps to transfer the data for all the databases within a data directory from Embedded InnoDB to MySQL:

1. Shut down the MySQL database before doing the actual transfer.
2. Recursively copy the Embedded InnoDB database directory to the MySQL data directory.
3. Start the `mysqld` server.
4. Verify that the appropriate InnoDB tables are recognized by issuing `show table status` in the `mysql` command processor.
5. Verify the data types of the columns by issuing `describe` *`table_name`* in the `mysql` command processor.
6. Run the reports, test suite, benchmarks, and so on from MySQL as usual.

The operations on the MySQL side might require the `.frm` files associated with the tables. For that reason, you might initially create the tables in MySQL and transfer the database from MySQL to Embedded InnoDB, and later back again, so that the `.frm` files are always present.

Chapter 11. Embedded InnoDB Change History

The complete change history of the Embedded InnoDB can be viewed in the file `ChangeLog` that is included in the source and binary distributions as well as at the InnoDB website [http://www.innodb.com/changelogs/embedded_innodb/ChangeLog-1.0].

The 1.0.x numbers are not entirely sequential because they correspond to version numbers of the InnoDB Plugin, and in some cases a version number of the Plugin was never released, or we did not release a corresponding version of Embedded InnoDB.

11.1. Changes in API 1.0.6.6750

Add a transaction handle argument to ib_table_drop() and ib_index_drop().

Add an API function to check whether a transaction currently holds a shared lock on the InnoDB data dictionary.

Add an API function to retrieve all configuration variables.

Normalize the table name argument specified by client. The table names are converted to lower case on Windows.

Check for reserved characters in DOS/Windows when validating tablenames.

Add IB_DECIMAL column type.

Do not attempt to access a clustered index record that has been marked for deletion, on READ UNCOMMITTED isolation level.

Remove API function ib_table_schema_set_temp_dir, which is a no-op.

Display the zlib version number at startup.

Fix a bug in row fetch to return the correct version of a record from the transaction's view.

Speed up search by not fetching the next N records when doing a unique search.

Add a new error code, DB_FATAL.

Introduce a new boolean configuration parameter "adaptive_flushing" which can be set to OFF to disable the adaptive flushing.

Fix a memory leak.

11.2. Changes in API 1.0.4.5940 (Not Released)

Enforce NOT NULL column constraint during insert and update.

Fix a bug in the merge sort that can corrupt indexes in fast index creation. Add some consistency checks. Check that the number of records remains constant in every merge sort pass.

Make it possible to tune the buffer pool LRU eviction policy to be more resistant against index scans. Introduce the settable global variables lru_old_blocks_pct and lru_block_access_recency for controlling the buffer pool eviction policy. The parameter lru_old_blocks_pct (5..95) controls the desired amount of "old" blocks in the LRU list. The default is 37, corresponding to the old fixed ratio of 3/8. Each time a block is accessed, it will be moved to the "new" blocks if its first access was at least lru_block_access_recency milliseconds ago (default 0, meaning every block). The idea is that in index scans, blocks will be accessed a few times within innodb_old_blocks_time, and they will remain in the "old" section of the LRU list. Thus, when lru_block_access_recency is nonzero, blocks retrieved for one-time index scans will be more likely candidates for eviction than blocks that are accessed in random patterns.

Enable the VARBINARY column type

Add a new function (ib_status_get_i64()) to the API that allows users to read the InnoDB system variables.

Support inlining of functions and prefetch with Sun Studio. These changes are based on contribution from Sun Microsystems Inc. under a BSD license.

Change the defaults for: sync_spin_loops: 20 -> 30 thread_sleep_delay: 5 -> 6

Implement the adaptive flushing of dirty pages, which uses a heuristics based flushing rate of dirty pages to avoid IO bursts at checkpoint.

Implement IO capacity tuning. The ibuf merge is also changed from synchronous to asynchronous. These changes are based on contribution from Google Inc. under a BSD license.

11.3. Changes in API 1.0.3.5325 (June 24, 2009)

Do not rollback the transaction if the table create fails. Let the user handle the transaction rollback.

Return a meaningful error code when creating a table if the table name, supplied by the user, is not in 'database/table_name' format instead of asserting.

Add a function to return the API version number. The version number format is a 64-bit unsigned integer encoded as four 16-bit fields, *`reserved|current|revision|age`*:

- If the library source code has changed at all since the last release, then revision will be incremented (“*`current|revision|age`*” becomes “*`current|revision+1|age`*”)
- If any interfaces have been added, removed, or changed since the last update, *`current`* will be incremented, and *`revision`* will be set to `0`.
- If any interfaces have been added (but not changed or removed) since the last release, then `age` is incremented.
- If any interfaces have been changed or removed since the last release, then `age` is set to `0`.

Change ib_database_create() to create the database sub-directory relative to the data_home_dir setting.

Change the ib_cfg_set*() functions to return ib_err_t instead of ib_bool_t.

Fix a bug in the 1.0.0 release were INTs were stored incorrectly. The sign flag was being set incorrectly when converting the value to the storage format.

Change the default value of `log_group_home_dir` from “log” to “.”.

Introduce two new types:

IB_CHAR_ANYCHARSET	Fixed width column with any charset
IB_VARCHAR_ANYCHARSET	Variable length column, any charset

Add an API function to list all tables in the data dictionary

Add a function to drop a database. All the tables are first dropped and then the database. If the tables are in use then the referenced tables will be put in a background drop queue. The tables in the background drop queue will be dropped once the reference count for that table reaches 0.

Write updates of PAGE_MAX_TRX_ID to the redo log and add debug assertions for checking that PAGE_MAX_TRX_ID is valid on leaf pages of secondary indexes and the insert buffer B-tree. This bug could cause failures in secondary index lookups in consistent reads right after crash recovery.

Correctly estimate the space needed on the compressed page when performing an update by delete-and-insert.

Remove unused variable.

Fix Bug#44320 InnoDB: missing DB_ROLL_PTR in Table Monitor COLUMNS output

Fix a bug that prevented creation of compressed tables.

Add a function to convert error codes to strings, similar to strerror(3).

Add row caching when fetching rows with row_search_for_client(). This change will fetch more than one row and cache it in the row_prebuilt_t structure to reduce CPU and locking overhead. This port from the plugin branch is different because in Embedded InnoDB we cache the rows in InnoDB row format, in the plugin we cache the rows in the MySQL row format.

Make the insert row code more efficient, we do a shallow copy when inserting rows and cache the dtuple_t instance in the query graph rather than creating it for each insert.

Aggressive inlining of some frequently called API functions. The ones that are called from within the API implementation.

Add row caching when fetching rows with row_search_for_client()

Fix a bug when retrieving a string configuration parameters with ib_cfg_get() [http://forums.innodb.com/read.php?8,584,584#msg-584]

Remove the dependency on the config.h file generated by autoconf and CMake. API code assumes C99 compiler to work out the integer width. We make an exception for VisualStudio by using the __int8 etc. instead of the standard aliases uint8_t etc.

Replace the public interface file api0api.h with innodb.h.

Allow users to set a callback function and the output stream to print the InnoDB error messages. Currently this callback function must be a drop-in replacement for fprintf().

Define the logical type names trx_id_t, roll_ptr_t, and undo_no_t and use them in place of dulint everywhere.

11.4. Embedded InnoDB 1.0.0 (April 21, 2009)

Initial early adopter release.

Appendix A. Third-Party Software

Innobase Oy acknowledges that certain Third Party and Open Source software has been used to develop or is incorporated in InnoDB (including the Embedded InnoDB). This appendix includes required third-party license information.

A.1. Google SMP Patch

Innobase Oy gratefully acknowledges the contributions of Google, Inc. to improve performance by replacing InnoDB's use of Pthreads mutexes with calls to GCC atomic builtins. This change means that InnoDB mutex and and rw-lock operations take less CPU time, and improves throughput on those platforms where the atomic operations are available.

Changes from the Google contribution were incorporated in the following source code files: `btr0cur.c`, `btr0sea.c`, `buf0buf.c`, `buf0buf.ic`, `os0sync.h`, `row0sel.c`, `srv0srv.c`, `srv0srv.h`, `srv0start.c`, `sync0arr.c`, `sync0rw.c`, `sync0rw.h`, `sync0rw.ic`, `sync0sync.c`, `sync0sync.h`, `sync0sync.ic`, and `univ.i`.

These contributions are incorporated subject to the conditions contained in the file `COPYING.Google`, which are reproduced here.

```
Copyright (c) 2008, Google Inc.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials
      provided with the distribution.
    * Neither the name of the Google Inc. nor the names of its
      contributors may be used to endorse or promote products
      derived from this software without specific prior written
      permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

A.2. Multiple Background I/O Threads Patch from Percona

Innobase Oy gratefully acknowledges the contribution of Percona, Inc. to improve Embedded InnoDB performance by implementing configurable background threads. Embedded InnoDB uses background threads to service various types of I/O requests. The change provides another way to make Embedded InnoDB more scalable on high end systems.

Changes from the Percona, Inc. contribution were incorporated in the following source code files: `os0file.c`, `os0file.h`, `srv0srv.c`, `srv0srv.h`, and `srv0start.c`.

This contribution is incorporated subject to the conditions contained in the file `COPYING.Percona`, which are reproduced here.

```
# Copyright (c) 2008, 2009, Percona Inc.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#     * Redistributions of source code must retain the above copyright
#       notice, this list of conditions and the following disclaimer.
#     * Redistributions in binary form must reproduce the above
#       copyright notice, this list of conditions and the following
#       disclaimer in the documentation and/or other materials
#       provided with the distribution.
#     * Neither the name of the Percona Inc. nor the names of its
#       contributors may be used to endorse or promote products
#       derived from this software without specific prior written
#       permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
```

A.3. Performance Patches from Sun Microsystems

Innobase Oy gratefully acknowledges the following contributions from Sun Microsystems, Inc. to improve Embedded InnoDB performance:

- Introducing the PAUSE instruction inside spin loops, This change increases performance in high concurrency, CPU-bound workloads.
- Enabling inlining of functions and prefetch with Sun Studio.

Changes from the Sun Microsystems, Inc. contribution were incorporated in the following source code files: `univ.i`, `ut0ut.c`, and `ut0ut.h`.

This contribution is incorporated subject to the conditions contained in the file `COPYING.Sun_Microsystems`, which are reproduced here.

```
# Copyright (c) 2009, Sun Microsystems, Inc.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#     * Redistributions of source code must retain the above copyright
#       notice, this list of conditions and the following disclaimer.
#     * Redistributions in binary form must reproduce the above
#       copyright notice, this list of conditions and the following
#       disclaimer in the documentation and/or other materials
#       provided with the distribution.
#     * Neither the name of Sun Microsystems, Inc. nor the names of its
#       contributors may be used to endorse or promote products
#       derived from this software without specific prior written
#       permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
```

Appendix B. Code Examples Supplied with Embedded InnoDB

This section describes the code examples shipped in the Embedded InnoDB `examples/` subdirectory, and lists the particular features illustrated by each one.

On Linux and UNIX systems, these examples are installed in the directory `/usr/local/share/embedded_innodb-1.0/examples`. On Windows systems, these examples are installed in the directory `embedded_innodb-1.0\examples`.

`ib_cfg.c`
: This example demonstrates how to configure Embedded InnoDB using the API. This program also uses a very simple config file parser to read the configuration values from a file. See test0aux.c.

`ib_compressed.c`
: Example that creates a few compressed tables and checks for valid page_size.

`ib_cursor.c`
: Single-threaded example that does the equivalent of:

```
-- Create a database
 CREATE TABLE T(c1 INT, PRIMARY KEY(c1));
 INSERT INTO T VALUES(1); ...
 SELECT * FROM T;
 SELECT * FROM T WHERE c1 = 5;
 SELECT * FROM T WHERE c1 > 5;
 SELECT * FROM T WHERE c1 < 5;
 SELECT * FROM T WHERE c1 >= 1 AND c1 < 5;
 DROP TABLE T;
```

`ib_index.c`
: Single-threaded example that does the equivalent of:

```
-- Create a database
 CREATE TABLE T(c1 VARCHAR(n), c2 INT, PRIMARY KEY(c1, 4));
-- Should succeed.
 INSERT INTO T VALUES('xxxxaaaa', 1);
-- Should result in duplicate key error
 INSERT INTO T VALUES('xxxxbbbb', 2);
```

This example demonstrates how to create indexes with a prefix length. Note how Embedded InnoDB uses only the first 4 characters to match on key value.

`ib_logger.c`
: This example demonstrates how to redirect Embedded InnoDB messages to a user defined function, in this case to perform startup and shutdown silently. It creates a message logging function `null_logger()` that does not print anything, and sets this function as the default InnoDB message logger. This function should not print any messages to stderr. The example program simply does a startup and shutdown of InnoDB.

`ib_mt_stress.c`
: Available on Linux and UNIX only, not on Windows systems. This is a multi-threaded example that does the equivalent of:

```
-- Create a database
 CREATE TABLE blobt3(
 A       INT,
 D       INT,
 B       BLOB,
 C       TEXT,
 PRIMARY KEY(B(10), A, D),
 INDEX(D),
 INDEX(A),
 INDEX(C(255), B(255)),
 INDEX(B(5), C(10), A));
-- Insert random rows into blobt3.
-- In separate threads, do the following:
 INSERT INTO blobt3 VALUES(
        <random_integer>,
        5,
        <random_string>,
        <random_string>);
 UPDATE blobt3 SET B = <random_string> WHERE A = <random_integer>;
```

The test creates 4 types of worker threads that issue insert, update, delete and select operations. This is a complex example that also doubles as a stress test.

`ib_perf1.c` Available on Linux and UNIX only, not on Windows systems. This is a multi-threaded example that does the equivalent of:

```
-- Create a database
 CREATE TABLE Tn1(c1 INT AUTOINCREMENT, c2 INT, PRIMARY KEY(c1));
 CREATE TABLE Tn2(c1 INT AUTOINCREMENT, c2 INT, PRIMARY KEY(c1));
 INSERT N million rows into Tn1;
 INSERT INTO Tn2 SELECT * FROM Tn1;
 SELECT COUNT(*) FROM Tn1, Tn2 WHERE Tn1.c1 = Tn2.c1;
```

Since Embedded InnoDB does not support autoincrement, the program increments the value of `c1` explicitly.

This example also doubles as a test to measure the internal speed of the database engine. We start N threads, each running the above test, but on a self-contained set of tables independent of each other. The objective of the test is to measure any unnecessary locking issues/contention for resources.

`ib_search.c` This example demonstrates how to use the Embedded InnoDB search modes.

```
-- Create a database
 CREATE TABLE T(c1 VARCHAR(n), c2 VARCHAR(n), c3 INT,
               PRIMARY KEY(c1, c2));
 INSERT INTO T VALUES('abc', 'def', 1);
 INSERT INTO T VALUES('abc', 'zzz', 1);
 INSERT INTO T VALUES('ghi', 'jkl', 2);
 INSERT INTO T VALUES('mno', 'pqr', 3);
 INSERT INTO T VALUES('mno', 'xxx', 3);
 INSERT INTO T VALUES('stu', 'vwx', 4);
 SELECT * FROM T WHERE c1 = 'abc' AND c2 = 'def';
 SELECT * FROM T WHERE c1 = 'abc';
 SELECT * FROM T WHERE c1 >= 'g%';
 SELECT * FROM T WHERE c1 = 'mno' AND c2 >= 'x%';
 SELECT * FROM T WHERE c1 = 'mno' AND c2 >= 'z%';
 DROP TABLE T;
```

`ib_status.c` This example demonstrates how to check the Embedded InnoDB status variables. To see how many `fsync()` requests have been issued by Embedded InnoDB, you can check the value of the `fsync_req_done` status variable using `ib_status_get_i64()`.

`ib_test1.c` This example does the equivalent of:

```
-- Create a database
 CREATE TABLE T(c1 VARCHAR(n), c2 VARCHAR(n), c3 INT,
               PRIMARY KEY(first, last));
 INSERT INTO T VALUES('x', 'y', 1); ...
 SELECT * FROM T;
 UPDATE T SET c3 = c3 + 100 WHERE c1 = 'x';
 SELECT * FROM T;
 DELETE FROM T WHERE c1 = 'x';
 SELECT * FROM T;
 DROP TABLE T;
```

`ib_test2.c` This example does the equivalent of:

```
-- Create a database
  FOR 1 TO 100
   CREATE TABLE T(c1 VARCHAR(128), c2 BLOB, c3 INT,
                PRIMARY KEY(c1));
   FOR 1 TO 10
    BEGIN;
     FOR 1 TO 100
      INSERT INTO T VALUES(RANDOM(STRING), RANDOM(STRING), 0);
     END
     UPDATE T SET c1 = RANDOM(string), c3 = c3 + 1
              WHERE c1 = RANDOM(STRING);
    COMMIT;
   END
   DROP TABLE T;
  END
```

`ib_test3.c` An example to demonstrate the typed API of Embedded InnoDB.

`ib_test5.c` This example is similar to ib_test2.c, except that it demonstrates how to access the clustered index record via a secondary index.

`ib_types.c` This example demonstrates how to read and write the different column types supported by Embedded InnoDB. It does the equivalent of of:

```
 CREATE TABLE T(
        c1 VARCHAR(n),
        c2 INT NOT NULL,
        c3 FLOAT,
        c4 DOUBLE,
        c5 BLOB,
        PRIMARY KEY(c1));
 INSERT INTO T VALUES('x', 1, 2.0, 3.0, 'xxx'); ...
 SELECT * FROM T;
 DROP TABLE T;
```

`test0aux.c` This is an auxiliary file with support functions that are used by the other example programs.

`test0aux.h` This is the common header file that is included by all the example programs.

`ib_update.c` This example demonstrates how to update all records in a table by scanning the table from start to end, the equivalent of:

```
-- Create a database
 CREATE TABLE t(c1 INT, c2 VARCHAR(10), PRIMARY KEY (c1));
 FOR I IN 1 ... 10 STEP 2
  INSERT INTO t VALUES(I, CHAR('a' + I)); ...
 END
 SELECT * FROM t;
 UPDATE t SET c1 = c1 / 2;
 SELECT * FROM t;
 DROP TABLE t;
```

Embedded InnoDB Glossary

These terms are commonly used in information about the Embedded InnoDB product and InnoDB in general.

A

ACID	An acronym standing for atomicity, consistency, isolation, and durability. These properties are all desirable in a database system, and are all closely tied to the notion of a **transaction**. The transactional features of InnoDB adhere to the ACID principles. Transactions are atomic units of work that can be committed or rolled back. When a transaction makes multiple changes to the database, either all the changes succeed when the transaction is committed, or all the changes are undone when the transaction is rolled back. The database remains in a consistent state at all times -- after each commit or rollback, and while transactions are in progress. Transactions are protected (isolated) from each other while they are in progress; they cannot interfere with each other or see each other's uncommitted data. This isolation is achieved through the notion of **locking**. (Expert users can adjust the **isolation level**, trading off less protection in favor of increased performance, when they can be sure that the transactions really do not interfere with each other.) The results of transactions are durable, in that once a commit operation succeeds, the changes made by that transaction are safe from power failures, system crashes, race conditions, or other potential dangers that many non-database applications are vulnerable to. Durability typically involves writing to disk storage, with a certain amount of redundancy to protect against power failures or software crashes during write operations. (In InnoDB, the **doublewrite buffer** assists with durability.) See Also transaction, commit, rollback, locking, isolation level, doublewrite buffer.
Antelope	The code name for the original InnoDB **file format**. It supports the **redundant** and **compact** row formats, but not the newer **dynamic** and **compressed** row formats available in the **Barracuda** file format. You can select the file format to use through the `innodb_file_format` option. See Also Barracuda, file format, ibdata file, redundant row format, compact row format, dynamic row format, compressed row format, row format, innodb_file_format.
application programming interface (API)	A set of functions and/or procedures. An API provides a stable set of names and types for functions, procedures, parameters, and return values. To work with InnoDB at the API level typically involves building an application with the Embedded InnoDB library. See Also Embedded InnoDB.
auto-increment	A property of a table column (specified by the `AUTO_INCREMENT` keyword) that automatically adds an ascending sequence of values in the column. InnoDB supports auto-increment only for primary key columns. The auto-increment feature is available in the built-in InnoDB storage engine, and in the InnoDB Plugin. However, it is not available in Embedded InnoDB applications.

It saves work for the developer, not to have to produce new unique values when inserting new rows. It provides useful information for the query optimizer, because the column is known to be not null and with unique values. The values from such a column can be used as lookup keys in various contexts, and because they are auto-generated there is no reason to ever change them; for this reason, primary key columns are often specified as auto-incrementing.

Auto-increment columns can be problematic with statement-based replication, because replaying the statements on a slave might not produce the same set of column values as on the master, due to timing issues. When you have an auto-incrementing primary key, you can use statement-based replication only with the setting `innodb_autoinc_lock_mode=1`. If you have `innodb_autoinc_lock_mode=2`, which allows higher concurrency for insert operations, use **row-based replication** rather than **statement-based replication**. The setting `innodb_autoinc_lock_mode=0` is the previous (traditional) default setting and should not be used except for compatibility purposes.
See Also auto-increment locking, statement-based replication, row-based replication, innodb_autoinc_lock_mode, row-based replication, statement-based replication.

auto-increment locking

The convenience of an **auto-increment** primary key involves some tradeoff with concurrency. In the simplest case, if one transaction is inserting values into the table, any other transactions must wait to do their own inserts, so that rows inserted by the first transaction receive consecutive primary key values. InnoDB includes optimizations, and the `innodb_autoinc_lock_mode` option, so that you can choose how to trade off between predictable sequences of auto-increment values and maximum **concurrency** for insert operations.
See Also auto-increment, concurrency, innodb_autoinc_lock_mode.

B

B-tree

A tree data structure that is popular for use in database indexes. The structure is kept sorted at all times, enabling fast lookup for exact matches and ranges (for example, greater than, less than, and `BETWEEN` operators).

Because B-tree nodes can have many children, a B-tree is not the same as a binary tree, which is limited to 2 children per node.

Barracuda

The code name for an InnoDB **file format** that supports compression for table data. This file format was first introduced in the InnoDB Plugin. It supports the **dynamic** and **compressed** row formats. You can select it through the `innodb_file_format` option.

For maximum compatibility with older InnoDB tables, the default file format remains **Antelope** and the default row format remains **compact**, for the built-in InnoDB storage engine, the InnoDB Plugin, and Embedded InnoDB applications.
See Also Antelope, file format, ibdata file, row format, compact row format, compressed row format, dynamic row format, innodb_file_format.

beta

An early stage in the life of a software product, when it is available only for evaluation, typically without a definite release number or a number less than 1. InnoDB does not use the beta designation, preferring an **early adopter** phase that can extend over several point releases, leading to a **GA** release.
See Also early adopter, GA.

buffer pool

The memory area that holds cached data, read from both tables and indexes. For efficiency of high-volume read operations, the buffer pool is divided into **pages** that can potentially hold multiple rows. For efficiency of cache management, the buffer pool is implemented as a linked list

	of pages, so that data that is rarely used can be aged out of the cache, using a variation of the **LRU** algorithm. See Also page, LRU.

C

clustered index	The InnoDB term for a **primary key** index. This special term signifies that the table storage is organized based on the values of the primary key columns. (In the Oracle Database product, this type of table is known as an **index-organized table**.) See Also index, primary key, secondary index.
column index	An **index** on a single column. See Also index, concatenated index.
column prefix	When an index is created with a length specification, such as `CREATE INDEX idx ON t1 (c1(N))`, only the first N characters of the column value are stored in the index. Keeping the index prefix small makes the index compact, and the memory and disk I/O savings help performance. (Although making the index prefix too small can hinder query optimization by making rows with different values appear to the query optimizer to be duplicates.) For columns containing binary values or long text strings, where sorting is not a major consideration and storing the entire value in the index would waste space, the index automatically uses the first N (typically 768) characters of the value to do lookups and sorts. See Also index.
commit	A **SQL** statement that ends a **transaction**, making permanent any changes made by the transaction. It is the opposite of **rollback**, which undoes any changes made in the transaction. InnoDB uses an **optimistic** mechanism for commits, so that changes can be written to the data files before the commit actually occurs. This technique makes the commit itself faster, with the tradeoff that more work is required in case of a rollback. By default, MySQL uses the **auto-commit** setting, which automatically issues a commit following each SQL statement. See Also rollback, SQL, transaction.
compact row format	The default InnoDB **row format** since MySQL 5.0.3. It has a more compact representation for nulls and variable-length fields than the prior default (**redundant row format**). Because of the **B-tree** indexes that make row lookups so fast in InnoDB, there is little if any performance benefit to keeping all rows the same size. See Also row format, redundant row format.
compressed row format	A **row format** introduced in the InnoDB Plugin, available as part of the **Barracuda** file format. Large fields are stored away from the page that holds the rest of the row data, as in dynamic row format. Both index pages and the large fields are compressed, yielding memory and disk savings. Depending on the structure of the data, the decrease in memory and disk usage might or might not outweigh the performance overhead of uncompressing the data as it is used. Support for compression can be turned on or off for the **Embedded InnoDB** product as a build-time option. See Also row format, dynamic row format, Barracuda, Embedded InnoDB.
concatenated index	An **index** that includes multiple columns.

	See Also index, index prefix.
concurrency	The ability of multiple operations (in database terminology, **transactions**) to run simultaneously, without interfering with each other. Concurrency is also involved with performance, because ideally the protection for multiple simultaneous transactions works with a minimum of performance overhead, using efficient mechanisms for **locking**. See Also transaction, ACID, locking.
consistent read	A read operation that uses snapshot information to present query results based on a point in time, regardless of changes performed by other transactions running at the same time. If queried data has been changed by another transaction, the original data is reconstructed based on the contents of the **undo log**. This technique avoids some of the **locking** issues that can reduce **concurrency** by forcing transactions to wait for other transactions to finish. With the **repeatable read** isolation level, the snapshot is based on the time when the first read operation is performed. With the **read committed** isolation level, the snapshot is reset to the time of each consistent read operation. Consistent read is the default mode in which InnoDB processes `SELECT` statements in **READ COMMITTED** and **REPEATABLE READ** isolation levels. A consistent read does not set any locks on the tables it accesses, and therefore other sessions are free to modify those tables at the same time a consistent read is being performed on the table. See Also undo log, concurrency, locking, isolation level, transaction, ACID, MVCC, repeatable read, read committed, read uncommitted, serializable read.
covering index	An **index** that includes all the columns in a query. It could be a **column index** or a **concatenated index**. See Also index, column index, concatenated index.
crash recovery	The cleanup activities that occur when InnoDB is started again after a crash. Changes that were committed before the crash, but not yet written into the tablespace files, are reconstructed from the **doublewrite buffer**. When the database is shut down normally, this type of activity is performed during shutdown by the **purge** operation. During normal operation, committed data can be stored in the insert buffer for a period of time before being written to the tablespace files. There is always a tradeoff between keeping the tablespace files up-to-date, which introduces performance overhead during normal operation, and buffering the data, which can make shutdown and crash recovery take longer. See Also purge, doublewrite buffer, insert buffer.
cursor	An internal data structure that is used to represent the result set of a query, or other operation that performs a search using a SQL `WHERE` clause. It works like an iterator in other high-level languages, producing each value from the result set as requested. Although usually SQL handles the processing of cursors for you, you might delve into the inner workings when dealing with performance-critical code. Embedded InnoDB requires you to be familiar with cursors and to manipulate them using C API calls. See Also query, Embedded InnoDB.
cursor match mode	In Embedded InnoDB, a setting that controls prefetching and locking when retrieving records through a cursor. See Also Embedded InnoDB.
cursor search mode	In Embedded InnoDB, a setting that determines how precisely a search condition must match an index key before a comparison succeeds.

See Also Embedded InnoDB.

D

data definition language	See DDL.
data dictionary	A set of tables, controlled by the InnoDB storage engine, that keeps track of InnoDB-related objects such as tables, indexes, and table columns. These tables are part of the **system tablespace**. Because the InnoDB Hot Backup product always backs up the system tablespace, all backups include the contents of the data dictionary. See Also system tablespace.
data manipulation language	See DML.
data files	The files that physically contain the InnoDB table and index data. There can be a one-to-many relationship between data files and tables, as in the case of the **system tablespace**, which can hold multiple InnoDB tables as well as the **data dictionary**. There can also be a one-to-one relationship between data files and tables, as when the **file-per-table** setting is enabled, causing each newly created table to be stored in a separate **tablespace**. In Embedded InnoDB, data files are stored by default underneath the current working directory of the application. You can specify a different location by calling the configuration APIs. See Also Embedded InnoDB, tablespace, system tablespace, data dictionary, file-per-table.
database	An InnoDB database is largely defined by its **data files**. For long-time MySQL users, a database is a familiar notion. Users coming from an Oracle background will find that the MySQL meaning of a database is closer to what Oracle calls a schema. The **Embedded InnoDB** product maps a database to a single directory. See Also data files, Embedded InnoDB.
DDL	Data definition language, a set of **SQL** statements for manipulating the database itself rather than individual table rows. Includes all forms of the `CREATE`, `ALTER`, and `DROP` statements. Also includes the `TRUNCATE` statement, because it works differently than a `DELETE FROM` *`table_name`* statement, even though the ultimate effect is similar. DDL statements automatically **commit** the current **transaction**; they cannot be **rolled back**. The API of the **Embedded InnoDB** product currently supports `CREATE` and `DROP` operations, but not `ALTER` operations. See Also SQL, transaction, commit, rollback, Embedded InnoDB.
deadlock	A situation where different **transactions** are unable to proceed, because each holds a **lock** that the other needs. Because both transactions are waiting for a resource to become available, neither will ever release the locks it holds. A deadlock can occur when the transactions acquire locks on multiple tables, but in the opposite order. A deadlock can also occur when statements such as `UPDATE` or `SELECT ... FOR UPDATE` lock ranges of index records and **gaps**, with each transaction acquiring some locks but not others due to a timing issue. To reduce the possibility of deadlocks, use transactions rather than `LOCK TABLE` statements; keep transactions that insert or update data small enough that they do not stay open for long

periods of time; when different transactions update multiple tables or large ranges of rows, use the same order of operations (such as `SELECT ... FOR UPDATE`) in each transaction; create indexes on the columns used in `SELECT ... FOR UPDATE` and `UPDATE ... WHERE` statements. The possibility of deadlocks is not affected by the **isolation level**, because the isolation level changes the behavior of read operations, while deadlocks occur because of write operations.

If a deadlock does occur, InnoDB detects the condition and **rolls back** one of the transactions (the **victim**). Thus, even if your application logic is perfectly correct, you must still handle the case where a transaction must be retried. To monitor how frequently deadlocks occur, use the command `SHOW ENGINE INNODB STATUS`.
See Also lock, locking, gap, concurrency, isolation level, transaction, rollback.

delete

When InnoDB processes a `DELETE` statement, the rows are immediately marked for deletion and no longer are returned by queries. The storage is reclaimed sometime later, during the periodic garbage collection known as the **purge** operation, performed by a separate thread.
See Also purge.

dirty read

An operation that retrieves unreliable data, data that was updated by another transaction but not yet **committed**. It is only possible with the **isolation level** known as **read uncommitted**.

This kind of operation does not adhere to the **ACID** principle of database design. It is considered very risky, because the data could be **rolled back**, or updated further before being committed; then, the transaction doing the dirty read would be using data that was never confirmed as accurate.

Its polar opposite is **consistent read**, where InnoDB goes to great lengths to ensure that a transaction does not read information updated by another transaction, even if the other transaction commits in the meantime.
See Also commit, rollback, isolation level, ACID, consistent read, read uncommitted, read committed.

DML

Data manipulation language, a set of **SQL** statements for performing insert, update, and delete operations. The `SELECT` statement is sometimes considered as a DML statement, because the `SELECT ... FOR UPDATE` form is subject to the same considerations for **locking** as `INSERT`, `UPDATE`, and `DELETE`.

DML statements operate in the context of a **transaction**, so their effects can be **committed** or **rolled back** as a single unit.

In the **Embedded InnoDB** product, the equivalent operations are performed using API calls rather than SQL statements.
See Also SQL, locking, transaction, commit, rollback, Embedded InnoDB.

doublewrite buffer

InnoDB uses a novel file flush technique called doublewrite. Before writing **pages** to a data file, InnoDB first writes them to a contiguous area called the doublewrite buffer. Only after the write and the flush to the doublewrite buffer have completed, does InnoDB write the pages to their proper positions in the data file. If the operating system crashes in the middle of a page write, InnoDB can later find a good copy of the page from the doublewrite buffer during **crash recovery**.

Although data is always written twice, the doublewrite buffer does not require twice as much I/O overhead or twice as many I/O operations. Data is written to the buffer itself as a large sequential chunk, with a single `fsync` call to the operating system.

The doublewrite buffer can be turned off by specifying the option `innodb_doublewrite=0`.

See Also crash recovery, purge.

dynamic row format

A row format introduced in the InnoDB Plugin, available as part of the **Barracuda** file format. Because `TEXT` and `BLOB` fields are stored outside of the rest of the page that holds the row data, it is very efficient for rows that include large objects, resulting in fewer I/O operations when the large fields do not need to be accessed.
See Also row format, Barracuda.

E

early adopter

A stage similar to beta, when a software product is typically evaluated for performance, functionality, and compatibility in a non-mission-critical setting. InnoDB uses the **early adopter** designation rather than **beta**, through a succession of point releases leading up to a **GA** release.
See Also beta, GA.

Embedded InnoDB

A product that allows the capabilities of the InnoDB storage engine to be embedded within an application program, the same as any other library, without the need for a full MySQL installation. Instead of using SQL, the application manipulates data using C API calls. It is intended for applications that require maximal performance and minimal configuration and setup, such as in the embedded systems market.
See Also cursor.

eviction

The process of removing an item from a cache or other temporary storage area. Often, but not always, uses the the LRU algorithm to determine which item to remove.
See Also LRU.

exclusive lock

A kind of **lock** that prevents any other **transaction** from locking the same row. Depending on the transaction **isolation level**, this kind of lock might block other transactions from writing to the same row, or might also block other transactions from reading the same row. The default InnoDB isolation level, **REPEATABLE READ**, enables higher **concurrency** by allowing transactions to read rows that have exclusive locks, a technique known as **consistent read**.
See Also lock, transaction, isolation level, concurrency, repeatable read, consistent read, shared lock.

F

fast index creation

A capability first introduced in the InnoDB Plugin, that speeds up creation of secondary indexes by avoiding the need to completely rewrite the associated table. The speedup applies to dropping secondary indexes also.

Because index maintenance can add performance overhead to many data transfer operations, consider doing operations such as `ALTER TABLE ... ENGINE=INNODB` or `INSERT INTO ... SELECT * FROM ...` without any secondary indexes in place, and creating the indexes afterwards.

Even if you do not use the InnoDB Plugin as your primary storage engine, you can take advantage of this capability by enabling the Plugin temporarily, just to create or drop indexes, and then switch back to the built-in InnoDB storage engine for normal use.
See Also index, secondary index.

fast shutdown

A shutdown procedure that is required before installation of the InnoDB Plugin. From the MySQL command line, issue the following command before performing the shutdown:

```
SET GLOBAL innodb_fast_shutdown=0;
```

To make this type of shutdown the default, specify by the configuration parameter `innodb_fast_shutdown=0`.
See Also slow shutdown, shutdown.

file format
: The format used by InnoDB for its data files named `ibdata1`, `ibdata2`, and so on. Each file format supports one or more row formats.
See Also Antelope, Barracuda, ibdata file, row format.

file-per-table
: A general name for the setting controlled by the `innodb_file_per_table` option. For each table created while this setting is in effect, the data is stored in a separate file rather than in the system tablespace. When table data is stored in a separate file, you have more flexibility to choose non-default **file formats** and **row formats**, which are required for features such as data compression. The `TRUNCATE TABLE` operation is also much faster, and the reclaimed space can be used by the operating system rather than remaining reserved for InnoDB.

The InnoDB Hot Backup product is more flexible for tables that are in their own files. For example, tables can be excluded from a backup, but only if they are in separate files. Thus, this setting is suitable for tables that are backed up less frequently or on a different schedule.

In the **Embedded InnoDB** product, the options do not use the `innodb_` prefix, so the option name there is `file_per_table`.
See Also system tablespace, ibdata file, innodb_file_per_table, file format, row format, Embedded InnoDB.

fixed row format
: This row format is used by the MyISAM storage engine, not by InnoDB. If you create an InnoDB table with the option `row_format=fixed`, InnoDB translates this option to use the **compact row format** instead, although the `fixed` value might still show up in output such as `SHOW TABLE STATUS` reports.
See Also row format, compact row format.

full table scan
: An operation that requires reading the entire contents of a table, rather than just selected portions using an index. Typically performed either with small lookup tables, or in data warehousing situations with large tables where all available data is aggregated and analyzed. How frequently these operations occur, and the sizes of the tables relative to available memory, have implications for the algorithms used in query optimization and managing the buffer pool.

The purpose of **indexes** is to allow lookups for specific values or ranges of values within a large table, thus avoiding full table scans when practical.
See Also buffer pool, LRU, index.

G

GA
: 'Generally available', the stage when a software product leaves beta and is available for sale, official support, and production use.
See Also beta, early adopter.

gap
: A place in an **index** data structure where new values could be inserted. When you lock a set of rows with a statement such as `SELECT ... FOR UPDATE`, InnoDB can create locks that apply to the gaps as well as the actual values in the index. For example, if you select all values greater than 10 for update, a gap lock prevents another transaction from inserting a new value that is greater than 10.

Gap locks are part of the tradeoff between performance and **concurrency**, and are used in some transaction **isolation levels** and not others.

	See Also index, concurrency, isolation level, supremum record, infimum record.

H

handle	In the **Embedded InnoDB** product, special types that are returned by various create and open calls. You do not manipulate their contents directly. You just pass the returned handles as parameters to subsequent function calls. The Embedded InnoDB API defines handles to represent **tuples**, **transactions**, **cursors**, **table schemas**, and **index schemas**. See Also Embedded InnoDB, tuple, transaction, cursor, table schema, index schema.

I

ib_logfile	A set of files, typically named `ib_logfile0` and `ib_logfile1`, that form the **redo log**. These files record statements that attempt to change data in InnoDB tables. These statements are replayed automatically to correct data written by incomplete transactions, on startup following a crash. This data can not be used for manual recovery; for that type of operation, use the **binary log**. See Also redo log.
ibdata file	A set of files with names such as `ibdata1`, `ibdata2`, and so on, that make up the InnoDB **system tablespace**. These files contain metadata about InnoDB tables, and can contain some or all of the table data also (depending on whether the file-per-table option is in effect when each table is created). See Also Antelope, Barracuda, file format, file-per-table, system tablespace.
index	A data structure that provides a fast lookup capability for rows of a table, typically by forming a tree structure representing all the values of a particular column or set of columns. See Also clustered index, primary key, secondary index, B-tree, column index, concatenated index, covering index, partial index.
index prefix	In an **index** that applies to multiple columns (known as a **concatenated index**), the initial or leading columns of the index. A query that references the first 1, 2, 3, and so on columns of a **concatenated index** can use the index, even if the query does not reference all the columns in the index. See Also index, concatenated index.
index schema	In the **Embedded InnoDB** product, representation of the columns in an **index**. See Also Embedded InnoDB, index.
infimum record	A **pseudo-record** in an index, representing the **gap** below the smallest value in that index. If a transaction has a statement such as `SELECT ... FOR UPDATE ... WHERE col < 10;`, and the smallest value in the column is 5, it is a lock on the infimum record that prevents other transactions from inserting even smaller values such as 0, -10, and so on. See Also gap, supremum record, pseudo-record.
innodb_autoinc_lock_mode	The option that controls the algorithm used for **auto-increment locking**. When you have an auto-incrementing primary key, you can use statement-based replication only with the setting `innodb_autoinc_lock_mode=1`. This setting is known as **consecutive** lock mode, because multi-row inserts within a transaction receive consecutive auto-increment values. If you have `innodb_autoinc_lock_mode=2`, which allows higher concurrency for in-

	sert operations, use row-based replication rather than statement-based replication. This setting is known as **interleaved** lock mode, because multiple multi-row insert statements running at the same time can receive autoincrement values that are interleaved. The setting `innodb_autoinc_lock_mode=0` is the previous (traditional) default setting and should not be used except for compatibility purposes. See Also auto-increment locking.
innodb_file_format	A setting that determines the **file format** for all **tablespaces** created after you specify a value for this option. To create tablespaces other than the **system tablespace**, you must use the **file-per-table** option also. See Also file format, tablespace, system tablespace, Antelope, Barracuda, file-per-table, innodb_file_per_table.
innodb_file_per_table	The option that allows you to use the **file-per-table** setting, which stores newly created tables in their own data files, outside the **system tablespace**. This option is needed to take full advantage of many other features, such as such as table compression in the InnoDB Plugin, or backups of named tables in InnoDB Hot Backup. This option was once static, but can now be set using the `SET GLOBAL` command. See Also data files, system tablespace, file-per-table.
insert buffer	A special index data structure that holds values that are newly inserted into secondary indexes. These values could result from both SQL `INSERT` or `UPDATE` statements. Periodically (during times when the system is mostly idle, and during a normal shutdown or a slow shutdown), the new index values are written to disk, merged with the existing values. The merge operation can write the disk blocks for a series of index values more efficiently than if each value were written to disk immediately. The insert buffer is not used if the relevant page from the secondary index is already in the buffer pool, because that change can be made quickly in memory. If the relevant page from the secondary index is brought into the buffer pool while associated changes are still in the insert buffer, the changes for that index page are merged immediately. The insert buffer is not used if the secondary index is unique, because the uniqueness of new values cannot be verified before the new entries are written out. Physically, the insert buffer is part of the **system tablespace**, so that in theory, the new index data could remain buffered across database restarts. To see information about the current data in the insert buffer, issue the `SHOW INNODB STATUS` command. See Also crash recovery, purge, system tablespace.
intention exclusive lock	See intention lock.
intention lock	A kind of **lock** that applies to the table level, used to indicate what kind of lock the transaction intends to acquire on rows in the table. Different transactions can acquire different kinds of intention locks on the same table, but the first transaction to acquire an **intention exclusive** (IX) lock on a table prevents other transactions from acquiring any S or X locks on the table. Conversely, the first transaction to acquire an **intention shared** (IS) lock on a table prevents other transactions from acquiring any X locks on the table. The two-phase process allows the lock requests to be resolved in order, without blocking locks and corresponding operations that are compatible. See Also locking, lock mode, lock.
intention shared lock	See intention lock.

isolation level

One of the foundations of database processing. Isolation is the **I** in the acronym **ACID**; the isolation level is the setting that fine-tunes the balance between performance and reliability, consistency, and reproducibility of results when multiple **transactions** are making changes and performing queries at the same time.

From highest amount of consistency and protection to the least, the isolation levels supported by InnoDB are: **serializable read**, **repeatable read**, **consistent read**, and **read uncommitted**.

With the built-in InnoDB storage engine and the InnoDB Plugin, many users can keep the default isolation level (**repeatable read**) for all operations. Expert users might choose the **read committed** level as they push the boundaries of scalability with OLTP processing, or during data warehousing operations where minor inconsistencies do not affect the aggregate results of large amounts of data. The levels on the edges (**serializable read** and **read uncommitted**) change the processing behavior to such an extent that they are rarely used.

For applications that use the Embedded InnoDB product, the isolation level must be specified for each transaction. Because Embedded InnoDB is likely to be used in performance-critical situations, choosing the isolation level for each transaction can be an important part of the planning process.
See Also ACID, transaction, repeatable read, consistent read, serializable read, read uncommitted.

L

latch

A lightweight structure used internally by InnoDB to implement a **lock**. Often used interchangeably with **mutex**. Certain latches are the focus of performance tuning within the InnoDB storage engine, such as the **data dictionary** mutex.
See Also lock, locking, mutex, data dictionary.

lock

The high-level notion of an object that controls access to a resource, such as a table, row, or internal data structure, as part of a **locking** strategy. For intensive performance tuning, you might delve into the actual structures that implement locks, such as **mutexes** and **latches**.
See Also locking, lock mode, mutex, latch, Pthreads.

locking

The system of protecting a **transaction** from seeing or changing data that is being queried or changed by other transactions. The locking strategy must balance reliability and consistency of database operations (the principles of the **ACID** philosophy) against the performance needed for good **concurrency**. Fine-tuning the locking strategy often involves choosing an **isolation level** and ensuring all your database operations are safe and reliable for that isolation level.
See Also lock, isolation level, ACID, mutex, latch, transaction, concurrency.

lock mode

A shared (S) lock allows a transaction to read a row. Multiple transactions can acquire an S lock on that same row at the same time.

An exclusive (X) lock allows a transaction to update or delete a row. No other transaction can acquire any kind of lock on that same row at the same time.

Intention locks apply to the table level, and are used to indicate what kind of lock the transaction intends to acquire on rows in the table. Different transactions can acquire different kinds of intention locks on the same table, but the first transaction to acquire an intention exclusive (IX) lock on a table prevents other transactions from acquiring any S or X locks on the table. Conversely, the first transaction to acquire an intention shared (IS) lock on a table prevents other transactions from acquiring any X locks on the table. The two-phase process allows the lock requests to be resolved in order, without blocking locks and corresponding operations that are compatible.

See Also lock, locking, intention lock.

LRU

An acronym meaning “least recently used”, a common method for managing storage areas. The items that have not been used recently are removed, when space is needed to cache newer items. InnoDB uses the LRU mechanism by default to manage the **pages** within the **buffer pool**, but makes exceptions in cases where a page might be read only a single time, such as during a full-table scan. The amount that the buffer cache policy differs from the strict LRU algorithm is governed by the options `innodb_old_blocks_pct` and `innodb_old_blocks_time`.
See Also buffer pool, eviction.

M

multiversion concurrency control

See MVCC.

mutex

Informal abbreviation for “mutex variable”. The low-level data structure that InnoDB uses to represent and enforce locks. Mutex itself is short for “mutual exclusion”; once the lock is acquired, any other process, thread, and so on is prevented from acquiring the same lock.
See Also Pthreads, lock, locking.

MVCC

Acronym for multiversion concurrency control. This technique allows InnoDB transactions with certain **isolation levels** perform **consistent read** operations; that is, to query rows that are being updated by other transactions, and see the values from before those updates occurred. This is a powerful technique to increase **concurrency**, by allowing queries to proceed without waiting due to **locks** held by the other transactions.

This technique is not universal in the database world. Some other database products, and some other storage engines within MySQL, do not support it.
See Also ACID, isolation level, consistent read, concurrency, lock.

N

NULL

A special value in **SQL**, indicating the absence of data. Any arithmetic operation or equality test involving a `NULL` value, in turn produces a `NULL` result. (Thus it is similar to the IEEE floating-point concept of NaN, **not a number**.) Any aggregate calculation such as `AVG()` ignores rows with `NULL` values, when determining how many rows to divide by. The only test that works with `NULL` values uses the SQL idioms `IS NULL` or `IS NOT NULL`.

`NULL` values play a part in index operations, because for performance a database must minimize the overhead of keeping track of missing data values, Typically, `NULL` values are not represented in an index, because a query that tests an indexed column using a standard comparison operator could never match a row with a `NULL` value for that column. For the same reason, unique indexes do not prevent `NULL` values; those values simply are not represented in the index. Declaring a `NOT NULL` constraint on a column provides reassurance that there are no rows left out of the index, allowing for better query optimization (accurate counting of rows and estimation of whether to use the index).

Because the **primary key** must be able to uniquely identify every row in the table, a single-column primary key cannot contain any `NULL` values, and a multi-column primary key cannot contain any rows with `NULL` values in all columns.

Although the Oracle database allows a `NULL` value to be concatenated with a string, InnoDB treats the result of such an operation as `NULL`.

	See Also SQL, index, primary key.

P

page	A unit representing how much data InnoDB transfers at any one time between disk (the **data files**) and memory (the **buffer pool**). A page can contain one or more rows, depending on how much data is in each row. If a row does not fit entirely into a single page, InnoDB sets up additional pointer-style data structures so that the information about the row can be stored in one page. One way to fit more data in each page is to use **compressed row format**. See Also data files, buffer pool, page size, compressed row format.
page size	Currently, this value is fixed at 16 kilobytes. This is considered a reasonable compromise: large enough to hold the data for most rows, yet small enough to minimize the performance overhead of transferring unneeded data to memory. Other values are not tested or supported. See Also page.
partial index	An **index** that represents only part of a column value, typically the first N characters (the **prefix**) of a long `VARCHAR` value. See Also index, index prefix.
prefix	See index prefix.
primary key	A set of columns -- and by implication, the index based on this set of columns -- that can uniquely identify every row in a table. As such, it must be a unique index that does not contain any `NULL` values. InnoDB requires that every table has such an index (also called the **clustered index** or **cluster index**), and organizes the table storage based on the column values of the primary key. See Also clustered index, index.
pseudo-record	An artificial record in an index, used for **locking** key values or ranges that do not currently exist. See Also locking, supremum record, infimum record.
Pthreads	The POSIX threads standard, which defines an API for threading and locking operations on UNIX and Linux systems. On UNIX and Linux systems, InnoDB uses this implementation for mutexes. See Also mutex, lock.
purge	A type of garbage collection performed by a separate thread, running on a periodic schedule. The purge includes these actions: removing obsolete values from indexes; physically removing rows that were marked for deletion by previous `DELETE` statements. See Also delete, crash recovery, doublewrite buffer.

Q

query	An operation that reads information from one or more **tables** in a **database**. Depending on the organization of data and the parameters of the query, the lookup might be optimized by consulting an **index**. Normally with MySQL, queries are coded using SQL. (Even if you are using some other language for the main application logic.) With the **Embedded InnoDB** product, you skip SQL and code your own query logic, through calls to a C API.

See Also table, database, index, SQL, Embedded InnoDB.

R

read committed

An **isolation level** that uses a locking strategy that relaxes some of the protection between transactions, in the interest of performance. Transactions cannot see uncommitted data from other transactions, but they can see data that is committed by another transaction after the current transaction started. Thus, a transaction never sees any bad data, but the data that it does see may depend to some extent on the timing of other transactions.

When a transaction with this isolation level performs `UPDATE ... WHERE` or `DELETE ... WHERE` operations, other transactions might have to wait. The transaction can perform `SELECT ... FOR UPDATE`, and `LOCK IN SHARE MODE` operations without making other transactions wait.

In the **Embedded InnoDB**, this isolation level is specified by the constant `IB_READ_COMMITTED`.
See Also repeatable read, serializable read, locking, isolation level, transaction, ACID, Embedded InnoDB.

read uncommitted

The **isolation level** that provides the least amount of protection between transactions. Queries employ a **locking** strategy that allows them to proceed in situations where they would normally wait for another transaction. However, this extra performance comes at the cost of less reliable results, including data that has been changed by other transactions and not committed yet (known as **dirty read**). Use this isolation level only with great caution, and be aware that the results might not be consistent or reproducible, depending on what other transactions are doing at the same time. Typically, transactions with this isolation level do only queries, not insert, update, or delete operations.
See Also isolation level, locking, transaction, ACID, dirty read.

read tuple

In the **Embedded InnoDB** product, a read tuple is a larger kind of **tuple** than a **search tuple**. It contains extra columns so that you can retrieve values from all columns, not just from the indexed ones. When inserting or updating rows, you still use a **read** tuple; for that reason, we also refer to a **read/write** tuple, even though physically these structures are the same.
See Also Embedded InnoDB, search tuple, tuple, read/write tuple.

read/write tuple

Synonym for **read tuple**. The same structure is used for retrieving all the columns of a row, as for performing insert or update operations on a row.
See Also Embedded InnoDB, tuple, read tuple, search tuple.

redo log

A set of files, typically named `ib_logfile0` and `ib_logfile1`, that record statements that attempt to change data in InnoDB tables. These statements are replayed automatically to correct data written by incomplete transactions, on startup following a crash.
See Also ib_logfile.

redundant row format

The oldest InnoDB row format. Prior to MySQL 5.0.3, it was the only row format available in InnoDB. In My SQL 5.0.3 and later, the default is compact row format. You can still specify redundant row format for compatibility with older InnoDB tables.
See Also row format, compact row format.

repeatable read

The default **isolation level** for InnoDB. It prevents any rows that are queried from being changed by other transactions, thus blocking **non-repeatable reads** but not **phantom** reads. It uses a moderately strict **locking** strategy so that all queries within a transaction see data from the same snapshot, that is, the data as it was at the time the transaction started.

	When a transaction with this isolation level performs `UPDATE ... WHERE`, `DELETE ... WHERE`, `SELECT ... FOR UPDATE`, and `LOCK IN SHARE MODE` operations, other transactions might have to wait. In the **Embedded InnoDB** product, this isolation level is specified by the constant `IB_REPEATABLE_READ`. See Also consistent read, locking, isolation level, serializable read, transaction, ACID.
replication	The practice of sending changes from a **master database**, to one or more **slave databases**, so that all databases have the same data. This technique has a wide range of uses, such as load-balancing for better scalability, disaster recovery, and testing software upgrades and configuration changes. The changes can be sent between the database by methods called **row-based replication** and **statement-based replication**. See Also row-based replication, statement-based replication.
rollback	A **SQL** statement that ends a **transaction**, undoing any changes made by the transaction. It is the opposite of **commit**, which makes permanent any changes made in the transaction. By default, MySQL uses the **auto-commit** setting, which automatically issues a commit following each SQL statement. You must change this setting before you can use the rollback technique. Because the auto-commit setting is controlled at the MySQL level, it does not apply to **Embedded InnoDB** programs. See Also commit, ACID, transaction, Embedded InnoDB.
rollback segment	The storage area containing the **undo log**, part of the **system tablespace**. See Also undo log, system tablespace.
row-based replication	This form of **replication** is safe to use for all settings of the **innodb_autoinc_lock_mode** option. See Also replication, statement-based replication, auto-increment locking, innodb_autoinc_lock_mode.
row format	The disk storage format for a row from an InnoDB table. As InnoDB gains new capabilities such as compression, new row formats are introduced to support the resulting improvements in storage efficiency and performance. Each table has its own row format, specified through the `ROW_FORMAT` option. To see the row format for each InnoDB table, issue the command `SHOW TABLE STATUS`. Because all the tables in the system tablespace share the same row format, to take advantage of other row formats typically requires setting the `innodb_file_per_table` option, so that each table is stored in a separate tablespace. See Also fixed row format, dynamic row format, compact row format, redundant row format, compressed row format.
row lock	A **lock** that prevents a row from being accessed in an incompatible way by another **transaction**. See Also lock, lock mode, transaction.

S

savepoint	Savepoints help to implement nested **transactions**. They can be used to provide scope to operations on tables that are part of a larger transaction. For example, scheduling a trip in a reservation system might involve booking several different flights; if a desired flight is unavailable, you might **roll back** the changes involved in booking that one leg, without rolling back the earlier flights that were successfully booked.

	See Also transaction, rollback.
search tuple	In the **Embedded InnoDB** product, a search tuple is a fast, compact structure that only contains the columns needed to perform a search. You use this kind of **tuple** to perform queries where the answer requires only the columns from a single index. See Also Embedded InnoDB, tuple, read tuple, read/write tuple.
secondary index	A type of InnoDB **index** that represents a subset of table columns. An InnoDB table can have zero, one, or many secondary indexes. (Contrast with the **clustered index**, which is required for each InnoDB table, and stores the data for all the table columns.) A secondary index can be used to satisfy queries that only require values from the indexed columns. For more complex queries, it can be used to identify the relevant rows in the table, which are then retrieved through lookups using the clustered index. Creating and dropping secondary indexes has traditionally involved significant overhead from copying all the data in the InnoDB table. The **fast index creation** feature of the InnoDB Plugin makes both `CREATE INDEX` and `DROP INDEX` statements much faster for InnoDB secondary indexes. See Also index, clustered index, fast index creation.
semi-consistent read	A type of read operation used for `UPDATE` statements, that is a combination of **read committed** and **consistent read**. When an `UPDATE` statement examines a row that is already locked, InnoDB returns the latest committed version to MySQL so that MySQL can determine whether the row matches the `WHERE` condition of the `UPDATE`. If the row matches (must be updated), MySQL reads the row again, and this time InnoDB either locks it or waits for a lock on it. This type of read operation can only happen when the transaction has the read committed **isolation level**, or when the `innodb_locks_unsafe_for_binlog` option is enabled. See Also read committed, consistent read, isolation level.
serializable read	The **isolation level** that uses the most conservative locking strategy, to prevent any other transactions from inserting or changing data that was read by this transaction, until it is finished. This way, the same query can be run over and over within a transaction, and be certain to retrieve the same set of results each time. Any attempt to change data that was committed by another transaction since the start of the current transaction, cause the current transaction to wait. This is the default isolation level specified by the SQL standard. In practice, this degree of strictness is rarely needed, so the default isolation level for InnoDB is the next most strict, **repeatable read**. In Embedded InnoDB, this isolation level is specified by the constant `IB_SERIALIZABLE`. See Also repeatable read, consistent read, locking, isolation level, transaction, ACID.
shared lock	A kind of **lock** that allows other **transactions** to read the locked object, and to also acquire other shared locks on it, but not to write to it. The opposite of **exclusive lock**. See Also lock, transaction, exclusive lock.
shutdown	The process of stopping the InnoDB storage engine. This process can do various cleanup operations, so it is **slow** to shut down but fast to start up later; or it can skip the cleanup operations, so it is **fast** to shut down but must do the cleanup the next time it starts. In normal MySQL usage, the shutdown mode is controlled by the `innodb_fast_shutdown` option. In the **Embedded InnoDB** product, you specify the shutdown mode as a parameter to the shutdown API. See Also slow shutdown, fast shutdown, Embedded InnoDB.

slow shutdown	A type of shutdown that does additional flushing operations before completing. Specified by the configuration parameter `innodb_fast_shutdown=0`. Although the shutdown itself can take longer, that time will be saved on the subsequent startup. See Also shutdown, fast shutdown.
statement-based replication	This form of **replication** requires some care with the setting for the **innodb_autoinc_lock_mode** option, to avoid potential timing problems with **auto-increment locking**. See Also replication, auto-increment locking, innodb_autoinc_lock_mode, row-based replication.
SQL	The Structured Query Language that is standard for performing database operations. Often divided into the categories **DDL**, **DML**, and **queries**. The Embedded InnoDB product provides a C/C++ API for performing database operations, allowing you to skip the SQL language processing. See Also DDL, DML, query, Embedded InnoDB.
supremum record	A **pseudo-record** in an index, representing the **gap** above the largest value in that index. If a transaction has a statement such as `SELECT ... FOR UPDATE ... WHERE col > 10;`, and the largest value in the column is 20, it is a lock on the supremum record that prevents other transactions from inserting even larger values such as 50, 100, and so on. See Also gap, infimum record, pseudo-record.
system tablespace	By default, this single data file stores all the table data for a database, as well as all the metadata for InnoDB-related objects (the **data dictionary**). Turning on the **innodb_file_per_table** option causes each newly created table to be stored in its own **tablespace**, reducing the size of, and dependencies on, the system tablespace. Keeping all table data in the system tablespace has implications for the InnoDB Hot Backup product (backing up one large file rather than several smaller files), and prevents you from using certain InnoDB features that require the newer **Barracuda** file format. on the See Also data dictionary, file-per-table, innodb_file_per_table, ibdata file, tablespace, file format, Barracuda.

T

table	Although a table is a distinct, addressable object in the context of SQL, for backup purposes we are often concerned with whether the table is part of the system tablespace, or was created under the file-per-table setting and so resides in its own tablespace. See Also clustered index, file-per-table, table, system tablespace.
table schema	In the **Embedded InnoDB** product, a representation of the columns and **indexes** for a **table**. See Also Embedded InnoDB, table, index.
tablespace	A data file that can hold data for one or more tables. The **system tablespace** contains the tables that make up the **data dictionary**, and by default holds all the other InnoDB tables. Turning on the `innodb_file_per_table` option allows newly created tables to each have their own tablespace, with a separate data file for each table. Tablespaces created by the built-in InnoDB storage engine are upward compatible with the InnoDB Plugin. Tablespaces created by the InnoDB Plugin are downward compatible with the built-in InnoDB storage engine, if they use the **Antelope** file format. Tablespaces created with the built-in InnoDB storage engine or the InnoDB Plugin can be interchanged with Embedded InnoDB applications, subject to restrictions. Likewise, tablespaces

created by Embedded InnoDB applications can be interchanged with the InnoDB storage engine running as part of MySQL, either the built-in InnoDB or the InnoDB Plugin. For a tablespace created by the InnoDB storage engine in MySQL to be used by an Embedded InnoDB application, the Embedded InnoDB library must be compiled with compression enabled, if any tables use the **Barracuda** file format and **compressed row format**.
See Also file-per-table, innodb_file_per_table, system tablespace, data dictionary, ibdata file, Antelope, Barracuda, compressed row format.

transaction

Transactions are atomic units of work that can be committed or rolled back. When a transaction makes multiple changes to the database, either all the changes succeed when the transaction is committed, or all the changes are undone when the transaction is rolled back.

Database transactions, as implemented by InnoDB, have properties that are collectively known by the acronym **ACID**, for atomicity, consistency, isolation, and durability.
See Also ACID, commit, rollback, lock, isolation level.

truncate

A **DDL** operation that removes the entire contents of a table. Although conceptually it has the same result as a `DELETE` statement with no `WHERE` clause, it operates differently behind the scenes: InnoDB creates a new empty table, drops the old table, then renames the new table to take the place of the old one. Because this is a DDL operation, it cannot be **rolled back**.

If the table being truncated contains foreign keys that reference another table, the truncation operation uses a slower method of operation, deleting one row at a time so that corresponding rows in the referenced table can be deleted as needed by any `ON DELETE CASCADE` clause.

In the API for the Embedded InnoDB product, truncating a table gives the table a new ID, so that you must open the table again by name before using it. Any cursors that were open on the old table are invalid and produce undefined results.
See Also DDL, rollback, Embedded InnoDB.

tuple

In the **Embedded InnoDB** product, a data structure holding one or more table columns. The tuple is an intermediate holding area where query results are stored before the columns are extracted, or where column values are stored before being used in an insert or update operation.
See Also cursor, Embedded InnoDB, read tuple, read/write tuple, search tuple.

U

undo log

A storage area that holds copies of data modified by active **transactions**. If another transaction needs to see the original data (as part of a **consistent read** operation), the unmodified data is retrieved from this storage area. This area is physically part of the **system tablespace**. It is split into separate portions, the **insert undo buffer** and the **update undo buffer**. Collectively, these parts are also known as the **rollback segment**, a familiar term for Oracle DBAs.
See Also transaction, consistent read, system tablespace, rollback segment.

W

Windows

The built-in InnoDB storage engine and the InnoDB Plugin are supported on all the same Microsoft Windows versions as MySQL.

The InnoDB Hot Backup product is available on Windows, although the `innobackup` Perl script is not part of the Windows edition of the product.

The Embedded InnoDB product is available on Windows. Building from source requires the Microsoft Visual Studio product. The table compression feature also requires the `zlib` library, which is not a standard part of Windows; you can turn this feature off when building from source, to eliminate this dependency.

Index