CSQL User Manual

Prabakaran Thirumalai

Contents

1 Introduction

CSQL is a fast, multi-threaded SQL main memory database engine. It is a free software, licensed with the GNU GENERAL PUBLIC LICENSE http://www.gnu.org/ It aids in the development of high performance, fault-resilent applications requiring concurrent access to the shared data.

This database is built keeping ultra fast performance in mind. This database suits well as a front end database for other commercial database to increase the throughput. Throughput of queries involving single table may improve by 20x. This loads all the data into memory and avoids on demand fetches from disk and avoids the overhead of buffer manager found in commercial disk based database systems. It provides fast access through shared memory.

1.1 Platforms Supported

Linux
Solaris

1.2 Compilers Supported

g++ in Linux
CC in Solaris

1.3 Key Features

1. A very fast allocation system with thread cache.
2. It supports Atomicity, Consistency and Isolation. Does not support durability, which means that data is not persistent.
3.Algorithms are optimized as data is available always in memory rather than disk.
4.Tiny SQL Engine which supports single table queries and statements. Group by, Aggregates are not supported.
5.Multi Layered: ODBC, JDBC, SQL API, DB API
6.Isolation Level support:
READ UNCOMMITED, READ COMMITTED, READ REPEATABLE are supported. SERIALIZABLE isolation level is not supported.
7.Hash and TTree Index for faster lookups and scan.
8.Uses native test and set atomic instruction to implement latches.
9.Protection from process failures(resources held are released for dead processes).
10.Fine grained tuple level locking.

1.4 Architecture

Database file in CSQL is nothing but a unit of physical memory space mapped into the address space of the user process. It currently supports only one database file per instance. Apart from this user database file, a special database file named system database file, is mapped on to every user process accessing the database. User data is stored in the user database file and meta data information and control data such as catalog tables, lock information, log information are stored in the system database.
DB API provides interface for opening the database file and to create and manipulate tables. Every process or thread which needs to manipulate the database should first register itself with the database server. This enables the server to book keep all the resources held by the process or thread. If any process dies or crashes for some reason, its resources (lock held) are released by the database server automatically.
Concurrency Manager is implemented using locks and latches. and it supports first three levels of the isolation. Transaction manager supports interface for commit and rollback.

1.5 Compiling the source:

Go to the root directory and enter the following commands.

```
$./configure --prefix=‘pwd‘/install
$make
$make install
```

This will create ”install” directory under the current directory and places all the executables created in bin directory and libraries in lib directory.

1.6 Generating API Reference

Go to the root directory and enter

```
$doxygen
```

This will create ”docs/html” directory under which API Reference html files are stored. Refer index.html in that directory.

2 Getting Started

2.1 Starting the Server

You should find csqlserver executable under the bin directory of the installation.

```
$./csqlserver
```

This starts the server and creates the database file.

2.2 Shutting down the Server

Pressing Ctrl-C on the terminal where csqlserver is running, is safe and will stop the server gracefully by removing the database file.

2.3 DB API

This allows user process and threads to access or manipulate the database. Main interfaces are Connection, DatabaseManager, Table, etc. Connection provides interface to connect and disconnect to the database file DatabaseManager provides interface to create and drop database objects including tables and indexes. Table provides interface to insert, update, delete and fetch tuples.

Refer API reference under the directory docs/html. (If there is no html, you shall generate it by yourself using doxygen tools. Refer the previous section for this.)

2.3.1 Getting a Connection

Connection interface is the heart of all the interfaces as it is the entry point for database access and it provides interface for transaction commit/rollback.

You can obtain a connection to database using the following code snippet:

```
Connection conn;
DbRetVal rv = OK;
rv = conn.open("praba", ",manager");
if (rv!= OK) return -1;
...
```

2.3.2 Creating Tables

Database Manager provides interface for table creation and deletion. We shall obtain the DatabaseManager object from the connection object. The table or schema definition is encapsulated in TableDef interface. It provides methods to specify the field definition of the table. For example to create table with two fields,

```
DatabaseManager *dbMgr = conn.getDatabaseManager();
if (dbMgr == NULL) { printf("Bad connection \n"); return -1;}
TableDef tabDef;
tabDef.addField("f1", typeInt, 0, NULL, true, true);
tabDef.addField("f2", typeString, 196);
rv = dbMgr->createTable("t1", tabDef);
if (rv != OK) { printf("Table creation failed\n"); return -1; }
```

First argument of addField method is field name, second is the type of the field, third argument is length, fourth is default value, fifth is not null flag and last is primary key flag. In our example, field ”f1” is not null and it is the primary key for the table. Call addField for all the field definition in the table and call createTable passing table name as first argument and TableDef as second argument.

2.3.3 Inserting into the tables

Any DML operation requires a transaction to be started. All the operations happen within the context of this transaction. Application buffer should first be binded to the respective fields using the bindFld method as mentioned in the below example; insertTuple method will pick values from the binded application buffer and creates a new row or tuple in the table.

```
Table *table = dbMgr->openTable("t1");
if (table == NULL) { printf("Unable to open table\n"); return -1; }
int id = 100;
char name[196] = "PRABAKARAN";
table->bindFld("f1", &id);
table->bindFld("f2", name);
conn.startTransaction();
ret = table->insertTuple();
if (ret != OK) { printf("Unable to insert tuple\n"); return -1; }
conn.commit();
```

2.3.4 Selecting tuples from the tables

Application buffer should first be binded to the respective fields using the bindFld method as we did for insert; fetch method will copy the values from the row to the binded application buffer. Predicate if required shall be created using the condition interface and set in the table object before the execution. Calling execute method is must before calling the fetch method, as it will create the execution plan for the scan. It selects appropriate index based on the predicate set.

```
Condition p1;
int val1 = 100;
p1.setTerm("f1", OpEquals, &val1);
table->setCondition(p1);
int id = 100;
char name[196] = "PRABAKARAN";
table->bindFld("f1", &id);
table->bindFld("f2", name);
table->execute();
void *tuple = (char*)table->fetch() ;
if (tuple == NULL) {printf(" No tuple found \n" ); table->close();return -1;}
printf("tuple value is %d %s \n",id ,name);
table->close();
```

2.3.5 Updating or Deleting tuples

This operation is allowed always on existing scan. When this method is called the current tuple in the scan in either updated with application buffer values which are binded, or gets deleted based on the method called. For example:

```
tuple = (char*)table->fetch() ;
if (tuple == NULL)  {printf(" No tuple found \n" ); table->close();return -1;}
strcpy(name, "PRABAKARAN0950576543210"); //update the value
table->updateTuple();
```