

SimpleDBM Database API

Dibyendu Majumdar
d dot majumdar at gmail dot com

18 October 2009

Version: 1.0.13

Copyright: Copyright by Dibyendu Majumdar, 2008-2009

Contents

1	Introduction	5
1.1	Intended Audience	5
1.2	Pre-requisite Reading	5
2	Getting Started	7
2.1	SimpleDBM binaries	7
2.2	Creating a SimpleDBM database	7
2.2.1	Server Options	8
2.3	Opening a database	9
2.4	Performance impact of server options	10
2.5	Problems starting a database	11
2.6	Managing log messages	11
3	Transactions	15
3.1	Creating new Transactions	15
3.2	Working with Transactions	15
3.2.1	Transaction API	15
3.2.2	Transaction Savepoints	16
4	Tables and Indexes	17
4.1	Limitations	17
4.2	Creating a Table and Indexes	17
4.3	Isolation Modes	18
4.3.1	Common Behaviour	19
4.3.2	Read Committed/Cursor Stability	19
4.3.3	Repeatable Read (RR)	19
4.3.4	Serializable	19
4.4	Inserting rows into a table	19
4.5	Accessing table data	20
4.6	Updating tuples	21
4.7	Deleting tuples	22
5	The Database API	23
5.1	DatabaseFactory	23
5.2	Database	23
5.3	TableDefinition	25
5.4	Table	26
5.5	TableScan	27

Chapter 1

Introduction

This document describes the SimpleDBM Database API.

1.1 Intended Audience

This document is targeted at users of [SimpleDBM](#).

1.2 Pre-requisite Reading

Before reading this document, the reader is advised to go through the [SimpleDBM Overview](#) document.

Chapter 2

Getting Started

A SimpleDBM server is a set of background threads and a library of API calls that clients can invoke. The background threads take care of various tasks, such as writing out buffer pages, writing out logs, archiving older log files, creating checkpoints, etc.

A SimpleDBM server operates on a set of data and index files, known as the SimpleDBM database.

Only one server instance is allowed to access a SimpleDBM database at any point in time. SimpleDBM uses a lock file to detect multiple concurrent access to a database, and will refuse to start if it detects that a server is already accessing a database.

Multiple simultaneous threads can access SimpleDBM. Multiple transactions can be executed in parallel. SimpleDBM is fully multi-threaded and supports concurrent reads and writes.

Internally, SimpleDBM operates on logical entities called Storage Containers. From an implementation point of view, Storage Containers are mapped to files.

Tables and Indexes are stored in Containers known as TupleContainers and IndexContainers, respectively. Each container is identified by a numeric ID, called the Container ID. Internally, SimpleDBM reserves the container ID zero (0), so the first available ID is one (1).

The SimpleDBM database initially consists of a set of transaction log files, a lock file and a special container (ID 0) used internally by SimpleDBM.

2.1 SimpleDBM binaries

SimpleDBM makes use of Java 5.0 features, hence you will need to use JDK1.5 or above if you want to work with SimpleDBM.

You can download the SimpleDBM binaries from the SimpleDBM GoogleCode project download area. The following jar files are required:

- [simpledbm-common-1.0.x.jar](#) - this provides common utilities.
- [simpledbm-rss-1.0.x.jar](#) - this is the core database engine.
- [simpledbm-typesystem-1.0.x.jar](#) - provides a simple type system.
- [simpledbm-database-1.0.x.jar](#) - provides a higher level database API with support for tables and indexes.

You should make sure that required jars are in your class path.

2.2 Creating a SimpleDBM database

A SimpleDBM database is created by a call to `DatabaseFactory.create()`, as shown below:

```

import org.simpledbm.database.api.DatabaseFactory;
...
Properties properties = new Properties();
properties.setProperty("log.ctl.1", "ctl.a");
properties.setProperty("log.ctl.2", "ctl.b");
properties.setProperty("log.groups.1.path", ".");
properties.setProperty("log.archive.path", ".");
properties.setProperty("log.group.files", "3");
properties.setProperty("log.file.size", "5242880");
properties.setProperty("log.buffer.size", "5242880");
properties.setProperty("log.buffer.limit", "4");
properties.setProperty("log.flush.interval", "30");
properties.setProperty("log.disableFlushRequests", "true");
properties.setProperty("storage.createMode", "rw");
properties.setProperty("storage.openMode", "rw");
properties.setProperty("storage.flushMode", "noforce");
properties.setProperty("bufferpool.numbuffers", "1500");
properties.setProperty("bufferpool.writerSleepInterval", "60000");
properties.setProperty("transaction.ckpt.interval", "60000");
properties.setProperty("logging.properties.type", "log4j");
properties.setProperty("logging.properties.file",
    "classpath:simpledbm.logging.properties");
properties.setProperty("lock.deadlock.detection.interval", "3");
properties.setProperty("storage.basePath",
    "demodata/DemoDB");

DatabaseFactory.create(properties);

```

The `DatabaseFactory.create()` method accepts a `Properties` object as the sole argument. The `Properties` object can be used to pass a number of parameters. The available options are shown below. Note that some of the options have an impact on the performance and reliability of the server - especially those that control how SimpleDBM treats file IO.

2.2.1 Server Options

Property Name	Description
<code>log.ctl.{n}</code>	The fully qualified path to the log control file. The first file should be specified as <code>log.ctl.1</code> , second as <code>log.ctl.2</code> , and so on. Up to a maximum of 3 can be specified. Default is 2.
<code>log.groups.{n}.path</code>	The path where log files of a group should be stored. The first log group is specified as <code>log.groups.1.path</code> , the second as <code>log.groups.2.path</code> , and so on. Up to a maximum of 3 log groups can be specified. Default number of groups is 1. Path defaults to current directory.
<code>log.archive.path</code>	Defines the path for storing archive files. Defaults to current directory.
<code>log.group.files</code>	Specifies the number of log files within each group. Up to a maximum of 8 are allowed. Defaults to 2.
<code>log.file.size</code>	Specifies the size of each log file in bytes. Default is 2 KB.
<code>log.buffer.size</code>	Specifies the size of the log buffer in bytes. Default is 2 KB.

Property Name	Description
<code>log.buffer.limit</code>	Sets a limit on the maximum number of log buffers that can be allocated. Default is <code>10 * log.group.files</code> .
<code>log.flush.interval</code>	Sets the interval (in seconds) between log flushes. Default is 6 seconds.
<code>log.disableFlushRequests</code>	Boolean value, if set, disables log flushes requested explicitly by the Buffer Manager or Transaction Manager. Log flushes still occur during checkpoints and log switches. By reducing the log flushes, performance is improved, but transactions may not be durable. Only those transactions will survive a system crash that have all their log records on disk.
<code>storage.basePath</code>	Defines the base location of the SimpleDBM database. All files and directories are created relative to this location.
<code>storage.createMode</code>	Defines mode in which files will be created. Default is <code>"rws"</code> .
<code>storage.openMode</code>	Defines mode in which files will be opened. Default is <code>"rws"</code> .
<code>storage.flushMode</code>	Defines mode in which files will be flushed. Possible values are <code>noforce</code> , <code>force.true</code> (default), and <code>force.false</code>
<code>bufferpool.numbuffers</code>	Sets the number of buffers to be created in the Buffer Pool.
<code>bufferpool.writerSleepInterval</code>	Sets the interval in milliseconds between each run of the BufferWriter. Note that BufferWriter may run earlier than the specified interval if the pool runs out of buffers, and a new page has to be read in. In such cases, the Buffer Writer may be manually triggered to clean out buffers.
<code>lock.deadlock.detection.interval</code>	Sets the interval in seconds between deadlock scans.
<code>logging.properties.file</code>	Specifies the name of logging properties file. Precede <code>classpath:</code> if you want SimpleDBM to search for this file in the classpath.
<code>logging.properties.type</code>	Specify <code>"log4j"</code> if you want to SimpleDBM to use Log4J for generating log messages.
<code>transaction.lock.timeout</code>	Specifies the default lock timeout value in seconds. Default is 60 seconds.
<code>transaction.ckpt.interval</code>	Specifies the interval between checkpoints in milliseconds. Default is 15000 milliseconds (15 secs).

The `DatabaseFactory.create()` call will overwrite any existing database in the specified storage path, so it must be called only when you know for sure that you want to create a database.

2.3 Opening a database

Once a database has been created, it can be opened by creating an instance of `Database`, and starting it. The same properties that were supplied while creating the database, can be supplied when starting it.

Here is a code snippet that shows how this is done:

```
Properties properties = new Properties();
properties.setProperty("log.ctl.1", "ctl.a");
properties.setProperty("log.ctl.2", "ctl.b");
properties.setProperty("log.groups.1.path", ".");
properties.setProperty("log.archive.path", ".");
```

```

properties.setProperty("log.group.files", "3");
properties.setProperty("log.file.size", "5242880");
properties.setProperty("log.buffer.size", "5242880");
properties.setProperty("log.buffer.limit", "4");
properties.setProperty("log.flush.interval", "30");
properties.setProperty("log.disableFlushRequests", "true");
properties.setProperty("storage.createMode", "rw");
properties.setProperty("storage.openMode", "rw");
properties.setProperty("storage.flushMode", "noforce");
properties.setProperty("bufferpool.numbuffers", "1500");
properties.setProperty("bufferpool.writerSleepInterval", "60000");
properties.setProperty("transaction.ckpt.interval", "60000");
properties.setProperty("logging.properties.type", "log4j");
properties.setProperty("logging.properties.file",
    "classpath:simpledbm.logging.properties");
properties.setProperty("lock.deadlock.detection.interval", "3");
properties.setProperty("storage.basePath",
    "demodata/DemoDB");

Database db = DatabaseFactory.getDatabase(properties);
db.start();
try {
    // do some work
}
finally {
    db.shutdown();
}

```

Some points to bear in mind when starting SimpleDBM databases:

1. Make sure that you invoke `shutdown()` eventually to ensure proper shutdown of the database.
2. Database startup/shutdown is relatively expensive, so do it only once during the life-cycle of your application.
3. A Database object can be used only once - after calling `shutdown()`, it is an error to do any operation with the database object. Create a new database object if you want to start the database again.

2.4 Performance impact of server options

Some of the server options impact the performance or recoverability of SimpleDBM. These are discussed below.

log.disableFlushRequests Normally, the write ahead log is flushed to disk every time a transaction commits, or there is a log switch, or a checkpoint is taken. By setting this option to true, SimpleDBM can be configured to avoid flushing the log at transaction commits. The log will still be flushed for other events such as log switches or checkpoints. This option improves the performance of SimpleDBM, but will have a negative impact on recovery of transactions since the last checkpoint. Due to deferred log flush, some transaction log records may not be persisted to disk; this will result in such transactions being aborted during restart.

storage.openMode This is set to a `mode` supported by the standard Java `RandomAccessFile`. The recommended setting for recoverability is "rws", as this will ensure that modifications to the file

are persisted on physical storage as soon as possible. The setting “rw” may improve performance by allowing the underlying operating system to buffer file reads/writes. However, the downside of this mode is that if there is a crash, some of the file contents may not be correctly reflected on physical storage. This can result in corrupted files.

storage.flushMode This setting influences how/whether SimpleDBM invokes `force()` on underlying `FileChannel` when writing to files. A setting of “noforce” disables this; which is best for performance. A setting of “force.true” causes SimpleDBM to invoke `force(true)`, and a setting of “force.false” causes SimpleDBM to invoke `force(false)`. As for the other settings, this setting can favour either performance or recoverability.

While changing the default settings for above options can improve performance, SimpleDBM, like any database management system, requires high performance physical storage system to get the best balance between performance and recoverability.

A few other settings that affect the performance or scalability are discussed below.

bufferpool.numbuffers This setting affects the bufferpool size, and hence impacts the performance of the buffer cache. A bigger size is preferable; some experimentation may be required to determine the optimum size for a particular workload. Suggested default: 1000.

log.file.size SimpleDBM will not span log records across log files. Hence the maximum log file size affects the maximum size of an individual log record. See also the note on `log.buffer.size`. The maximum size of a log record limits the size of data operations (insert, update or delete). Suggested default: 5MB.

log.buffer.size For performance reasons, log records are buffered in memory. SimpleDBM will not span log records across log buffer boundaries, hence the maximum log buffer size restricts the size of a log record. Together, this option and the `log.file.size` setting dictate the maximum size of a log record. Suggested default: 5MB.

2.5 Problems starting a database

SimpleDBM uses a lock file to determine whether an instance is already running. At startup, it creates the file at the location `_internal\lock` relative to the path where the database is created. If this file already exists, then SimpleDBM will report a failure such as:

```
SIMPLEDBM-EV0005: Error starting SimpleDBM RSS Server, another
instance may be running - error was: SIMPLEDBM-ES0017: Unable to create
StorageContainer .._internal\lock because an object of the name already exists
```

This message indicates either that some other instance is running, or that an earlier instance of SimpleDBM terminated without properly shutting down. If the latter is the case, then the `_internal/lock` file may be deleted enabling SimpleDBM to start.

2.6 Managing log messages

SimpleDBM has support for JDK 1.4 style logging as well as Log4J logging. By default, if Log4J library is available on the classpath, SimpleDBM will use it. Otherwise, JDK 1.4 `util.logging` package is used.

You can specify the type of logging to be used using the Server Property `logging.properties.type`. If this is set to “log4j”, SimpleDBM will use Log4J logging. Any other value causes SimpleDBM to use default JDK logging.

The configuration of the logging can be specified using a properties file. The name and location of the properties file is specified using the Server property `logging.properties.file`. If the filename is

prefixed with the string “classpath:”, then SimpleDBM will search for the properties file in the classpath. Otherwise, the filename is searched for in the current filesystem.

A sample logging properties file is shown below. Note that this sample contains both JDK style and Log4J style configuration.:

```
#####
#       JDK 1.4 Logging
#####
handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler
.level= INFO

java.util.logging.FileHandler.pattern = simpledbm.log.%g
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.level = ALL

java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.ConsoleHandler.level = ALL

org.simpledbm.registry.level = INFO
org.simpledbm.bufmgr.level = INFO
org.simpledbm.indexmgr.level = INFO
org.simpledbm.storagemgr.level = INFO
org.simpledbm.walogmgr.level = INFO
org.simpledbm.lockmgr.level = INFO
org.simpledbm.freespacemgr.level = INFO
org.simpledbm.slotpagemgr.level = INFO
org.simpledbm.transactionmgr.level = INFO
org.simpledbm.tuplemgr.level = INFO
org.simpledbm.latchmgr.level = INFO
org.simpledbm.pagemgr.level = INFO
org.simpledbm.rss.util.level = INFO
org.simpledbm.util.level = INFO
org.simpledbm.server.level = INFO
org.simpledbm.trace.level = INFO
org.simpledbm.database.level = INFO

# Default Log4J configuration

# Console appender
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %p %c %m%n

# File Appender
log4j.appender.A2=org.apache.log4j.RollingFileAppender
log4j.appender.A2.MaxFileSize=10MB
log4j.appender.A2.MaxBackupIndex=1
log4j.appender.A2.File=simpledbm.log
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%d [%t] %p %c %m%n
```

```
# Root logger set to DEBUG using the A1 and A2 appenders defined above.
log4j.rootLogger=DEBUG, A1, A2

# Various loggers
log4j.logger.org.simpledbm.registry=INFO
log4j.logger.org.simpledbm.bufmgr=INFO
log4j.logger.org.simpledbm.indexmgr=INFO
log4j.logger.org.simpledbm.storagemgr=INFO
log4j.logger.org.simpledbm.walogmgr=INFO
log4j.logger.org.simpledbm.lockmgr=INFO
log4j.logger.org.simpledbm.freespacemgr=INFO
log4j.logger.org.simpledbm.slotpagemgr=INFO
log4j.logger.org.simpledbm.transactionmgr=INFO
log4j.logger.org.simpledbm.tuplemgr=INFO
log4j.logger.org.simpledbm.latchmgr=INFO
log4j.logger.org.simpledbm.pagemgr=INFO
log4j.logger.org.simpledbm.rss.util=INFO
log4j.logger.org.simpledbm.util=INFO
log4j.logger.org.simpledbm.server=INFO
log4j.logger.org.simpledbm.trace=INFO
log4j.logger.org.simpledbm.database=INFO
```

By default, SimpleDBM looks for a logging properties file named “simpledbm.logging.properties”.

Chapter 3

Transactions

Most SimpleDBM operations take place in the context of a Transaction. Following are the main API calls for managing transactions.

3.1 Creating new Transactions

To start a new Transaction, invoke the `Database.startTransaction()` method as shown below. You must supply an `IsolationMode`, try `READ_COMMITTED` to start with.:

```
import org.simplifiedb.database.api.Database;
import org.simplifiedb.rss.api.tx.IsolationMode;
import org.simplifiedb.rss.api.tx.Transaction;

Database database = ...;

// Start a new Transaction
Transaction trx = database.startTransaction(IsolationMode.READ_COMMITTED);
```

Isolation Modes are discussed in more detail in [Isolation Modes](#).

3.2 Working with Transactions

3.2.1 Transaction API

The Transaction interface provides the following methods for clients to invoke:

```
public interface Transaction {

    /**
     * Creates a transaction savepoint.
     */
    public Savepoint createSavepoint(boolean saveCursors);

    /**
     * Commits the transaction. All locks held by the
     * transaction are released.
     */
    public void commit();
```

```

/**
 * Rolls back a transaction upto a savepoint. Locks acquired
 * since the Savepoint are released. PostCommitActions queued
 * after the Savepoint was created are discarded.
 */
public void rollback(Savepoint sp);

/**
 * Aborts the transaction, undoing all changes and releasing
 * locks.
 */
public void abort();
}

```

A transaction must always be either committed or aborted. Failure to do so will lead to resource leaks, such as locks, which will not be released. The correct way to work with transactions is shown below:

```

// Start a new Transaction
Transaction trx = database.startTransaction(IsolationMode.READ_COMMITTED);
boolean success = false;
try {
    // do some work and if this is completed successfully ...
    // set success to true.
    doSomething();
    success = true;
}
finally {
    if (success) {
        trx.commit();
    }
    else {
        trx.abort();
    }
}
}

```

3.2.2 Transaction Savepoints

You can create transaction savepoints at any point in time. When you create a savepoint, you need to decide whether the scans associated with the transaction should save their state so that in the event of a rollback, they can be restored to the state they were in at the time of the savepoint. This is important if you intend to use the scans after you have performed a rollback to savepoint.

Bear in mind that in certain IsolationModes, locks are released as the scan cursor moves. When using such an IsolationMode, rollback to a Savepoint can fail if after the rollback, the scan cursor cannot be positioned on a suitable location, for example, if a deadlock occurs when it attempts to reacquire lock on the previous location. Also, in case the location itself is no longer valid, perhaps due to a delete operation by some other transaction, then the scan may position itself on the next available location.

If you are preserving cursor state during savepoints, be prepared that in certain IsolationModes, a rollback may fail due to locking, or the scan may not be able to reposition itself on exactly the same location.

Note that the cursor restore functionality has not been tested thoroughly in the current release of SimpleDBM.

Chapter 4

Tables and Indexes

SimpleDBM provides support for tables with variable length rows. Tables can have associated BTree indexes. In this section we shall see how to create new tables and indexes and how to use them.

4.1 Limitations

SimpleDBM supports creating tables and indexes but there are some limitations at present that you need to be aware of.

- All indexes required for the table must be defined at the time of table creation. At present you cannot add an index at a later stage.
- Table structures are limited in the type of columns you can have. At present Varchar, Varbinary, DateTime, Number, Integer and Long types are supported. More data types will be available in a future release of SimpleDBM.
- Null columns cannot be indexed.
- There is no support for referential integrity constraints or any other type of constraint. Therefore you need to enforce any such requirement in your application logic.
- Generally speaking, table rows can be large, but be aware that large rows are split across multiple database pages. The SimpleDBM page size is 8K.
- An Index key must be limited in size to about 1K in storage space.

4.2 Creating a Table and Indexes

You start by creating the table's row definition, which consists of an array of `TypeDescriptor` objects. Each element of the array represents a column definition for the table.

You use the `TypeFactory` interface for creating the `TypeDescriptor` objects as shown below.:

```
Database db = ...;
TypeFactory ff = db.getTypeFactory();
TypeDescriptor employee_rowtype[] = {
    ff.getIntegerType(), /* primary key */
    ff.getVarcharType(20), /* name */
    ff.getVarcharType(20), /* surname */
    ff.getVarcharType(20), /* city */
    ff.getVarcharType(45), /* email address */
}
```

```

    ff.getDateTimeType(), /* date of birth */
    ff.getNumberType(2) /* salary */
};

```

The next step is to create a `TableDefinition` object by calling the `Database.newTableDefinition()` method.:

```

TableDefinition tableDefinition = db.newTableDefinition("employee.dat", 1,
    employee_rowtype);

```

The `newTableDefinition()` method takes 3 arguments:

1. The name of the table container.
2. The ID for the table container. IDs start at 1, and must be unique.
3. The `TypeDescriptor` array that you created before.

Now you can add indexes by invoking the `addIndex()` method provided by the `TableDefinition` interface.:

```

tableDefinition.addIndex(2, "employee1.idx", new int[] { 0 }, true, true);
tableDefinition.addIndex(3, "employee2.idx", new int[] { 2, 1 }, false,
    false);
tableDefinition.addIndex(4, "employee3.idx", new int[] { 5 }, false, false);
tableDefinition.addIndex(5, "employee4.idx", new int[] { 6 }, false, false);

```

Above example shows four indexes being created.

The `addIndex()` method takes following arguments.

1. The ID of the index container. Must be unique, and different from the table container ID.
2. The name of the index container.
3. An array of integers. Each element of the array must refer to a table column by position. The table column positions start at zero. Therefore the array `{ 2, 1 }` refers to 3rd column, and 2nd column of the table.
4. The next argument is a boolean value to indicate whether the index is the primary index. The first index must always be the primary index.
5. The next argument is also a boolean value to indicate whether duplicate values are allowed in the index. If set, this makes the index unique, which prevents duplicates. The primary index must always be unique.

Now that you have a fully initialized `TableDefinition` object, you can proceed to create the table and indexes by invoking the `createTable()` method provided by the `Database` interface.:

```

db.createTable(tableDefinition);

```

Tables are created in their own transactions, and you have no access to such transactions.

It is important to bear in mind that all container names must be unique. Think of the container name as the file name. Also, the container IDs are used by SimpleDBM to identify each container uniquely. As explained before, SimpleDBM internally uses a special container with ID=0. Any tables and indexes you create must have container IDs ≥ 1 , and you must ensure that these are unique.

4.3 Isolation Modes

Before describing how to access table data using scans, it is necessary to describe the various lock isolation modes supported by SimpleDBM.

4.3.1 Common Behaviour

Following behaviour is common across all lock isolation modes.

1. All locking is on Row Locations (rowids) only. The SimpleDBM Rowid is called a TupleId.
2. When a row is inserted or deleted, its rowid is first locked in EXCLUSIVE mode, the row is inserted or deleted from data page, and only after that, indexes are modified.
3. Updates to indexed columns are treated as key deletes followed by key inserts. The updated row is locked in EXCLUSIVE mode before indexes are modified.
4. When fetching, the index is looked up first, which causes a SHARED or UPDATE mode lock to be placed on the row, before the data pages are accessed.

4.3.2 Read Committed/Cursor Stability

During scans, the rowid is locked in SHARED or UPDATE mode while the cursor is positioned on the key. The lock on current rowid is released before the cursor moves to the next key.

For most use cases, this is the recommended isolation mode as it provides the best concurrency.

4.3.3 Repeatable Read (RR)

SHARED mode locks obtained on rowids during scans are retained until the transaction completes. UPDATE mode locks are downgraded to SHARED mode when the cursor moves.

4.3.4 Serializable

Same as Repeatable Read, with additional locking (next key) during scans to prevent phantom reads.

4.4 Inserting rows into a table

To insert a row into a table, following steps are needed.

Obtain a transaction context in which to perform the insert.:

```
Transaction trx = db.startTransaction(IsolationMode.READ_COMMITTED);
boolean okay = false;
try {
```

Get the Table object associated with the table. Tables are identified by their container Ids.:

```
int containerId = 1;
Table table = db.getTable(trx, containerId);
```

Create a blank row. It is best to create new row objects rather than reusing existing objects.:

```
Row tableRow = table.getRow();
```

You can assign values to the columns as shown below.:

```
tableRow.setInt(0, i);
tableRow.setString(1, "Joe");
tableRow.setString(2, "Blogg");
tableRow.setDate(5, getDOB(1930, 12, 31));
tableRow.setString(6, "500.00");
```

Any columns you do not assign a value will be set to null automatically. The final step is to insert the row and commit the transaction.:

```

    table.addRow(trx, tableRow);
    okay = true;
} finally {
    if (okay) {
        trx.commit();
    } else {
        trx.abort();
    }
}
}

```

4.5 Accessing table data

In order to read table data, you must open a scan. A scan is a mechanism for accessing table rows one by one. Scans are ordered using indexes.

Opening an `TableScan` requires you to specify a starting row. If you want to start from the beginning, then you may specify `null` as the starting row. The values from the starting row are used to perform an index search, and the scan begins from the first row greater or equal to the values in the starting row.

In SimpleDBM, scans do not have a stop value. Instead, a scan starts fetching data from the first row that is greater or equal to the supplied starting row. You must determine whether the fetched key satisfies the search criteria or not. If the fetched key no longer meets the search criteria, you should call `fetchCompleted()` with a `false` value, indicating that there is no need to fetch any more keys. This then causes the scan to reach logical EOF.

The code snippet below shows a table scan that is used to count the number of rows in the table.:

```

Transaction trx = db.startTransaction(IsolationMode.READ_COMMITTED);
boolean okay = false;
int count = 0;
try {
    Table table = db.getTable(trx, 1);
    /* open a scan with null starting row */
    /* scan will use index 0 - ie - first index */
    TableScan scan = table.openScan(trx, 0, null, false);
    try {
        while (scan.fetchNext()) {
            scan.fetchCompleted(true);
            count++;
        }
    } finally {
        scan.close();
    }
    okay = true;
} finally {
    if (okay) {
        trx.commit();
    } else {
        trx.abort();
    }
}
}

```

The following points are worth noting.

1. The `openScan()` method takes an index identifier as the second argument. The scan is ordered by the index. Indexes are identified by the order in which they were associated with

the table, therefore, the first index is 0, the second is 1, and so on. Note that the index number is not the container ID for the index.

2. The third argument is the starting row for the scan. If `null` is specified, as in the example above, then the scan will start from logical negative infinity, ie, from the first row (as per selected index) in the table.
3. The scan must be closed in a finally block to ensure proper cleanup of resources.

4.6 Updating tuples

In order to update a row, you must first set the `RowId` using a scan. Typically, if you intend to update the tuple, you should open the scan in `UPDATE` mode. This is done by supplying a boolean `true` as the fourth argument to `openScan()` method.

Here is an example of an update. The table is scanned from first row to last and three of the columns are updated in all the rows.:

```
Transaction trx = db.startTransaction(IsolationMode.READ_COMMITTED);
boolean okay = false;
try {
    Table table = db.getTable(trx, 1);
    /* start an update mode scan */
    TableScan scan = table.openScan(trx, 0, null, true);
    try {
        while (scan.fetchNext()) {
            Row tr = scan.getCurrentRow();
            tr.setString(3, "London");
            tr.setString(4, tr.getString(1) + "." + tr.getString(2) + "@gmail.com");
            tr.setInt(6, 50000);
            scan.updateCurrentRow(tr);
            scan.fetchCompleted(true);
        }
    } finally {
        scan.close();
    }
    okay = true;
} finally {
    if (okay) {
        trx.commit();
    } else {
        trx.abort();
    }
}
```

The following points are worth noting:

1. If you update the columns that form part of the index that is performing the scan, then the results may be unexpected. As the data is updated it may alter the scan ordering.
2. The update mode scan places `UPDATE` locks on rows as these are accessed. When the row is updated, the lock is promoted to `EXCLUSIVE` mode. If you skip the row without updating it, the lock is either released (`READ_COMMITTED`) or downgraded (in other lock modes) to `SHARED` lock.

4.7 Deleting tuples

Start a table scan in UPDATE mode, if you intend to delete rows during the scan. Row deletes are performed in a similar way as row updates, except that `TableScan.deleteRow()` is invoked on the current row.

Chapter 5

The Database API

5.1 DatabaseFactory

```
/**
 * The DatabaseFactory class is responsible for creating and obtaining
 * instances of Databases.
 */
public class DatabaseFactory {

    /**
     * Creates a new SimpleDBM database based upon supplied properties.
     * For details of available properties, please refer to the SimpleDBM
     * User Manual.
     */
    public static void create(Properties properties);

    /**
     * Obtains a database instance for an existing database.
     */
    public static Database getDatabase(Properties properties);

}
```

5.2 Database

```
/**
 * A SimpleDBM Database is a collection of Tables. The Database runs as
 * an embedded server, and provides an API for creating and
 * maintaining tables.
 * A Database is created using DatabaseFactory.create(). An
 * existing Database can be instantiated using
 * DatabaseFactory.getDatabase().
 */
public interface Database {

    /**
     * Constructs a new TableDefinition object. A TableDefinition object
     * is used when creating new tables.
     */
}
```

```

*
* @param name Name of the table
* @param containerId ID of the container that will hold the table data
* @param rowType A row type definition.
* @return A TableDefinition object.
*/
public abstract TableDefinition newTableDefinition(String name,
        int containerId, TypeDescriptor[] rowType);

/**
 * Gets the table definition associated with the specified container ID.
 *
 * @param containerId Id of the container
 * @return TableDefinition
 */
public abstract TableDefinition getTableDefinition(int containerId);

/**
 * Starts the database instance.
 */
public abstract void start();

/**
 * Shuts down the database instance.
 */
public abstract void shutdown();

/**
 * Gets the SimpleDBM RSS Server object that is managing this database.
 * @return SimpleDBM RSS Server object.
 */
public abstract Server getServer();

/**
 * Starts a new Transaction
 */
public abstract Transaction startTransaction(IsolationMode isolationMode);

/**
 * Returns the TypeFactory instance associated with this database.
 * The TypeFactory object can be used to create TypeDescriptors
 * for various types that can become columns in a row.
 */
public abstract TypeFactory getTypeFactory();

/**
 * Returns the RowFactory instance associated with this database.
 * The RowFactory is used to generate rows.
 */
public abstract RowFactory getRowFactory();

/**
 * Creates a Table and associated indexes using the information

```

```

    * in the supplied TableDefinition object. Note that the table
    * must have a primary index defined.
    * The table creation is performed in a standalone transaction.
    */
public abstract void createTable(TableDefinition tableDefinition);

/**
 * Drops a Table and all its associated indexes.
 *
 * @param tableDefinition
 *       The TableDefinition object that contains information about the
 *       table to be dropped.
 */
public abstract void dropTable(TableDefinition tableDefinition);

/**
 * Gets the table associated with the specified container ID.
 *
 * @param trx Transaction context
 * @param containerId Id of the container
 * @return Table
 */
public abstract Table getTable(Transaction trx, int containerId);
}

```

5.3 TableDefinition

```

/**
 * A TableDefinition holds information about a table, such as its name,
 * container ID, types and number of columns, etc..
 */
public interface TableDefinition extends Storable {

    /**
     * Adds an Index to the table definition. Only one primary index
     * is allowed.
     *
     * @param containerId Container ID for the new index.
     * @param name Name of the Index Container
     * @param columns Array of Column identifiers - columns to be indexed
     * @param primary A boolean flag indicating that this is
     *                the primary index or not
     * @param unique A boolean flag indicating whether the index
     *               should allow only unique values
     */
    public abstract void addIndex(int containerId, String name, int[] columns,
                                  boolean primary, boolean unique);

    /**
     * Gets the Container ID associated with the table.
     */
    public abstract int getContainerId();
}

```

```

/**
 * Returns the Table's container name.
 */
public abstract String getName();

/**
 * Constructs an empty row for the table.
 * @return Row
 */
public abstract Row getRow();

/**
 * Returns the number of indexes associated with the table.
 */
public abstract int getNumberOfIndexes();

/**
 * Constructs an row for the specified Index. Appropriate columns
 * from the table are copied into the Index row.
 *
 * @param index The Index for which the row is to be constructed
 * @param tableRow The table row
 * @return An initialized Index Row
 */
public abstract Row getIndexRow(int indexNo, Row tableRow);
}

```

5.4 Table

```

/**
 * A Table is a collection of rows. Each row is made up of
 * columns (fields). A table must have a primary key defined
 * which uniquely identifies each row in the
 * table.
 * <p>
 * A Table is created by Database.createTable().
 * Once created, the Table object can be accessed by calling
 * Database.getTable() method.
 */
public interface Table {

    /**
     * Adds a row to the table. The primary key of the row must
     * be unique and different from all other rows in the table.
     *
     * @param trx The Transaction managing this row insert
     * @param tableRow The row to be inserted
     * @return Location of the new row
     */
    public abstract Location addRow(Transaction trx, Row tableRow);
}

```

```

/**
 * Updates the supplied row in the table. Note that the row to be
 * updated is identified by its primary key.
 *
 * @param trx The Transaction managing this update
 * @param tableRow The row to be updated.
 */
public abstract void updateRow(Transaction trx, Row tableRow);

/**
 * Deletes the supplied row from the table. Note that the row to be
 * deleted is identified by its primary key.
 *
 * @param trx The Transaction managing this delete
 * @param tableRow The row to be deleted.
 */
public abstract void deleteRow(Transaction trx, Row tableRow);

/**
 * Opens a Table Scan, which allows rows to be fetched from the Table,
 * and updated.
 *
 * @param trx Transaction managing the scan
 * @param indexno The index to be used for the scan
 * @param startRow The starting row of the scan
 * @param forUpdate A boolean value indicating whether the scan will
 *                  be used to update rows
 * @return A TableScan
 */
public abstract TableScan openScan(Transaction trx, int indexno,
                                   Row startRow, boolean forUpdate);

/**
 * Constructs an empty row for the table.
 * @return Row
 */
public abstract Row getRow();

/**
 * Constructs an row for the specified Index. Appropriate columns from the
 * table are copied into the Index row.
 *
 * @param index The Index for which the row is to be constructed
 * @param tableRow The table row
 * @return An initialized Index Row
 */
public abstract Row getIndexRow(int index, Row tableRow);
}

```

5.5 TableScan

```
/**
```

```
* A TableScan is an Iterator that allows clients to iterate through the
* contents of a Table. The iteration is always ordered through an Index.
* The Transaction managing the iteration defines the Lock Isolation level.
*/
public interface TableScan {

    /**
     * Fetches the next row from the Table. The row to be fetched depends
     * upon the current position of the scan, and the Index ordering of
     * the scan.
     * @return A boolean value indicating success of EOF
     */
    public abstract boolean fetchNext();

    /**
     * Returns a copy of the current Row.
     */
    public abstract Row getCurrentRow();

    /**
     * Returns a copy of the current Index Row.
     */
    public abstract Row getCurrentIndexRow();

    /**
     * Notifies the scan that the fetch has been completed
     * and locks may be released (depending upon the
     * Isolation level).
     * @param matched A boolean value that should be true
     *   if the row is part of the search criteria match result.
     *   If set to false, this indicates that no further
     *   fetches are required.
     */
    public abstract void fetchCompleted(boolean matched);

    /**
     * Closes the scan, releasing locks and other resources
     * acquired by the scan.
     */
    public abstract void close();

    /**
     * Updates the current row.
     */
    public abstract void updateCurrentRow(Row tableRow);

    /**
     * Deletes the current row.
     */
    public abstract void deleteRow();
}
```