



CSQL Main Memory Database Cache

User Manual

www.csqldb.com

Contents

Page No.

1. Introduction	7
1.1 What is CSQL.....	7
1.2 CSQL is Unique.....	7
1.3 CSQL as Open Source software	8
1.4 Who can use CSQL	8
1.5 CSQL Features	9
2. Getting Started	9
2.1 Where to find CSQL	10
2.2 OS / Platform and Compiler support.....	10
2.3 How to build / compile CSQL	10
2.3.1 Building CSQL from the CVS repository.....	11
2.4 Directory Layout and Files.....	12
2.4.1 src	12
2.4.2 include.....	13
2.4.3 docs	13
2.4.4 examples	13
2.4.5 test.....	13
2.5 How to Configure CSQL	14
2.6 Starting and Stopping csqserver	14
2.7 Running examples	15
2.8 How to run test scripts from test directory	15
3. How to Use CSQL	16
3.1 Environmental variables	16
3.1.1 CSQL_CONFIG_FILE.....	16
3.1.2 LD_LIBRARY_PATH	16
3.1.3 PATH	17
3.1.4 CLASSPATH.....	17
3.2 CSQL Tool	17
3.3 SQL Data Definition Language (DDL)	17
3.3.1 CREATE TABLE	17
3.3.2 DROP TABLE	19
3.4 SQL Data Manipulation Language (DML)	19
3.4.1 INSERT.....	19
3.4.2 UPDATE.....	20
3.4.3 DELETE	21
3.5 SQL Data Query Language (DQL)	22

3.5.1	SELECT	22
4.	JDBC Driver	24
4.1	What is JDBC	24
4.1.1	Implemented Interfaces.....	24
4.1.2	JDBC Datatype	25
4.2	Establishing a Connection	26
4.2.1	Loading the Driver:.....	26
4.2.2	Making the Connection.....	27
4.3	Creating Table.....	28
4.3.1	Creating Index.....	28
4.3.2	Closing and committing the Statement object	28
4.4	Inserting tuples	29
4.5	Updating Rows	31
4.6	Deleting Rows	31
4.7	Fetching the Rows	32
4.8	Drop the table	33
4.9	Close the connection.....	33
5.	ODBC Driver.....	34
5.1	Why ODBC	34
5.2	Components of ODBC	34
5.3	ODBC API Overview.....	35
5.3.1	ODBC Handles	35
5.4	ODBC API	37
5.4.1	SQLPrepare.....	37
5.4.2	SQLExecute	38
5.4.3	SQLBindParameter	38
5.4.4	SQLBindCol	39
5.5	Data Types	40
5.6	ODBC API with Examples	41
5.6.1	Connect to the CSQL.....	41
5.6.2	Transactions	44
5.6.3	Create Table	45
5.6.4	Insert records into the table.....	46
5.6.5	Fetch the records	49
5.6.6	Update Records.....	51
5.6.7	Delete Records	52
5.6.8	Drop the Table	52
5.6.9	Freeing Handles and Disconnect from the CSQL.....	53
6.	SQL API	53
6.1	Connect to the Database.....	53
6.2	Create and Set the statement for the connection	54
6.3	Create the table	54
6.3.1	Prepare Statement	54

6.3.2	Execute the Statement and release the memory.....	55
6.4	Insert tuples into the table.....	55
6.4.1	Prepare the statement.....	55
6.4.2	Start the transaction.....	55
6.4.3	Parameterize the fields.....	56
6.4.4	Execute the insert statement.....	56
6.4.5	Commit the transaction.....	57
6.5	Read the tuples (rows) from the table.....	57
6.5.1	Read tuples from the table.....	57
6.5.2	Prepare the statement.....	57
6.5.3	Bind the fields.....	57
6.5.4	Begin transaction and execute the statement.....	58
6.5.5	Fetch the tuples.....	58
6.6	Update some tuples.....	59
6.7	Delete tuples.....	59
7.	DB API.....	60
7.1	Connect to the database.....	60
7.2	Database Manager creates the table.....	61
7.2.1	Get the DatabaseManager.....	61
7.2.2	Define the table.....	61
7.2.3	Create table.....	61
7.2.4	Create index for primary key field.....	62
7.3	Insert tuples into the table.....	62
7.3.1	Open the table.....	62
7.3.2	Bind each field of the table.....	63
7.3.3	Start the transaction.....	63
7.3.4	Insert the tuples.....	63
7.3.5	Commit the transaction.....	63
7.4	Read the tuples from the database.....	64
7.4.1	Set and execute condition to read all the inserted tuples.....	64
7.4.2	Fetch the tuples.....	64
7.5	Update some of the tuples.....	64
7.5.1	Set a condition to update the tuples.....	64
7.5.2	Start transaction and be prepared to update.....	65
7.5.3	fetch and update the tuples.....	66
7.6	Delete some of the tuples.....	66
8.	Archive and Restore.....	66
	Archive the database.....	67
	Restore the database.....	67
9.	CSQL Cache.....	67
9.1	CSQL Cache Functionalities.....	68
9.1.1	Updateable Cache Tables.....	68
9.1.2	Bi-Directional Updates.....	68
9.1.3	Synchronous and Asynchronous update propagation.....	68
9.1.4	Multiple cache granularity.....	68

9.1.5	Recovery for cached tables.....	68
9.1.6	Tools to validate the coherence of cache	68
9.1.7	Horizontally Scalable	68
9.1.8	Transparent access to non-cached tables reside in target database.....	68
9.1.9	Transparent Fail over	68
9.2	CSQL Configuration.....	69
9.3	Unidirectional Cache Configuration Settings	70
9.3.1	MySQL Configuration	70
9.3.2	Postgres Configuration.....	72
9.4	Starting the csqserver	73
9.5	Working with CSQL gateway	74
9.6	Programming with CSQL gateway	75
9.7	Configuring Bi-Directional Cache.....	76
9.7.1	MySQL Configuration.....	76
9.7.2	Postgres Configuration.....	77
10	Configuration.....	79
10.1	Server section variables.....	79
10.1.1	PAGE_SIZE.....	79
10.1.2	MAX_PROCS.....	79
10.1.3	MAX_SYS_DB_SIZE	80
10.1.4	MAX_DB_SIZE	80
10.1.5	SYS_DB_KEY	80
10.1.6	USER_DB_KEY.....	80
10.1.7	LOG_FILE.....	80
10.1.8	DATABASE_FILE.....	80
10.1.9	MAP_ADDRESS.....	80
10.2	Client section variables	80
10.2.1	MUTEX_TIMEOUT_SECS.....	80
10.2.2	MUTEX_TIMEOUT_USECS	81
10.2.3	MUTEX_TIMEOUT_RETRIES	81
10.2.4	LOCK_TIMEOUT_SECS	81
10.2.5	LOCK_TIMEOUT_USECS	81
10.2.6	LOCK_TIMEOUT_RETRIES	81
10.3	Cache section variables	81
10.3.1	CACHE_TABLE	81
10.3.2	DSN.....	81
10.3.3	TABLE_CONFIG_FILE	81
10.3.4	ENABLE_BIDIRECTIONAL_CACHE	82
10.3.5	CACHE_RECEIVER_WAIT_SECS	82
11	Tool reference	82
11.1	csql.....	82
11.2	catalog	82
11.3	csqldump.....	84
11.4	cachetable	87
11.5	cacheverify.....	88

12.Troubleshooting	92
12.1 Errors while building CSQL	92
12.1.1 Please set JDK_HOME.....	92
12.1.2 Cannot find -lodbc.....	93
12.2 Errors while running csqserver.....	93
12.2.1 - bash: csqserver Command not found.....	93
12.2.2 Unable to create the log file.....	93
13.Getting Support	94
14.How to contribute.....	94
Appendix – A (Benchmark Results)	95
Machine Configuration	95
Schema Definition	95
CSQL MMDB Benchmark Results	95
CSQL Cache Benchmark Results	96

1. Introduction

The CSQL Main Memory Database Cache is an easily accessible and powerful database management system that can also serve as a source of information for performance related research. This constitutes of two major components

- CSQL Main memory Database
- CSQL Cache

1.1 What is CSQL

CSQL is a compact main memory database SQL engine that supports limited set of features and gives ultra fast response for database queries. Many applications like telecom, process control, airline reservation, stock market etc., require real time access to data.

Main memory databases, which have become more feasible recently with the increasing availability of large amounts of memory at very low costs, can provide better performance and consistent throughput than disk based database systems.

The basic philosophy behind CSQL's design is the fact that accessing data from main memory is an order of magnitude faster than accessing data from disk. And recent technological advancements have only shown that memory speeds and network speeds are advancing in leaps and bounds, whereas disk I/O speeds are not increasing in the same proportions. Hence the major platform dependency in CSQL is not on the disk, but on the memory and going ahead will be the network.

CSQLCache, a client side caching mechanism for any disk-based database (Oracle, Sybase, DB2, MySql, etc.) shall increase the throughput of existing applications by multi-folds without requiring any application changes. It will retrieve the frequently accessed tables from the target database and place it in main memory database (CSQL). Any further operations will be carried out from the main memory database rather than target database. There are many options supported for synchronizing the data between the cache and the target database.

CSQL and its associated suite of products have a single design objective and that is undisruptive performance, close to 20 times faster than traditional options and it has achieved these performances because it has been designed from scratch to achieve performance.

1.2 CSQL is Unique

CSQL is unique for the following reasons.

- It keeps the data in main memory rather than disk.

- It is 20 times* faster than any disk based database system.

* *Test results are shown in Appendix A*

- No buffer manager overhead is present since all the records are stored in main memory.
- The data structures and algorithms are targeted for memory access.
- Supports primitive SQL, ODBC and JDBC standard interface.
- Proprietary SQL API and DBAPI for faster access.
- Client side cache for any target database

The future will see an increase in demand for main memory databases as performance will be the crucial deciding factor and CSQL will ensure that performance expectations of all real time applications are met.

1.3 CSQL as Open Source software

Released on the 15th of May 2008, the CSQL MMDB is available as Open Source software at Sourceforge (www.sourceforge.net), world's largest development and download repository of open source code and applications.

CSQL is available at www.csqldb.com and also at <http://www.sourceforge.net/projects/csql>. This release also includes client side caching for any disk based database systems.

The current release of CSQL works on Linux Platform and it supports DDL operations like CREATE TABLE, DROP TABLE and DML operations like INSERT, SELECT, UPDATE and DELETE on single tables.

1.4 Who can use CSQL

CSQL has been designed from scratch with a single point benefit – to provide undisruptive performance benefits in application domains where real time access to data is a core necessity. For example –

- Financial and Insurance Industry
- Information Technology
- Telecommunication Industry

But again that does not mean CSQL is the preserve of white-coated DBAs sitting in chilled rooms!!!

CSQL through its simplicity of design and usage is equally appealing to individual researchers, students or some one simply interested in experimenting on another Open Source database.

As an Open Source initiative we would welcome brickbats and bouquets at feedback@csqldb.com

1.5 CSQL Features

CSQL shall work as stand-alone main memory database management system or as cache for any other database management system. Below is the list of features CSQL MMDB supports.

ACI Compliant

Hash Indexing for fast lookups

Tree Indexing for faster range lookups

Highly Concurrent - Row Level Locking

Multi Granular Locking - Table and Row Level Locking

Isolation Level Support – Read Uncommitted, Committed, Repeatable

Constraints - Primary Key, Unique, Not Null

Multi user

Data Type support - Primitive types, Date, Time and Timestamp, binary, char

Operator Support - >, <, <=, >=, =, !=, LIKE, IN, BETWEEN

Fault tolerance -Process cleanup for resources

Archive/Restore

Primitive ODBC Driver – DDL, DML operations with parameters

Primitive JDBC Driver - DDL, DML operations with parameters

SQL Support –

CREATE TABLE, DROP TABLE, CREATE INDEX, DROP INDEX

INSERT INTO <tablename> [fieldNameList] VALUES (valuelist)

UPDATE <tablename> SET fldname=value , ... [WHERE <condition>]

DELETE FROM <tablename> [WHERE <condition>]

SELECT <* | fieldNameList | aggList> FROM <tablename>

[WHERE <condition>] [GROUP BY <fieldname>]

condition shall be [NOT] <predicate> <AND | OR > <predicate>

predicate shall be <fieldName> <"=, !=, >,<, >=, <="> [fieldName | value] |

<fieldName> [NOT] <BETWEEN value AND value> |

<fieldName >[NOT] <IN valuelist> |

<fieldName> LIKE for character datatype

aggList shall contain MIN|MAX|SUM|AVG|COUNT(<fieldname>), ...

Note: For list of caching features refer section 8.1

2. Getting Started

2.1 Where to find CSQL

- Go to www.csqldb.com and follow the Download link. It is also available at <http://www.sourceforge.net/projects/csql>
- Download the source file `csql-src-2.1.tar.gz`

2.2 OS / Platform and Compiler support

- CSQL runs on Linux operating system on Intel x86 architecture.
- The g++ compiler must be present to build CSQL (it is a default install with any Linux distro).

2.3 How to build / compile CSQL

To build CSQL, i.e. to compile the CSQL source make sure you have the following tools installed on the Linux box:

- Make
- Automake
- Autoconf
- libtool
- unixODBC
- jdk 1.6

Most of these are installed by default during the Linux installation; just make sure that “Development packages” are included during the installation process. During the installation process, it will ask for the packages that need to be installed. It will appear in the screen with combo box unchecked under the item “Development packages”. This check box needs to be checked before you press the ‘Next’ button for these packages to be installed during the process.

Building CSQL from source file

- Copy the source file `csql-src-2.1.tar.gz` into your `<home-dir>`.
- Extract the files

```
$ tar zxvf csql-src-2.1.tar.gz
```

This will extract the files under `csql-src-2.1` directory in the current directory.

- Go to `csql-src-2.1` directory and run

```
$ export JDK_HOME=<path of your jdk installation>
```

To know the path to your JDK installation, please do the following –

```
$ locate javac
```

If the output is

```
/opt/java/jdk1.6.0_04/bin/javac
/opt/java/jdk1.6.0_04/man/ja_JP.eucJP/man1/javac.
1
/opt/java/jdk1.6.0_04/man/man1/javac.1
/opt/java/jdk1.6.0_04/sample/javac
/usr/share/vim/vim70/compiler/javac.vim
/usr/share/vim/vim70/syntax/javacc.vim
```

Then set the directory which has java1.6 compiler as below

```
$ export JDK_HOME=/opt/java/jdk1.6.0_04
$ export PATH=$JDK_HOME/bin:$PATH
$ ./build.ksh
```

If you see the below message “Build completed Successfully” then it means that, CSQL build is ready.

If you face any issues post it in the below forum

http://sourceforge.net/forum/forum.php?forum_id=562615

Before posting questions, check whether any thread discusses your issue.

Check Section 2.5 on how to configure CSQL.

2.3.1 Building CSQL from the CVS repository

This project's SourceForge.net CVS repository can be checked out through anonymous (pserver) CVS with the following instruction set.

```
$ cvs - \
>d:pserver:anonymous@csql.cvs.sourceforge.net:/cv
sroot/csql \
login
```

When prompted for a password for anonymous, simply press the Enter key.

```
$ cvs -z3 \
```

```
>  
d:pserver:anonymous@csql.cvs.sourceforge.net:/cvsroot/csql  
\  
> co -P csql
```

This will place all the files in the csql directory, which will be created under the current working directory.

```
$ cd csql  
$ export JDK_HOME=<path of your jdk installation>
```

Note: Refer section 2.3 for information on find the correct jdk installation path.

```
$ export PATH=$JDK_HOME/bin:$PATH  
$ ./build.ksh
```

Now the CSQL build is ready. Check Section 2.5 on how to configure CSQL.

2.4 Directory Layout and Files

CSQL directory layout is divided mainly into the following five directories.

- src
- include
- docs
- examples
- test

2.4.1 src

This directory contains the following subdirectories that contain c++ source code for all the modules like CSQLCache, SQL Engine, JDBC and ODBC drivers, SQL API, DB API etc.

- adapter
- cache
- gateway
- jdbc
- network
- server
- sql
- sqllog
- tools

2.4.2 include

This directory contains all the header files that include class declarations of various classes of CSQL.

2.4.3 docs

This directory contains the User Manual in pdf format.

For creating API documentation, the “doxygen” tool can be used. Refer README file for details. You can also find the API documentation for DBAPI and SQLAPI at <http://csql.sourceforge.net/apidocs/index.html>

2.4.4 examples

This directory is subdivided into the following subdirectories that explain how to interact with CSQL with different supported APIs.

- dbapi
- isql
- odbc
- jdbc
- sqlapi

Refer to Section 2.7 for how to run the examples present in these directories.

2.4.5 test

This directory includes most of the test scripts that have been written during the development phases of the CSQL database. These test scripts are the most comprehensive and exhaustive.

It includes the following subdirectories.

- dbapi
- jdbc
- odbc
- sqlapi
- performance
- system
- tools

Each of its subdirectory represents test module and each module contains test scripts, which tests each and every functionality of that module.

Please refer to Section 2.8 to know, how to run these test scripts.

2.5 How to Configure CSQL

The default database size for CSQL is 10 MB. If you wish to change the size to more than 30 MB, then the system variable `kernel.shmmax` should be set to either same size or more than the size of the database. Only the superuser has the privilege to run this command.

You can ask your system administrator to set it for you in case you are not the super user of the system.

The following command sets the kernel parameter to 1 GB,

```
# /sbin/sysctl -w kernel.shmmax=1000000000
kernel.shmmax=1000000000

$ cd <CSQL_ROOT>
$ . ./setupenv.ksh
```

This will set all the environmental variables defined in `setupenv.ksh` file present in `<CSQL_ROOT>` directory. Refer to Section 3.1 for environmental variables.

Note: Run `setupenv.ksh` in each and every terminal or console you open before you do any operation with CSQL.

2.6 Starting and Stopping csqserver

Once CSQL is built and configured the server is ready to start. The CSQL Server can be invoked by running the following command:

```
$ csqserver
ConfigValues
getPageSize 8192
getMaxProcs 100
....
....
....
getMaxLogStoreSize 1048576
getNetworkID 1
getCacheNetworkID -1
sysdb size 1048576 dbsize 10485760
System Database initialized
Database server started
```

If the ensuing screen output looks similar to the above, then the server is ready for operations. To stop the server just press <Ctrl + C> from the terminal where the server is running –

```
Received signal 2
Stopping the server
Server Exiting.
```

The above output message is displayed during the exit. This will stop the server gracefully by removing the database and doing the necessary clean ups.

2.7 Running examples

You are now ready to run some examples given in examples directory. This directory contains following subdirectories

dbapi – contains dbapiexample.c with Makefile and README

sqlapi - contains sqlapiexample.c with Makefile and README

isql - contains sql input files and README

jdbc - contains jdbcexample.java, gwexample.java with Makefile and README

odbc - contains odbcexample.c with Makefile and README

Each of these subdirectory contain README file that will guide you to compile and run these examples.

Note: Make sure the server is started in another terminal and setupenv.ksh is executed before running the examples.

2.8 How to run test scripts from test directory

Set the environment variables by running setupenv.ksh present in the CSQL root directory

```
$ cd <CSQL_ROOT>
$ . ./setupenv.ksh
```

Create a directory in your home directory

```
$ cd
$ mkdir testResults
```

Set the environmental variable TEST_RUN_ROOT to point to this directory.

```
$ export TEST_RUN_ROOT=<user-home>/testResults
```

Go to the test directory in CSQL root directory.

```
$ cd <CSQL_ROOT>/test
```

Run the following command and wait for it to finish.

```
$ make runall
```

It may take several minutes to finish this test. It will generate a test report once running through all the test scripts mentioning the number of scripts passed and failed in each module. The testResults directory will have the logs of all the test scripts under appropriate subdirectories.

If you face any issues post it in the below forum

http://sourceforge.net/forum/forum.php?forum_id=562615

Before posting questions, check whether any thread discusses your issue.

3. How to Use CSQL

3.1 Environmental variables

CSQL has a set of environmental variables that need to be set after the build is ready and before starting the server. To carry out the various database operations, either by using the `csql` tools or using executables which link with `csql` libraries, the following environmental variables need to be set.

Note: Let us assume `CSQL_ROOT` is the absolute path where CSQL is installed.

3.1.1 CSQL_CONFIG_FILE

There is a configuration file called `csql.conf` in the `CSQL_ROOT` directory, which the `csqlserver` reads during loading up. This file has the configuration variables for the CSQL system. These variables are explained in detail in Section 8.

```
$ export CSQL_CONFIG_FILE=<CSQL_ROOT>/csql.conf
```

3.1.2 LD_LIBRARY_PATH

This variable is set to locate the CSQL `lib` directory that contains CSQL specific libraries.

```
$ export \  
> LD_LIBRARY_PATH=<CSQL_ROOT>/install/lib:$LD_LIBRARY_PATH
```


3.1.3 PATH

This variable is set to locate the CSQL bin directory that contains the CSQL executables.

```
$ export PATH=<CSQL_ROOT>/install/bin:$PATH
```

3.1.4 CLASSPATH

This variable is set to locate the libraries for the JDBC driver of CSQL.

```
$ export /  
> CLASSPATH=<CSQL_ROOT>/install/lib/CSqlJdbcDriver.jar:.
```

Running the following script from the CSQL root directory will set the above variables automatically.

```
$ . ./setupenv.ksh
```

3.2 CSQL Tool

CSQL provides an interactive SQL client tool called `csql`, which communicates with the CSQL database. It supports most of the standard SQL statements like the DDL and DML. It executes as a sub-shell and executes the SQL statements on the database. The `csql` interface is invoked as below –

```
$ csql  
CSQL>
```

Refer Section 10 for more about CSQL tools.

3.3 SQL Data Definition Language (DDL)

The CSQL Tool supports standard DDL statements such as `CREATE TABLE` and `DROP TABLE`.

3.3.1 CREATE TABLE

`CREATE TABLE` creates a table in the database.

Syntax:

```
CREATE TABLE <table name>  
( <col1> <datatype> constr,  
  <col2> <datatype> constr,
```

```

    . . . ,
    [primary key (col1)];

```

Where

col1, col2 - are the columns in the table

datatype - type of the data that the column represents,

constr - the constraint that applies to the column.

- **DATATYPES**

The Datatypes supported by CSQL are

Datatype name	Description
CHAR(size)	A string of fixed length
TINYINT	A 8-bit signed integer value
SMALLINT	A 16-bit signed integer value
INTEGER or INT	A 32-bit signed integer value. The range of INTEGER is -2147483648 to 2147483647
BIGINT	A 64-bit signed integer value
FLOAT	A 64-bit precision floating point value. These types are analogous to the Java double type
REAL	A higher precision numeric value
DATE	A date value in month/day/year
TIME	A time of day value
TIMESTAMP	A month/day/year and time of day value
BINARY(size)	A binary value of fixed length

- **CONSTRAINT**

At present only NOT NULL constraint is supported by CSQL in CREATE TABLE statement. The UNIQUE constraint is supported through the CREATE INDEX statement.

Syntax:

```

create table t1
(
    f1 int, f2 char(20),
    f3 float,
    primary key (f1));

```

f1 is a primary key and it will be NOT NULL and UNIQUE.

```

create table t2
(
    f1 int Not Null,

```

```
f2 char(30) Not Null);
```

f1 and f2 are not null fields.

CSQL supports only one primary key field per table and it should be mentioned at the end of the table definition.

3.3.2 DROP TABLE

DROP TABLE removes the table from the database.

Syntax:

```
DROP TABLE <table name>;
```

3.4 SQL Data Manipulation Language (DML)

CSQL tool supports standard DML statements such as INSERT, UPDATE and DELETE commands to store, modify and remove data from the database.

3.4.1 INSERT

The INSERT command adds one row at a time into the table.

Syntax:

```
INSERT INTO <table-name> [(column-list)]  
VALUES (value list)
```

This form has an optional column-list specification. Only the columns listed will be assigned values. Unlisted columns are set to null, so these columns must allow null values. The values from the VALUES Clause are assigned to the corresponding column in column-list in order.

If the optional column-list is missing, the default column list is substituted. The default column list contains all columns in table-name in the order they were declared in CREATE TABLE.

Table T1 before INSERT –

F1	f2	f3
----	----	----

```
INSERT INTO t1 VALUES (1001, 'Ravi', 5000.00);
```

Table T1 after INSERT –

F1	f2	f3
1001	Ravi	5000.00

```
INSERT INTO t1(f1) VALUES(1002);
```

Table T1 after INSERT –

F1	f2	f3
1001	Ravi	5000.00
1002	NULL	NULL

Presently CSQL adds only one row at a time.

3.4.2 UPDATE

The UPDATE statement modifies column values in selected table rows. It has the following general format:

```
UPDATE <table-name>  
SET <set-list>  
[WHERE predicate]
```

The optional WHERE clause chooses which table rows to be updated. If it is missing, all rows in table-name are updated. The set-list contains assignments of new values for selected columns.

- SET clause

The SET clause in the UPDATE statement updates (assigns new value to) columns in the selected table rows. It has the following general format:

```
SET col1 = value1  
[, col2 = value2]  
...
```

- WHERE Clause

The WHERE clause specifies which rows needs to be updated. If it is not specified, then all the rows in the table are updated.

col1 and col2 are columns in the table. value1 and value2 are expressions that can reference columns from the table which is being updated. They also can be the keyword -- NULL. Since the assignment expressions can reference columns from the current row, the expressions are evaluated first. After the values of all

set expressions have been computed, they are then assigned to the referenced columns.

```
UPDATE t2 SET qty=100;
```

T2 before updation –

no	item	qty
1	Bolts	10
2	Nuts	20

T2 after updation –

no	item	qty
1	Bolts	100
2	Nuts	100

```
UPDATE t1 SET qty = 150 WHERE item='Nuts';
```

T2 before updation –

no	item	qty
1	Bolts	100
2	Nuts	100

T2 after updation –

no	item	Qty
1	Bolts	100
2	Nuts	150

3.4.3 DELETE

The DELETE Statement removes selected rows from a table.

Syntax:

```
DELETE FROM <table-name>
[WHERE predicate]
```

The optional WHERE Clause has the same format as in the UPDATE Statement.

The WHERE clause specifies which rows needs to be deleted. If it is not specified, then all the rows in the table are removed.

```
DELETE FROM t2 WHERE no=2;
```

T2 before deletion –

no	item	Qty

1	Bolts	100
2	Nuts	150

T2 after deletion –

No	item	Qty
1	Bolts	100

```
DELETE FROM t2;
```

T2 before deletion –

no	item	Qty
1	Bolts	100

T2 after deletion –

no	item	Qty
----	------	-----

3.5 SQL Data Query Language (DQL)

CSQL tool supports standard DQL statement, `SELECT` to view the data present in the table.

3.5.1 SELECT

The SQL `SELECT` statement queries data from tables in the database. The statement begins with the `SELECT` keyword. The basic `SELECT` statement has 3 clauses:

- `SELECT`
- `FROM`
- `WHERE`

The `SELECT` clause specifies the table columns that are to be retrieved.

The `FROM` clause specifies the table to be accessed.

The `WHERE` clause specifies which table rows are selected. It is optional; if missing, all table rows are selected.

Syntax:

```
SELECT <select-list>
FROM <table-name>
[WHERE predicate]
```

If select-list is specified as *, then it projects all the columns for those rows that satisfy the where clause.

```
SELECT *
FROM <table-name>
[WHERE predicate]
```

To view the records of a table named tab

No	Name	Age	Wt
1	Tom	25	60
2	Dick	30	70
3	Harry	35	80

```
SELECT Name, Age FROM tab
WHERE Wt < 75;
```

Result set of the above select statement –

Name	Age
Tom	25
Dick	30

```
SELECT Name, Wt FROM tab WHERE Age > 25;
```

Output –

Name	Wt
Dick	70
Harry	80

```
SELECT Name FROM tab WHERE Wt < 100;
```

Output –

Name
Tom
Dick
Harry

```
SELECT * FROM tab;
```

Output –

No	Name	Age	Wt
1	Tom	25	60
2	Dick	30	70
3	Harry	35	80

4. JDBC Driver

This section explain various Interfaces and methods in JDBC APIs and their uses, which would help in writing applications to access CSQL main memory database using the CSQL's JDBC Driver.

The JDBC (Java Database Connectivity) is one of the standard interfacing subsystems in CSQL Database. It supports most of the JDBC 2.0 APIs and all-primitive data types, along with the `date`, `time` and `timestamp`.

4.1 What is JDBC

JDBC is an API (Application Programming Interface) specification, which consists of a set of classes, interfaces and exceptions used in JAVA application development for accessing any database management system.

Using these standard interfaces and classes, programmers can write applications that connect to CSQL, send queries written in SQL and process the results.

4.1.1 Implemented Interfaces

The JDBC API is consistent with the style of the core Java interfaces and classes, such as `java.lang` and `java.awt`. The table below describes the interfaces, classes and exception classes that make up the JDBC API.

Interface/class/exception	Description
Interfaces:	
<code>java.sql.Connection</code>	Interface used to establish a connection to CSQL.SQL statements run within the context of a connection.
<code>java.sql.PreparedStatement</code>	Interface used to send precompiled SQL statements to the database driver and obtain results.
<code>java.sql.ResultSet</code>	Interface used to process the results returned after executing an SQL statement.
<code>java.sql.Statement</code>	Interface used to send static SQL statements to the database server
Classes:	
<code>java.sql.Date</code>	Subclass of <code>java.util.Date</code> used for the SQL DATE data type.

<code>java.lang.DriverManager</code>	Class used to manage a set of JDBC drivers.
<code>java.sql.Time</code>	Subclass of <code>java.util.Date</code> used for the SQL TIME data type.
<code>java.sql.Timestamp</code>	Subclass of <code>java.util.Date</code> used for the SQL TIMESTAMP data type.
Exception classes:	
<code>java.sql.SQLException</code>	Exception that provides information about a database error.

Because JDBC is a standard specification, any Java program that uses the JDBC API can connect to CSQL using the JDBC driver.

What is a JDBC driver

The JDBC API defines the Java Interfaces and Classes that programmers use to connect to CSQL and send queries.

Basically, JDBC consists of two parts:

- **JDBC API:**

It provides a programmatic access to relational data from the Java programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source.

- **JDBC Driver Manager:**

The JDBC `DriverManager` class defines objects, which can connect Java applications to a JDBC Driver. It has traditionally been the backbone of the JDBC architecture.

4.1.2 JDBC Datatype

This JDBC 2.0 supports all primitive types like – `integer`, `char`, `float`, `string`, including `Date`, `Time`, `TimeStamp`.

The table below shows the JDBC prescribed “SQL-to-Java datatype” mappings.

SQL datatypes supported in JDBC 2.0:

SQL Type (from <code>java.sql.types</code>)	Java Type
TINYINT	Byte
SMALLINT	Short
INTEGER	Int
BIGINT	Long
REAL	Float
FLOAT DOUBLE	Double
CHAR	<code>java.lang.String</code>
DATE	<code>java.sql.Date</code>
TIME	<code>java.sql.Time</code>
TIMESTAMP	<code>java.sql.Timestamp</code>

These mappings are the JDBC specification for direct type mapping.

Lets take an example where you create a table in the CSQL with two fields,

```
create table t1
(f1 integer, f2 char(20));
```

The following sections describe how to connect to the CSQL and to create a table 'T1' in it, and perform insertion, updation, fetch and deletion of records and ultimately drop the table 'T1' from CSQL Database.

Please refer to the `jdbcxample.java` file for source code, which is present in the `jdbc` sub-directory of `example` directory.

4.2 Establishing a Connection

First, you need to establish a connection with the CSQL. Establishing a connection involves two steps: Loading the driver, and making the connection.

4.2.1 Loading the Driver:

The `jdbcxample.java` program must first open a connection to a database, and can then execute SQL statements. But before opening a connection, it is necessary to load the appropriate drivers for the database by using `Class.forName`.

Loading the driver you want to use is very simple. It involves just one line of code in your program. To add the appropriate Driver, add the following line of code:

```
Class.forName("csql.jdbc.JdbcSqlDriver");
```

In the above code, `JdbcSqlDriver` is the driver for CSQL.

Calling the `class.forName` automatically creates an instance of a driver and registers it with the `DriverManager`, so you don't need to create an instance of the class.

Now you have loaded the driver and it can make a connection with a DBMS.

4.2.2 Making the Connection

The second step in establishing a connection is to have the appropriate driver connect to the CSQL.

- Using the `DriverManager` Class

The `DriverManager` class works with the driver interface to manage the set of drivers available to a JDBC Client. When the client requests a connection and provides a URL, the `DriverManager` is responsible for finding a driver that recognizes the URL and connects to the data source. The connection URL would have the following syntax:

```
"jdbc:csql"
```

The `jdbc:csql` portion of the URL identifies the database.

- The `getConnection` method establishes a connection:

Actually basic database access starts with the `Connection` object, when this object is created this is simply a direct link to the database.

```
Connection con =  
    DriverManager.getConnection("jdbc:csql", "root",  
    "manager");
```

You need to use `root` for the user name and `manager` for the password. This will establish a connection with CSQL.

If the loaded driver recognizes the parameters supplied to the method `DriverManager.getConnection`, then it establishes a connection to the CSQL database. The `DriverManager` class encapsulates and manages the connection process.

The connection returned by the method `DriverManager.getConnection()` is connection to the CSQL Database. Now we can create JDBC functions that pass the SQL statements to CSQL using this connection.

4.3 Creating Table

Now you can create a statement handle on the `Connection` and use it to execute an SQL statement. Once you are connected, you can create a new SQL statement object by using the `Connection` object.

```
Statement cStmt = con.createStatement();
```

The `Statement` class provides an `execute()` method to execute the SQL Statements.

You can then execute this statement to create a table `T1` with two fields in the CSQL and we do that by executing a `CREATE SQL` statement.

```
cStmt.execute("CREATE TABLE T1 (f1 integer, f2 char (20));");
```

4.3.1 Creating Index

You can use the same statement object to create a unique index.

```
cStmt.execute("CREATE INDEX IDX ON T1 ( f1);");
```

After executing this SQL Statement, it will be creating a unique index 'IDX' for `f1` column.

4.3.2 Closing and committing the Statement object

```
cStmt.close();
```

The code closes the `Statement` object `cStmt` through a `close()` method. This `cStmt` object holds database resources, so its good to close any instance of this object when we are done with them.

In the code snippet above, you used one statement object, which is generated by `Connection`. `Connection` object provides methods to commit (or rollback) all the statement executed in that connection context.

```
con.commit();
```

In the example `jdbccexample.java`, you can observe that the `Statement` object is used for SQL Create and Drop statements. And for other SQL statements like `INSERT`, `UPDATE`, `DELETE`, you use the `PreparedStatement` interface. Because, each SQL statement needs to be parsed by the database engine, a corresponding query plan (“query-tree”) has to be formulated before it can actually be executed.

If each DDL statement needs a new query plan then the database builds one for it.

If statements, which generate the same query plan, were being executed consecutively, then going by the above criteria you would waste processing cycles. In order to get rid of this, you use the `PreparedStatement` interface, which ensures a one-time creation of query plan and subsequently the same query plan is just re-used whenever needed. Refer to example `jdbccexample.java`.

4.4 Inserting tuples

In the last example, you created a table T1 with two fields. Now we will try to insert 10 tuples (rows) with different values for both fields.

```
PreparedStatement stmt = null, selStmt= null;
stmt = con.prepareStatement("INSERT INTO T1 (f1, f2)
VALUES (?, ?);");
int count =0;
int ret =0;
for (int i =0 ; i< 10 ; i++) {
    stmt.setInt(1, i);
    stmt.setString(2, String.valueOf(i+100));
    ret = stmt.executeUpdate();
    if (ret != 1) break; //error
    count++;
}
stmt.close();
con.commit();
System.out.println("Total Rows inserted " + count);
```

The steps involved in the record insertion code snippet above are –

- Use the `prepareStatement` function to generate the one-time query plan.
- Bind the parameters with the respective fields and execute the prepared statement (the same query plan is re-used).

- Close the `PreparedStatement` object
- Commit the transaction

The `prepareStatement` method:

The `prepareStatement` method enables a SQL statement to contain parameters, and you can execute a single statement repeatedly with different values for those parameters and to assign values to these parameters is called binding.

```
stmt = con.prepareStatement("INSERT INTO T1 (f1, f2)
VALUES (?, ?);");
```

Binding the parameters and executing the prepared statement:

Using the `PreparedStatement` object, you pass the SQL statement to the database through the `prepareStatement()` method in `java.sql.Connection`. You execute the resulting “prepared” SQL statement multiple times inside the `for` loop, the query plan had been built once at the start.

```
stmt.setInt(1, i);
stmt.setString(2, String.valueOf(i+100));
ret = stmt.executeUpdate();
```

Before each execution of the prepared statement, you pass to JDBC the values to be used as input for that execution cycle. In order to bind the input parameters, `PreparedStatement` provides `setInt()` and `setString()` method. These methods bind the parameters from left to right in the order you placed them in the prepared statement.

Close the `PreparedStatement` object:

The `PreparedStatement` object is closed with the `close()` method.

```
stmt.close();
```

Closing `stmt` object implicitly closes instances associated with it and frees the associated memory.

Commit the transaction:

The `Connection` object sets up the connection with the CSQL database. In order to close the connection you use the `commit` method.

```
con.commit();
```

You can use the `con` object to generate implementations of `java.sql.Statement` tied to the same database transaction.

4.5 Updating Rows

You would now look into a code snippet for updating rows in table ‘T1’.

You create a `PreparedStatement` object that can receive 2 parameters using the `Connection` method - `prepareStatement`:

```
stmt = con.prepareStatement("UPDATE T1 SET f2 = ?
WHERE f1 = ?;");
for (int i =0 ; i< 10 ; i +=2) {
    stmt.setString(1, String.valueOf(i+200));
    stmt.setInt(2, i);
    ret = stmt.executeUpdate();
    if (ret != 1) break; //error
    count++;
}
stmt.close();
con.commit();
```

The variable `stmt` contains the SQL Statement, `UPDATE T1 SET f2=? WHERE f1=?`, which is sent to the DBMS and precompiled into a query plan.

Here the `f1` and `f2` fields can be supplied with values using the `setString` and `setInt` method in accordance to the column positions in the prepared statement from left to right.

Subsequently the `executeUpdate` method executes the update SQL statement at the database level.

4.6 Deleting Rows

In this example you will delete some rows from the table “T1” using `PreparedStatement` parameters.

```
stmt = con.prepareStatement("DELETE FROM T1 WHERE f1 =
?;");
for (int i =0 ; i< 10 ; i +=3) {
    stmt.setInt(1, i);
    ret = stmt.executeUpdate();
    if (ret != 1) break; //error
    count++;
}
```

```
stmt.close();  
con.commit();
```

In the prepared statement the `f1 = ?` signifies that the rows to be deleted will be based on the parameter being passed through `setInt()` method.

Post deletion, you commit the Statement objects associated with that Connection.

4.7 Fetching the Rows

You can create a `PreparedStatement` object based on a result-set generated through a `SELECT` query, for example –

```
PreparedStatement selStmt= null;  
selStmt = con.prepareStatement("SELECT * from T1 where  
f1 = ?;");
```

The `executeQuery()` method of the statement object is used to generate the result set. It returns an object that implements the `ResultSet` interface. The following code snippets depicts the above –

```
ResultSet rs = null;  
for (int i =0 ; i< 10 ; i++) {  
    selStmt.setInt(1, i);  
    rs = selStmt.executeQuery();  
    while (rs.next())  
    {  
        System.out.println("Tuple value is " +  
            rs.getInt(1)+ " "+ rs.getString(2));  
        count++;  
    }  
    rs.close();  
}  
selStmt.close();  
con.commit();
```

The `ResultSet` interface provides methods for retrieving and manipulating the results of executed queries, and these objects can have different functionality and characteristics. The result set is generated upon executing query statements on the table data.

The code snippet above displays how to generate the result set using a query statement through a Java program.

The variable `rs`, (in the code snippet above) is an instance of `ResultSet`, and contains the rows of `T1` which have been “selected”. In order to access the `f1` and `f2` field values, the `ResultSet` object maintains a cursor, which points to its current row of data.

When a `ResultSet` object is created, the cursor is positioned before the first row. To move the cursor down the rows, we use the following methods:

- **`next ()`** --- moves the cursor forward by one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.

You can iterate through the `ResultSet` to obtain values for each field of the `T1` table and print these values on the user console.

```
while (rs.next())
System.out.println("Tuple value is " + rs.getInt(1)+ "
"+ rs.getString(2));
```

At the end, close the `ResultSet`, `PreparedStatement` and `Connection` objects as depicted below –

```
rs.close();
selStmt.close();
con.commit();
```

Once the rows are fetched, commit the transaction.

4.8 Drop the table

In order to drop a table from the `CSQL` database –

```
cStmt.execute("DROP TABLE T1;");
```

Once this is executed, it will drop the `T1` table from the `CSQL` Database.

4.9 Close the connection

```
cStmt.close();
```

When `close ()` method is invoked, it implicitly closes all `PreparedStatement` instances associated with the `Connection`.

5. ODBC Driver

This section explains various functions in ODBC API and their uses, which would help in writing applications to access the CSQL database using the CSQL ODBC driver.

The ODBC (Open Database Connectivity) is the other standard interfacing subsystem in CSQL Database (the other one is JDBC). CSQL supports most of ODBC 2 APIs and all primitive data types, including Date, Time and TimeStamp data types.

Let us start with an example where you create a table in the CSQL database to store employee details, lets say the table is EMP and the fields are empId (int), name (char(20)) and sal (float).

The following sections describe how to connect to the database, how to create table, how to insert, update and delete tuples in the table and how to drop table.

5.1 Why ODBC

Due to the inherent limitations in SQL in performing certain complex computations, they must be written in a host language, such as C or C++, with embedded SQL queries which in turn access the data in the CSQL database.

To access the CSQL, the SQL statements need to be executed from within the host language application, using an API, which can be used to send DML and DDL statements to the CSQL and retrieve the results. This programming interface is called the ODBC API.

The ODBC API is a library of ODBC functions that lets ODBC-enabled applications to connect to CSQL, execute statements, and retrieve results. For doing this an ODBC driver is available.

The goal of ODBC is to make data access possible in any database from any application, this is achieved by inserting a middle layer called the “database driver” between an application and the DBMS. This layer translates the application’s data queries into commands that the DBMS understands.

5.2 Components of ODBC

- An ODBC compliant application i.e. an application that uses the ODBC API to talk to CSQL (DBMS).
- The ODBC Driver manager is a repository containing the list of installed ODBC drivers and data sources. It is the interface between an ODBC application and an ODBC Driver. Applications requiring ODBC access, interface with the driver

manager and make ODBC API calls, which causes the driver manager to load the appropriate ODBC Driver.

- An ODBC driver translates the ODBC API calls into something that the backend CSQL understands.
- Header files required to build the ODBC application – `stdlib.h`, `sql.h`, `sqlext.h` & `sqlucode.h`

Normally C applications need to include –
`sql.h` – which contains most of the definitions you'll need
`sqlext.h` – which contains mostly additions for ODBC 3
`sqlucode.h` is automatically included by `sqlext.h` and
`sqltypes.h` is automatically included from `sql.h`

5.3 ODBC API Overview

Before you get into the various functions provided by the ODBC API, let's look at a few key concepts. In this section, you will look at allocating various handles that are used by ODBC and some important APIs which are frequently used in any ODBC application.

5.3.1 ODBC Handles

The ODBC API introduces new handle types that are used to reference information about your application's ODBC environment, specific database connections and SQL statements.

In ODBC 3, each of these handle types are allocated with a single function `SQLAllocHandle()` and freed with a single function `SQLFreeHandle()`

In ODBC, there are three main handle types, which you need to know in order to do access the data.

- `SQLHENV` – environment handle
This is the first handle you will need as everything else is effectively in the environment. Once you have an environment handle you can define the versions of ODBC you require, enable connection pooling and allocate connection handles.
- `SQLHDBC` – connection handle
You need one connection handle to make the connection. Like environment handles, connection handles have attributes which you can retrieve and set.

- `SQLSTMT` – statement handle
Once you have a connection handle and have connected to CSQL you allocate handles to execute SQL or retrieve data. As with the other handles you can set and get statement attributes with this handle also.

SQLAllocHandle:

`SQLAllocHandle` is a generic function for allocating environment, connection and statement handles.

Prototype of the Function:

```
SQLRETURN SQLAllocHandle(SQL_SMALLINT HandleType,
    SQLHANDLE InputHandle, SQLHANDLE * OutputHandlePtr);
```

The arguments for `SQLAllocHandle` are listed in below table

Type	Name	Description
<code>SQLSMALLINT</code>	<code>HandleType</code>	The type of handle to allocate.
<code>SQLHANDLE</code>	<code>InputHandle</code>	The handle to base on the new handle. This is either an environment or connection handle. To create a new handle from scratch, pass in NULL.
<code>SQLHANDLE*</code>	<code>OutputHandle</code>	Pointer to the storage for the newly create handle.

Note: - An application allocates different handles to use with different API functions. The handle provides a context for each function .The supported handle types are.

Environment	<code>SQL_TYPE_ENV</code>	These handles are used to create an environment. Each environment contains generic information that allows you to access the CSQL. A new transaction is associated with a newly-created environment handle.
Connection	<code>SQL_TYPE_DBC</code>	A connection handle is used to open a connection to a specific CSQL Database. Connections can be based on the same environment handle, hence sharing the same transaction across multiple database connections. However, a maximum of eight connections can share a single environment.
Statement	<code>SQL_TYPE_STMT</code>	The statement handle contains information about the compiled SQL statement and its result sets.

Returns: `SQLAllocHandle` returns `SQL_SUCCESS` if it is successful. Otherwise, it returns `SQL_ERROR`.

SQLFreeHandle:

`SQLFreeHandle` is a generic function to free environment, connection, and statement handles.

Prototype of the Function:

```
RETCODE SQLFreeHandle(SQLSMALLINT handleType,
SQLHANDLE handle)
```

SQLFreeHandle Arguments:

Type	Name	Description
SQLSMALLINT	handleType	The type of handle to free.
SQLHANDLE	handle	The handle to free.

Returns

`SQLFreeHandle` returns `SQL_SUCCESS` if it is successful. Otherwise, it returns `SQL_ERROR`.

5.4 ODBC API

In this section, we describe some important APIs .

5.4.1 SQLPrepare

`SQLPrepare` prepares an SQL String for execution. In our examples you have used this function for DML statements.

ODBC allows us to prepare SQL statements in a separate step so that you can generate the query plan once after the parsing and for every subsequent execution you could simply re-use the same query plan.

Prototype of the function:

```
SQLRETURN SQLPrepare (
SQLHSTMT StatementHandle,
SQLCHAR * StatementText,
SQLINTEGER TextLength);
```

Arguments:

StatementHandle: [Input] Statement handle
 StatementText : [Input]SQL text string
 TextLength : [Input] Length of *Statement Text in Characters.

This function returns `SQL_SUCCESS`, if it successfully prepares the SQL Statements.

5.4.2 SQLExecute

`SQLExecute` executes a prepared statement using the current values of the parameter marker variables if any exists in the statement.

Prototype of the Function:

```
SQLRETURN SQLExecute(SQLHSTMT StatementHandle)
```

Arguments:

StatementHandle : [Input] StatementHandle.

This function returns `SQL_SUCCESS` on successful execution.

5.4.3 SQLBindParameter

It binds a buffer to a parameter marker in an SQL statement.

You will see in following sections that how we bind the parameter using this function for Insert, Update and Delete statements.

```
SQLRETURN SQLBindParameter(
    SQLHSTMT StatementHandle,
    SQLUSMALLINT ParameterNumber,
    SQLSMALLINT InputOutputType,
    SQLSMALLINT ValueType,
    SQLSMALLINT ParameterType,
    SQLULEN ColumnSize,
    SQLSMALLINT DecimalDigits,
    SQLPOINTER ParameterValuePtr,
    SQLINTEGER BufferLength,
```

```
SQLLEN *          StrLen_or_IndPtr );
```

Arguments:

StatementHandle	: [Input] Statement handle.
ParameterNumber	: [Input] Parameter number, sequentially increasing parameter order starting at 1.
InputOutputType	: [Input] The type of the parameter. SQL_PARAM_INPUT is used for parameters in SQL Statements and SQL_PARAM_OUTPUT or SQL_PARAM_INPUT_OUTPUT are used for stored procedure parameters.
ValueType	: [Input] The C data type of the parameter.
ParameterType	: [Input] The SQL data type of the parameter.
ColumnSize	: [Input] The size of the column.
DecimalDigits	: [Input] The decimal digits of the column or expression of the corresponding parameter marker .
ParameterValuePtr	: [Deferred Input] A pointer to a buffer for the parameter's data.
BufferLength	: [Input/Output] Length of the parameterValuePtr buffer in bytes.
StrLen_or_IndPtr	: [Deferred Input] A pointer to a buffer for the parameter's length.

This function returns SQL_SUCCESS, if it successfully binds the parameter.

5.4.4 SQLBindCol

This function binds application data buffers to columns in the result set. You use this function for select statements.

```
SQLRETURN SQLBindCol(
    SQLHSTMT      StatementHandle,
    SQLUSMALLINT  ColumnNumber,
    SQLSMALLINT   TargetType,
    SQLPOINTER    TargetValuePtr,
    SQLLEN        BufferLength,
    SQLLEN *      StrLen_or_Ind);
```

Arguments:

StatementHandle	: [Input] Statement handle
-----------------	----------------------------

ColumnNumber	: [Input] Number of the result set column to bind. Columns are numbered in increasing column order starting at 1.
TargetType	: [Input] The identifier of the C datatype of the *TargetValuePtr Buffer, when it is retrieving data from the data source with SQLFetch.
TargetValuePtr	: [Deferred Input/Output] pointer to the data buffer to bind to the column. SQLFetch return data in this buffer.
BufferLength	: [Input] Length of the targetValuePtr buffer in bytes.
StrLen_or_IndPtr	: [Deferred Input/Output] Pointer to the length buffer to bind to the column.

This function returns `SQL_SUCCESS`, if it successfully binds the column.

5.5 Data Types

CSQL supports all primitive Data types like `int`, `float`, `char` etc and `Date`, `Time`, `TimeStamp`.

ODBC uses two sets of data types – SQL datatypes and C Datatypes. SQL data types are used in the data source and C data types are used as part of the host language application.

SQL Type Identifier: SQL data types are the types in which data is stored in the data source.

For example, `SQL_CHAR` is the type identifier for a character column with a fixed length, typically between 1 and 254 characters. These characteristics correspond to the `CHAR` data type found in data source. Thus when application discovers that the type identifier for a column is `SQL_CHAR`, it can assume it is probably dealing with a `CHAR` column.

In our sample ODBC source code, you create an `emp` table with three fields. Field `eid` is integer, `ename` is character and `salary` is float, for these three fields the SQL Identifiers are `SQL_SHORT`, `SQL_CHAR`, `SQL_FLOAT`.

C Type Identifier: ODBC also defines the C data types that are used by application variables and their corresponding type identifiers. The buffers that are bound to the result set columns and statement parameters use these. For example, an application wants to retrieve data from a result set column in character format. It declares a variable with the `SQLCHAR*` data types and binds this variable to the result set column with type identifier of `SQL_C_CHAR`.

Here also for the three fields, the data types will be `SQL_C_SHORT`, `SQL_C_CHAR`, `SQL_C_FLOAT` for integer, character, float respectively.

SQL Definition	SQL Type Identifier	C Type Identifier
<code>CHAR(n)</code>	<code>SQL_CHAR</code>	<code>SQL_C_CHAR</code>
<code>SMALLINT</code>	<code>SQL_SMALLINT</code>	<code>SQL_C_SSHORT</code>
<code>INTEGER</code>	<code>SQL_INTEGER</code>	<code>SQL_C_SLONG</code>
<code>REAL</code>	<code>SQL_REAL</code>	<code>SQL_C_FLOAT</code>
<code>FLOAT</code>	<code>SQL_FLOAT</code>	<code>SQL_C_FLOAT</code>
<code>DOUBLE</code>	<code>SQL_DOUBLE</code>	<code>SQL_C_DOUBLE</code>
<code>TINYINT</code>	<code>SQL_TINYINT</code>	<code>SQL_C_TINYINT</code>
<code>BIGINT</code>	<code>SQL_BIGINT</code>	<code>SQL_C_SBIGINT</code>
<code>DATE</code>	<code>SQL_TYPE_DATE</code>	<code>SQL_C_TYPE_DATE</code>
<code>TIME</code>	<code>SQL_TYPE_TIME</code>	<code>SQL_C_TYPE_TIME</code>
<code>TIMESTAMP</code>	<code>SQL_TYPE_TIMESTAMP</code>	<code>SQL_C_TYPE_TIMESTAMP</code>

5.6 ODBC API with Examples

Let us start with an example where you create a table in the CSQL database to store employee details, say `emp(eid int, ename char(20), salary float)`.

5.6.1 Connect to the CSQL

Refer to `ODBCman1.c` for the source code

You have to allocate the internal structures for the various handle types through

`SQLAllocHandle` function.

```

SQLHENV    env;
SQLHDBC    dbc;
SQLRETURN  ret;

ret = SQLAllocHandle (SQL_HANDLE_ENV,
SQL_NULL_HANDLE, &env);

ret = SQLSetEnvAttr
(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3,
0);

ret = SQLAllocHandle (SQL_HANDLE_DBC, env, &dbc);

```

Firstly, `SQLAllocHandle` function allocates an environment handle.

`SQL_HANDLE_ENV` is used to create an environment and a new transaction is associated with it.

`SQL_NULL_HANDLE` specifies the structure from which a new handle is derived. An environment handle, however, isn't derived from another handle, so `InputHandle` should be set to `SQL_NULL_HANDLE`.

The `OutputHandle` pointer – `env` should point to the new handle that is to be allocated.

Secondly, `SQLSetEnvAttr` function sets the application's ODBC Version. The ODBC driver manager is designed to support version 2 drivers and applications, as well as newer *ODBC 3* components. Depending on the version of ODBC that your application is using, certain functionalities will behave differently. This requires that our application specify which version of the ODBC API it is using before you go on to allocate Connection handles. The version can be set using `SQLSetEnvAttr()` to set the `SQL_ATTR_ODBC_VERSION` environment attribute to `SQL_OV_ODBC3`.

Thirdly, `SQLAllocHandle` function allocates a Connection handle.

`SQL_HANDLE_DBC` is used to open a connection to a specific CSQL database.

The input handle `env` specifies the handle from which the Connection handle is allocated. The output handle ptr 'dbc' is allocated.

If the new handle is successfully allocated, `SQLAllocHandle()` will return `SQL_SUCCESS`; otherwise, it will return `SQL_ERROR`.

After you allocate a connection handle with `SQLAllocConnect()`, you must connect the handle to a data source before you can start operating on that data source.

ODBC functions to connect to a data source

`SQLConnect` :Loads a driver and establishes a connection to data source.

`SQLDriverConnect` :Connects to a specific driver using a connection string or requests the Driver Manager to connect to the specified DSN (Data Source Name).

SQLConnect

This function provides the most direct programmable control of the connection.

Prototype for **SQLConnect ()**.

```
SQLRETURN SQLConnect
    (SQLHDBC ConnectionHandle, SQLCHAR*
    ServerName, SQLSMALLINT NameLength1, SQLCHAR*
    Username, SQLSMALLINT NameLength2, SQLCHAR*
    Authentication, SQLSMALLINT NameLength3);
```

ConnectionHandle is allocated with `SQLAllocHandle()`.

ServerName passes the name or URL of the database server.

Username passes the login user id

Authentication passes the password.

The username and password can be passed as NULL if the DSN resides on the local machine.

Connect to CSQL:

```
rc = SQLConnect (dbc,
    (SQLCHAR *) "test", (SQLSMALLINT)
    strlen ("test"),
    (SQLCHAR *) "root",
    (SQLSMALLINT) strlen ("root"),
    (SQLCHAR *) "",
    (SQLSMALLINT) strlen (""));
```

When you call **SQLConnect ()**, the ODBC Driver manager will load the requested driver, if it isn't already loaded and will connect to the requested data source. If an error occurs, **SQLConnect ()** will return **SQL_ERROR**.

SQLDriverConnect :

It is an alternative to `SQLConnect`. It supports data sources that require more information. And those that are not defined in the `ODBC.INI` file.

```
SQLCHAR outstr[1024];

SQLSMALLINT outstrlen;

rc = SQLDriverConnect (dbc,
    NULL, (SQLCHAR*)"DSN=myodbc3;", SQL_NTS, outstr,
    sizeof(outstr), &outstrlen, SQL_DRIVER_NOPROMPT);
```

dbc – allocated connection handle with `SQLAllocHandle`.

NULL – is set for the window handle for the any dialog boxes that may be created.

`DSN=myodbc3` – `InConnectionString` parameter which points to a connection string that is passed into `SQLDriverConnect`.

All string parameters that are passed as inputs to ODBC functions will consist of a pointer to the string and a separate parameter for its length, which is used to support languages that require this. For C/C++ applications, you should pass a pointer to a null-terminated string and set the length parameter to `SQL_NTS` (Null-Terminated String).

`outstr` – can extract status of data source and its attributes.

`outstrlen` – records the buffer length.

`SQL_DRIVER_NOPROMPT` disables user interaction.

When the connection to the data source is established, `SQLDriverConnect()` will return the actual connection string that was used in `OutConnectionString`.

5.6.2 Transactions

So far, each SQL statement has been atomic, i.e., each statement can stand on its own and if one fails, others are not affected and the database is left in a consistent state. By default, ODBC is in “auto-commit” mode, where each statement is either committed as soon as it succeeds or rolled back if it fails.

SQLTransact

It requests a commit or rollback operation for all active operations on all statements associated with a connection. `SQLTransact` can also request that a commit or rollback operation be performed for all connections associated with the environment.

Prototype of `SQLTransact`:

```
RETCODE SQLTransact (env, dbc, ftype);
```

`env` : input environment variable.
`dbc` : input connection handle
`ftype` : input one of the following two values : `SQL_COMMIT`, `SQL_ROLLBACK`.

When you connect to a database, a transaction starts automatically; you can execute as many SQL statements as you need. Once these SQL statements are processed, if you want to commit a transaction, which means all changes made by

DML statements will be reflected in the table, call `SQLTransact` with the option `SQL_COMMIT`. In order to abort a transaction, replace `SQL_ROLLBACK` with `SQL_COMMIT`.

ODBC Commit Modes:

In ODBC, transactions can be handled in two different ways. The connection can be set to either auto-commit mode (the default) or manual-commit mode. The commit mode for a Connection is set by calling `SQLSetConnectAttr()` with the `SQL_ATTR_AUTOCOMMIT` option.

Auto-commit Mode – The default mode for a new connection is auto-commit, which is supported by all drivers. In this mode, each statement operates as a separate transaction; the driver will take care of committing each operation on the database automatically.

If you submit a batch of SQL statements in a single `SQLExecute()` call, ODBC doesn't define whether this is treated as a single transaction or each statement is a separate transaction. If you want to send a batch as a transaction, use manual-commit mode.

Manual-Commit Mode – In cases where you want to ensure that multiple SQL statements be executed as ONE transaction, you should use manual-commit mode, which requires the application to explicitly end the transaction with a call to `SQLEndTran()`.

```
ret = SQLSetConnectAttr(dbc, SQL_ATTR_AUTOCOMMIT,  
    (void*)SQL_AUTOCOMMIT_OFF, SQL_IS_UIINTEGER);
```

Applications can specify the transaction mode with the `SQL_ATTR_AUTOCOMMIT` attribute. `SQL_AUTOCOMMIT_OFF` changes to manual commit mode.

To commit or roll back a transaction in manual-commit mode, an application calls `SQLEndTran`.

5.6.3 Create Table

Refer to `ODBCman2.c` for the source code

Once the connection is established you can execute the SQL statements against the connected data source.

Statement Handles: Before executing a statement, you must allocate a statement handle, which provides a data structure for ODBC to keep track of the SQL statement to be executed and the results it will return.

```
SQLHSTMT stmt;
ret = SQLAllocHandle (SQL_HANDLE_STMT, dbc,
&stmt);
```

SQL_HANDLE_STMT– handle type should be set to SQL_HANDLE_STMT

dbc – InputHandle should receive a previously allocated connection handle

&stmt – OutputHandle should point to a new handle of type SQLHSTMT that will be initialized.

In our examples, you will use `SQLExecuteDirect()` function for DDL Statements like creating or dropping tables as these statements will execute. On the other hand `SQLPrepare` and `SQLExecute` functions for DML Statements as they will be executed multiple times.

SQLExecDirect () :

For statements that will be executed only once, this is the fastest method of submitting SQL statements.

```
SQLCHAR table[100] = "create table emp(eid int,
ename char(20),salary float)";
ret = SQLExecDirect (stmt, table, SQL_NTS);
```

This function simply takes a null-terminated string (`table`) containing an SQL statement and executes it on the data source connected to the statement handle (`stmt`). The length parameter is set to `SQL_NTS`.

If `SQLExecDirect()` returns `SQL_SUCCESS`, then the statement was successfully executed against data source otherwise it returns `SQL_ERROR`.

5.6.4 Insert records into the table.

Refer to `ODBCman3.c` for the source code.

In the last session you saw “emp” table has been created with three fields. This table is present in the CSQL database. Now, you can insert some records in it using the appropriate APIs.

The stages for Inserting Records:

- Prepare
- Bind buffers
- Execute the Prepared Statement
- Commit the Transaction

Prepare the Statement

During the statement preparation, the ODBC standard SQL grammar which is passed to `SQLPrepare()` is translated into SQL for the data source.

ODBC allows the preparation of a statement prior to its submission, to facilitate the parsing and creation of the query plan once, which can subsequently be re-used multiple times with different parameters each time. This as earlier stated enhances efficiency.

```
ret = SQLPrepare(stmt, (unsigned char*)"insert
into emp values(?, ?, ?);", SQL_NTS);
```

This function takes a statement (`stmt`) handle as its first parameter which is previously allocated with `SQLAllocHandle()`; the second parameter is a pointer to null terminated string that contains the `INSERT` statement. `SQL_NTS` is the last parameter for text length.

Bind buffer

The `SQLBindParameter()` function allows to bind a buffer in memory to a given parameter marker, prior to the execution of the statement.

```
int eid1 = 1001;
char ename1[20] = "Ritish";
float salary1 = 2500;
size_t slen=strlen(ename1);

ret = SQLBindParameter(stmt, 1, SQL_PARAM_INPUT,
SQL_C_SHORT, SQL_INTEGER, 0, 0, &eid1, 0, NULL);

ret = SQLBindParameter(stmt, 2, SQL_PARAM_INPUT,
SQL_C_CHAR, SQL_CHAR, 196, 0, (void*)ename1,
slen, NULL);

ret = SQLBindParameter(stmt, 3, SQL_PARAM_INPUT,
SQL_C_FLOAT, SQL_REAL, 0, 0, &salary1, 0,
NULL);
```

The 1st parameter - `stmt` refers to the statement handle that you are using to execute the SQL statement.

2nd parameter specifies the parameter position in the SQL Statement. The positions are numbered as 1, 2, 3, ...etc from left to right.

The `SQL_PARAM_INPUT` parameter specifies how the parameter is used – input to write value to the field, `SQL_PARAM_OUTPUT` to read value from the field.

The 4th parameter is the `ValueType` parameter. This is a C type Identifier. ODBC defines many standard data types. There are C data types – used in the application code, and SQL data types – used to describe the type of data that is used within the data source. Many of the ODBC calls that move data from the application to the data source, or vice versa, can perform implicit type conversion.

Here you have used `SQL_C_SHORT`, `SQL_C_CHAR`, `SQL_C_FLOAT` for the corresponding fields – `eid`, `ename`, `salary` which are represented at the data source level by the SQL type identifiers – `SQL_INTEGER`, `SQL_CHAR`, `SQL_REAL` for this three field.

Parameters 6 and 7 are used to specify the size of the SQL parameter and its precision.

Parameter 8 is points to the buffer in the application that holds the value to be substituted in the SQL Statement,

Parameter 9 is used to pass the length of the buffer for binary or character parameters.

Parameter 10 is usually set to NULL pointer or pointer to length of the buffer for binary or character parameters. If it is set to NULL pointer, the driver assumes that all input parameter values are non-NULL and that character and binary data is null-terminated. This is not used internally in CSQL. So it is usually set to NULL in CSQL.

Executing SQL Statements

After the statement is prepared by `SQLPrepare()`, you can execute the statement by calling `SQLExecute()`.

```
ret = SQLExecute(stmt);
```

The `SQLExecute()` executes a prepared statement, using the current values of the parameter marker variables.

Committing the transaction

After inserting the data into the database, committing it will make sure that all the records inserted will be present in the database permanently.

```
ret = SQLTransact (env, dbc, SQL_COMMIT);
```

5.6.5 Fetch the records

Refer to ODBCman4.c for the source code.

The result set returned by a query is like a temporary table. These rows can be retrieved from the result set using cursors, which come in several different flavors. The default cursor used in ODBC is a forward-only cursor, which allows to access the rows in the result set only one row at a time.

The stages for Fetching:

- Prepare
- Bind buffer for column values
- Execute the prepared statement
- Fetch the row values.
- Close the cursor

Prepare the statement

It's the same as in the previous section.

```
ret = SQLPrepare(stmt, (unsigned char*)"select *  
from emp", SQL_NTS);
```

Binding Columns

This is done to assign the memory location into which a column's data would be copied when the row is fetched. In most cases, the best way to retrieve data from a result set is to bind the columns to specific memory locations / buffers that you have bound for that column.

This is done using `SQLBindCol()`.

```
ret=SQLBindCol(stmt,1,SQL_C_SHORT,&eid1,0,NULL);
```

```
ret=SQLBindCol(stmt,2,SQL_C_CHAR,ename1,  
sizeof(ename1),NULL);
```

```
ret= SQLBindCol(stmt,3,SQL_C_FLOAT,&salary,
```

```
0, NULL);
```

The `SQLBindColumn()` is called once each for the 3 fields.

The 2nd parameter is the field identifier to be bound. As mentioned earlier the number is from left to right in the table structure.

The 3rd parameter specifies the target data type.

The 4th parameter is the buffer or memory location which will holding the data being read from that field.

Parameter 5 is the sizeof the memory location or buffer.

Parameter 6 is the pointer to the length/indicator buffer to bind to the column. This is an input/output parameter and is not used internally in CSQL. So it is usually set to NULL in CSQL.

Execute the Statement

The SQL statements prepared before are executed.

```
ret = SQLExecute(stmt);
```

Fetch the results

The next step is to fetch the rows from the result set. `SQLFetch()` fetches the next row-set of data from the result set and returns the data for all bound columns.

The application here calls `SQLFetch()` to retrieve the first row of data and place the data from that row in the variables bound with `SQLBindCol()`.

```
while(SQL_SUCCEEDED(ret = SQLFetch(stmt)))
{
    printf("eid = %d  ename = %s salary = %f",
        eid1, ename1, salary);
    count++;
}
```

If you want to get the row count of a result set, simply scroll through it using `SQLFetch()` until no more records are found.

Closing the cursor

When you call a function that creates a result set, such as `SQLExecute()`, a cursor is opened, when you are finished working with the result set, you should close the cursor and free the memory by using `SQLCloseCursor()`.

```
ret = SQLCloseCursor(stmt);
```

5.6.6 Update Records

Refer to `ODBCman5.c` for the source code.

You can make changes to the tables by executing `UPDATE` statements. Update statements are often more efficient when they are used with parameters.

The Stages for update:

- Prepare the statement
- Bind a Buffer.
- Execute the Prepared Statement
- Commit the Transaction.

Prepare Statement

For example, the following statement can be prepared and repeatedly executed to update rows in the `emp` table.

```
ret = SQLPrepare(stmt, (unsigned char*)"update emp  
set eid=?, salary=? where eid = ?;", SQL_NTS);
```

Bind the parameters

You have to bind the parameters with the help of `SQLBindParameter()` function.

```
ret= SQLBindParameter(stmt, 1, SQL_PARAM_INPUT,  
SQL_C_SHORT, SQL_INTEGER, 0, 0, &eid1, 0, NULL);
```

```
ret= SQLBindParameter(stmt, 2, SQL_PARAM_INPUT,  
SQL_C_FLOAT, SQL_REAL, 0, 0, &salary1, 0, NULL);
```

```
ret = SQLBindParameter(stmt, 3, SQL_PARAM_INPUT,  
SQL_C_SHORT, SQL_INTEGER, 0, 0, &eid2, 0, NULL);
```

Execute the statement

```
ret = SQLExecute(stmt);
```

SQLExecute() executes a prepared statement, using the current values of the parameter marker variables if any parameter markers exist in the statement.

Commit the transaction

```
ret = SQLTransact(env, dbc, SQL_COMMIT);
```

5.6.7 Delete Records

Refer to the ODBCman6.c for the source code.

The stages for Delete:

- Prepare the statement
- Bind a Buffer.
- Execute the Prepared Statement
- Commit the Transaction.

Prepare statement

```
ret = SQLPrepare(stmt, (unsigned char*)  
"delete from emp where eid = ?;", SQL_NTS);
```

Bind the Parameters

```
ret = SQLBindParameter(stmt, 1, SQL_PARAM_INPUT,  
SQL_C_SHORT, SQL_INTEGER, 0, 0, &eid1, 0, NULL);
```

Execute the statement

```
ret = SQLExecute(stmt);
```

After binding the value you execute the statements using the current value of the parameter marker.

Commit the transaction

Committing permanently deletes the records from the table.

```
ret = SQLTransact(env, dbc, SQL_COMMIT);
```

5.6.8 Drop the Table

Refer to the ODBCman6.c for the source code.

The SQL Statement `drop table emp;` will drop the mentioned table. As this is a one time activity, we might use the `SQLExecDirect()` function (it is the same for all DDL statements).

```
ret = SQLExecDirect(stmt, (unsigned
char*)"drop table emp;", SQL_NTS);
```

5.6.9 Freeing Handles and Disconnect from the CSQL

There is the `SQLFreeHandle()` to free up a handle and its associated resources.

Note that handles generally need to be freed in the opposite order to which they were allocated and that handles cannot be freed if they are in use.

In section 5.5.1, we discussed about the each and every parameters of `SQLFreeHandle()`

```
ret = SQLFreeHandle(SQL_HANDLE_STMT, stmt);

ret = SQLDisconnect dbc);

ret = SQLFreeHandle(SQL_HANDLE_DBC, dbc);

ret = SQLFreeHandle(SQL_HANDLE_ENV, env);
```

6. SQL API

This section explains various interfaces in SQL API and their references along the way, which would help in writing applications to access the CSQL database.

Let us start with an example where you create a table in the CSQL database to store employee details, say

```
table EMP (int empId, char name(20), float sal).
```

The following sections describe how to connect to the database, how to create a table, how to insert, update and delete tuples in the table and how to drop the table.

6.1 Connect to the Database

Refer `manSQLAPIinsert.c` for the source to understand how the SQL API works.

First of all you need to create the connection with the database which is done by the following classes – `SqlFactory`, `AbsSqlConnection`

```
AbsSqlConnection *con = SqlFactory::createConnection(CSql);  
rv = con->connect("root", "manager");
```

The first one declares an `SqlConnection` object that helps to connect with the CSQL database as is evident from the argument `CSql`.

The second class actually connects the user to the database using the `createConnection()` method.

6.2 Create and Set the statement for the connection

At this stage the `SqlStatement` object is created and set to the `SqlConnection` object as follows.

```
AbsSqlStatement *stmt =  
SqlFactory::createStatement(CSql);  
  
stmt->setConnection(con);
```

The first instruction declares the `SqlStatement` object that helps in preparing and executing the SQL statements to carry out the DDL and DML operations.

The second one sets the `SqlConnection` to let the `SqlStatement` talk to the database.

6.3 Create the table

Now you should prepare the sql statement to create the table into the CSQL database.

6.3.1 Prepare Statement

The statement is prepared by the following function.

```
rv = stmt->prepare(statement);
```

Where `statement` is a memory location pointing to the string holding the SQL statement that needs to be prepared for execution.

Make sure that you have the SQL statement to prepare and execute before calling the above function. Preparation of the statement, is done by allocating memory for `statement` and copying the create table statement into the memory.

```
char statement[200];
```

```
strcpy(statement, "CREATE TABLE EMP(EID INT,ENAME  
CHAR(20), SALARY FLOAT);");
```

Prepare sets all that need to be done to execute the statement, internally. In this scenario it will set the type of statement, which is Create Table, sets the table name, name of fields, and their types and size based on the data types specified in the statement.

6.3.2 Execute the Statement and release the memory

Once preparation is done its time to execute the statement. This is done as follows –

```
stmt->execute(rows);
```

It executes the SQL statement, internally and does the required job.

You can release the memory by calling

```
stmt->free();
```

This releases all the memory that was allocated internally for the `SqlStatement` object. This must be done, otherwise it might lead to memory leaks for long running processes.

6.4 Insert tuples into the table

Now let us insert some tuples into the table EMP.

6.4.1 Prepare the statement

You first copy the INSERT SQL statement into a memory location and pass the address of that memory location to the prepare function –

```
strcpy(statement, "INSERT INTO EMP  
VALUES (?, ?, ?);");
```

```
...
```

```
rv = stmt->prepare(statement);
```

If you notice the statement above, the values are given as three '?'s separated by commas. This is called “**parameterizing**” the fields of the table in the order mentioned during the definition of the table.

6.4.2 Start the transaction

For any operation to take place in the table like `INSERT`, `UPDATE`, `DELETE` or `SELECT`, a transaction needs to be started. This is done as below –

```
con->beginTrans();
```

6.4.3 Parameterize the fields

You need to tell the `SqlStatement` object to pick the values from an allocated area of memory and you need to parameterize all the three fields in the table as per our requirement in the example. It is done as show below –

```
stmt->setIntParam(1, eid);  
stmt->setStringParam(2, ename);  
stmt->setFloatParam(3, salary);
```

All three are functionally similar except for the fact that is for varying types of data. The 1st argument is the position of the '?' in the `INSERT` statement starting from 1 and the 2nd argument is the memory location from where the value for the field is picked up during execution

In our example, the 1st parameter to be inserted is `integer`, 2nd one is `String` and 3rd one is `float` type, hence the function calls are in that order.

Suppose if the statement were to be

```
“insert into EMP values(?, 'Kishor', 123.0);”
```

There is only one field to be parameterized and that should have been the `integer` type with position number 1.

If the statement were to be

```
“insert into EMP values(eid, ?, ?);”
```

There are two fields to be parameterized and that should be `String` and `Float` type with position numbers 1 and 2 respectively.

6.4.4 Execute the insert statement

The inserting of a row is done one row at a time by the following function.

```
stmt->execute(rows);
```

This function receives an argument `rows`, which is a reference variable populated by the `execute` function. This is always 1 if successful. Since this is an insert statement and always one row is inserted at a time.

In the example you have called this function in a loop so as to insert 10 rows.

6.4.5 Commit the transaction

After inserting 10 rows you can commit the transaction by using

```
con->commit();
```

The release the memory by calling

```
stmt->free();
```

This releases all the memory that was allocated internally for the `SqlStatement` object. As already mentioned, if this is not done, it might lead to memory leaks.

The first 3 rows in the table EMP look like this logically when you compile and run `manSQLAPIinsert.c`

Table: EMP

empId	name	salary
1001	Praba	1000.00
1002	Kishor	2000.00
1003	Jiten	3000.00

6.5 Read the tuples (rows) from the table

6.5.1 Read tuples from the table.

Now let us read all the tuples that were last inserted into the table. For that the statement `SELECT * FROM EMP;` needs to be prepared.

6.5.2 Prepare the statement

```
strcpy(statement, " SELECT * FROM EMP;");  
rv = stmt->prepare(statement);
```

6.5.3 Bind the fields

```
stmt->bindField(1, &eid);  
stmt->bindField(2, &ename);  
stmt->bindField(3, &salary);
```

Since the query projects all fields using `*`, the entire field values are to be fetched from the table according to the sequence they are in the table, the first parameter

being the column position in the tuple and second parameter being the memory location to store the fetched values.

Supposing the statement was to look like `select EID, ENAME from EMP;` Then the binding would take place for only two parameters EID and ENAME with param position 1 and 2 respectively.

```
stmt->bindField(1, &eid);  
stmt->bindField(2, ename);
```

Supposing the statement was to look like `select SALARY, ENAME from EMP;` Then the binding should take place for only two parameters SALARY and ENAME with param position 1 and 2 respectively.

```
stmt->bindField(1, &salary);  
stmt->bindField(2, ename);
```

6.5.4 Begin transaction and execute the statement

Any DML operation needs to start a transaction and the following statement does it

```
con->beginTrans();
```

Now execute the statement by calling the following function.

```
stmt->execute(rows);
```

This will set the condition for fetching the required tuples and initialize the appropriate iterations for picking up the tuples.

6.5.5 Fetch the tuples

Calling the `fetch()` function will return the address of each tuple in the database. Fetch will also copy the values from the database to the bound locations.

```
stmt->fetch();
```

Fetch will return one tuple at a time that satisfies the condition set in the statement. Here you have selected all the tuples hence it should show all the tuples present in the table. After fetching all the tuples commit the connection using

```
conn->commit();
```

6.6 Update some tuples

Now let us update some of the tuples in table EMP.

Refer `manSQLAPIupdate.c` which updates the tuples where EID = 1001, 1003, 1005. To update the tuples you need to repeat the Section 6.1 and 6.2.

Prepare the SQL statement – “Update EMP SET SALARY, ENAME where EID = ?;

This statement is prepared using

```
stmt->prepare(statement);
```

Begin the transaction by calling

```
con->beginTrans();
```

Now call

```
stmt->setIntParam(1, eid);  
stmt->setStringParam(2, ename);  
stmt->setFloatParam(3, salary);
```

Respectively corresponding to the order of their appearance.

Now call

```
stmt->execute(rows);
```

To execute the prepared statement

Now commit the transaction.

```
conn->commit();
```

Check for the updated values (as per section 6.5) by creating another select statement after freeing the current one.

6.7 Delete tuples

Refer `manSQLAPIdelete.c` which deletes the tuples where EID > 1006.

To delete the tuples you need to repeat the Section 6.1 and 6.2.

Prepare the statement as explained in Section 6.5.1. The statement for this one would be `-Delete from EMP WHERE EID > 1006;`

This statement is prepared using

```
stmt->prepare(statement);
```

The transaction would have been started using

```
con->beginTrans();
```

Now call

```
stmt->execute(rows);
```

Finally commit the transaction

```
conn->commit();
```

All the required fields are deleted based on the condition.

Check for the deleted values by creating another select statement after freeing that statement.

7. DB API

This section explains various interfaces in DB API and their references along the way, which would help in writing applications to access CSQL database directly through the storage engine.

Let us start with an example where you create a table in the CSQL database to store employee details, say `table EMP (int empId, char name(20), float sal)`.

The following sections describe how to connect to the database, how to create a table, how to insert, update and delete tuples in the table and how to drop the table.

7.1 Connect to the database

Refer to `manDBAPIinsert.c`

Firstly you need to create a Connection object and open the connection as follows.

```
Connection conn;  
rv = conn.open("root", "manager");
```

This will connect the application to the database. The return value `rv` is defined as an enumerator. The parameters “root” and “manager” are the username and password respectively. The return value of OK defines success and the rest are errors. Refer `ErrorType.h` for list of error codes.

7.2 Database Manager creates the table

To create the table you need the `DatabaseManager`, which is required to create, open, close and drop the table.

7.2.1 Get the DatabaseManager

```
DatabaseManager *dbMgr =  
conn.getDatabaseManager();
```

7.2.2 Define the table

Before creating the table you need to define the table and to do that you need to create a `TableDef` object. The `TableDef` object will have all the information that a table should have.

Call the `addField` function as many times as there are fields in the table with appropriate parameters as follows.

```
TableDef tabDef;  
tabDef.addField("empId", typeInt, 0, NULL, true);  
tabDef.addField("name", typeString, 20);  
tabDef.addField("salary", typeFloat);
```

Here `addField` is the overloaded function with the parameters being name, type, length, default value and `notNull`.

By default

length is 0 for known type of fields like integer, float etc.

default value is NULL.

`notNull` is false.

Here `empId` field is the field with primary key hence `notNull` parameter is true.

7.2.3 Create table

Now the table can be created with a name as follows

```
rv = dbMgr->createTable("EMP", tabDef);
```

EMP is the table name and since `tabDef` object has all the field information, passing the object as reference to the `dbMgr` will be sufficient to create the table.

7.2.4 Create index for primary key field

Primary key is created whenever a field is unique, which means that the field cannot have duplicate values. In our example `empId` will be the field with primary key value, for obvious reasons. Primary key is created using following lines of code.

```
HashIndexInitInfo *idxInfo = new  
HashIndexInitInfo();  
strcpy(idxInfo->tableName, "EMP");  
idxInfo->list.append("empId");  
idxInfo->isUnique = true;  
idxInfo->isPrimary = true;  
idxInfo->indType = hashIndex;
```

`HashIndex` is used for point look-ups for conditions like `empId = 1005` etc on primary key fields during retrieval of records.

The object is initialized with rest of the details and the `DatabaseManager` finally creates the index with the following function.

```
rv = dbMgr->createIndex("indx1", idxInfo);
```

7.3 Insert tuples into the table

The table has been created in the last section. Let us insert some tuples into the table.

7.3.1 Open the table

To insert tuples into the table you need to first open the table and store the table handle, which is done as follows

```
Table *table = dbMgr->openTable("EMP");
```

The `Database Manager` opens the table and the table name `EMP` is the parameter to it. It returns the table handle, which is stored in `table` variable.

7.3.2 Bind each field of the table

The table in the database needs to know from where to fetch the data for each field in the table.

```
table->bindFld("empId", &id1);  
table->bindFld("name", name);  
table->bindFld("salary", &sal);
```

bindFld is called by the table handle and it will give the address of the memory location from where the data can be fetched for each field in the table.

7.3.3 Start the transaction

For any operation to take place in the table like inserting, selecting, updating and deleting, a transaction needs to be started.

```
conn.startTransaction();
```

The above function is called using Connection object. To call this function a connection should be open. One connection can have only one transaction. To start any other transaction for this connection the current transaction should either be committed or rolled back.

7.3.4 Insert the tuples

Now as the connection is started you can insert some records in the table.

```
rv = table->insertTuple();
```

Before calling this function, make sure that you have the desired values stored in the fields that have been bound by bindFld function.

7.3.5 Commit the transaction

After inserting all the rows, commit the transaction by calling the following function.

```
conn.commit();
```

Remember any modification operation to the table has to be done between startTransaction and the commit / rollback functions of Connection class.

Now all the rows are inserted into the 'EMP' table and, it would look like

Table: EMP

empId	name	salary
-------	------	--------

1001	Praba	1000.00
1002	Kishor	2000.00
1003	Jiten	3000.00

7.4 Read the tuples from the database

Now let us read the tuples that you have inserted in the last section.

7.4.1 Set and execute condition to read all the inserted tuples

To read the tuples you need to set the condition to get all the tuples. This is done by the following function.

```
table->setCondition(NULL);  
rv = table->execute();
```

The `setCondition` function is called with the address of `Condition` object. Here the function is called with `NULL`, which means you set no condition and the table should return all the tuples. You will see how to set condition in a later section. The `execute` function sets the information about how to go about fetching the tuples that satisfy the given condition.

7.4.2 Fetch the tuples

Now the table is all ready to fetch the tuples. It is done by the following function.

```
tuple = (char*)table->fetch();
```

The `fetch` function traverses through each of the tuples in the table and returns the address of the first tuple that satisfies the condition.

It will fill the values of each field in the memory area bound earlier by `bindFld` function. You can rebind those fields with some other memory location if you wish by calling `bindFld` again.

Make sure you start the transaction to read the tuples and ultimately commit or rollback to complete the transaction.

7.5 Update some of the tuples

Now let us update some of the tuples in table EMP. Refer `manDBAPIupdate.c`. As usual to update the tuples, you need to open the table with the help of Database Manager and the Table handle.

7.5.1 Set a condition to update the tuples

Create a `Condition` object.

Call `setTerm` on that object to prepare the condition.

```
Condition p1;  
int val1 = 1006;  
p1.setTerm("empId", OpLessThan, &val1);
```

`setTerm` is an overloaded function which has four types.

```
void setTerm(const char* f1, ComparisionOp op,  
const char *f2);
```

To handle comparisons between two fields such as `f1 & f2`, etc.

```
void setTerm(const char* f1, ComparisionOp op, void  
*opnd);
```

To handle comparisons where operand `opnd` is a constant such as `f1 =5`, etc.

```
void setTerm(Predicate *p1, LogicalOp op,  
Predicate *p2 = NULL);
```

To handle conditions where two or more logical comparisons like `f1 < 5` and `f2 > 7`, etc

In our example, the second type is called by the `Condition` object on `empId` field.

The second and third parameters are the type of comparison and the operand used for that comparison. Here you used `OpLessThan` and `val1` as the parameters for the condition `empId < 1006`.

Now `setCondition` is called by table object with the `Condition` object as the parameter. (`Table.h`).

Now the condition for Update is set.

7.5.2 Start transaction and be prepared to update

Since you are doing an operation on the table you need to start a transaction first by calling

```
conn.startTransaction();
```

Then you call

```
table->execute()
```

as you did in last section to set up the table for scanning.

```
conn.startTransaction();  
rv=table->execute();
```

7.5.3 fetch and update the tuples

Now you fetch tuples that satisfy the above set condition by calling `fetch`.

```
tuple = (char*)table->fetch();
```

Remember `fetch` will populate the values of each field in the tuple with the memory location that was bound earlier by calling `bindFld`.

The tuples are now fetched for the condition `empId < 1006`
Now you can easily update the values as needed by the application as depicted by the following code by updating the values in the bound memory.

```
if (id1 == 1001) {  
    strcpy(name, "Shubha");  
    sal = 1111.00;  
    table->updateTuple();  
}
```

Now the tuples are updated where `id1 = 1001, 1003 and 1005`.

Once after updation transaction should be committed by calling `conn.commit()`. In the source the values are read again as explained in Section 7.4 to test for the updation.

7.6 Delete some of the tuples

Refer `manDBAPIdelete.c` for the source.

Deleting tuples is exactly similar to the way you update the tuples. You call

```
table->deleteTuple()
```

Finally the table is closed and it is deleted from the database by calling

```
dbMgr->dropTable("EMP");
```

8. Archive and Restore

All the statements executed by all the users are stored in the file specified by the `DATABASE_FILE` parameter in the `csql.conf` file.

This file is used to recover the records automatically when the server is shutdown normally and started next time. This automatic recovery works only in case of graceful shutdown. Pressing CTRL-C on the terminal where csqserver is running or sending SIGINT to csqserver process will shut down the server gracefully.

If there is abrupt power failure or disk failure or process failure, CSQL will not be able to recover the data when it is started next time. To cope up with these kinds of failures, CSQL supports archive/restore mechanism through which you shall recover the database up the recent archive.

This mechanism requires the user to take backup of the csq system periodically using the tool csqdump and store in disk file, In case of any failure mentioned above the archive file shall be used to restore the data in the database.

The interval at which the archive needs to be taken is decided based on the application requirement for durability.

Archive the database

```
$csqdump >/tmp/archive1
```

Restore the database

When csqserver is started it checks the file pointed by DATABASE_FILE parameter in csq.conf and loads it during the recovery process. You shall either delete that file and start the server and run the following command to restore the database from the archive

```
$csql -s /tmp/archive1
```

Alternatively you can replace the file pointed by DATABASE_FILE parameter in csq.conf with /tmp/archive1 file and then restart the csqserver to recover with the archived data.

9. CSQL Cache

In keeping with its design goals, CSQL MMDB shall also act as middle-tier data cache for any disk resident database. In this section we outline and demonstrate how CSQL can be configured to work as data cache for MySQL database.

CSQL Cache is a high performance, bi-directional updateable data-caching infrastructure that sits between the clustered application process and back-end data sources to provide unprecedented high throughput to your application by offloading the computing cycles from expensive backend systems along with reduction in costly network calls, thereby enabling real time application to provide faster and predictive response time.

CSQL Cache uses the fastest Main Memory Database (CSQL MMDB) designed for high performance and high volume data computing for caching the table and provides most flexible and cost-effective way to cache and manage enterprise information without compromising on transactional and indexed access to the data. This main memory

database is 10-20 times faster than traditional disk based database system as the database completely resides in main memory and developed to be used on real time high computing data platforms.

9.1 CSQL Cache Functionalities

9.1.1 Updateable Cache Tables

Most of the existing cache solutions are read only which limits their usage to small segment of the applications, non-real time applications.

9.1.2 Bi-Directional Updates

For updateable caches, updates, which happen in cache, should be propagated to the target database and any updates that happen directly on the target database should come to cache automatically.

9.1.3 Synchronous and Asynchronous update propagation

The updates on cache table shall be propagated to target database in two modes. Synchronous mode makes sure that after the database operation completes the updates are applied at the target database as well. In case of Asynchronous mode the updates are delayed to the target database.

Synchronous mode gives high cache consistency and is suited for real time applications. Asynchronous mode gives high throughput and is suited for near real time applications.

9.1.4 Multiple cache granularity

CSQL supports Table level and Result-set caching. Major portions of corporate databases are historical and infrequently accessed. But, there is some information that should be instantly accessible like premium customer's data, etc

9.1.5 Recovery for cached tables

In case of system or power failure, during the restart of caching platform all the committed transactions on the cached tables should be recovered.

9.1.6 Tools to validate the coherence of cache

In case of asynchronous mode of update propagation, cache at different cache nodes and target database may diverge. This needs to be resolved manually and the caching solution should provide tools to identify the mismatches and take corrective measures if required.

9.1.7 Horizontally Scalable

Clustering is employed in many solutions to increase the availability and to achieve load balancing. Caching platform should work in a clustered environment spanning to multiple nodes thereby keeping the cached data coherent across nodes.

9.1.8 Transparent access to non-cached tables reside in target database

Database Cache should keep track of queries and should be able to intelligently route to the database cache or to the origin database based on the data locality without any application code modification.

9.1.9 Transparent Fail over

There should not be any service outages, in case of caching platform failure. Client connections should be routed to the target database.

No or very minimal changes to application for the caching solution.

Support for standard interfaces JDBC, ODBC etc that will make the application to work seamlessly without any application code changes. It should route all stored procedure calls to target database so that they don't need to be migrated.

Caching happens at table level granularity in CSQL, which means that the tables, which are frequently accessed by the applications, shall be specified in CSQL configuration file (`csqtable.conf`), which will be loaded into CSQL during the `csqlserver` startup. Once these tables are loaded, they are treated like normal CSQL tables and any DML operation is allowed on these tables.

The difference between normal CSQL table and cached table is that, the DML operations `INSERT`, `UPDATE`, `DELETE` on cached tables are propagated to MySQL automatically by the CSQL server. For the application, they see only CSQL tables and CSQL takes care of propagating the updates internally to MySQL table.

Apart from caching the MySQL tables, CSQL also acts as a transparent gateway, which allows the application to access the tables, which are not cached in CSQL. This provides a unified interface to the applications and it does not need to deal with the location of the table, making the underlying database caching architecture transparent to the application layer.

9.2 CSQL Configuration

Configuration file, `csql.conf` has five parameters associated with caching. They can be found in the Cache section of the `csql.conf` file. They are `CACHE_TABLE`, `DSN`, `ENABLE_BIDIRECTIONAL_CACHE`, `CACHE_RECEIVER_WAIT_SECS` and `TABLE_CONFIG_FILE`.

- `CACHE_TABLE` is a boolean parameter which needs to be set to true when `csql` should cache tables from target database
- `DSN` is a string parameter, which needs to be set to the data source name of the target ODBC driver specified in the `odbc.ini` file.
- `TABLE_CONFIG_FILE` is a string parameter which contains the complete path to the file which holds the cache table information
- `ENABLE_BIDIRECTIONAL_CACHE` is a boolean parameter which needs to be set to true when direct updates to target database needs to be brought into CSQL cache table automatically.
- `CACHE_RECEIVER_WAIT_SECS` is an integer parameter, which needs to be set to interval it waits if there are no update logs from the target database.

If CSQL needs to cache table `t1` and `t2`, then entries for `t1` and `t2` needs to be present in `csqtable.conf` file. Add the following lines to `TABLE_CONFIG_FILE` that is

```
1:t1
1:t2
```

A sample configuration file is present in the `csql` root directory. If you want to cache tables at run time, that is when `csqlserver` is running, then `cachetable` tool shall be used for that. Refer tool reference section for the syntax and usage.

The first field denotes the mode and it should be specified as always 1, which represents the update propagation mode for cached tables.

- 1-> Synchronous mode (updates are propagated during the DML operation itself)
- 2-> Asynchronous mode (logs are generated and propagated later for DML operations on cached tables).

This release currently supports only the synchronous mode of update propagation.

9.3 Unidirectional Cache Configuration Settings

9.3.1 MySQL Configuration

For CSQCache to work you need to install the MySQL server, MySQL ODBC Connector, unixODBC packages on your system. Please make sure that these packages are installed in your system before you proceed.

Once you install `mysqlserver`, start the server by using the following command after logging in with user `'mysql'`

```
$/etc/init.d/mysqld start
```

After that you have to install MySQL ODBC connector, which contains the ODBC driver to connect to MySQL server. This shall be downloaded from the MySQL web site. Usually this library is named as `libmyodbc3.so`.

After this you need to install unixODBC package, which is the driver manager for ODBC drivers. Once you install unixODBC, copy the following lines into `~/odbc.ini` file

```
[ODBC Data Sources]
myodbc3          = MyODBC 3.51 Driver DSN

[myodbc3]
Driver           = /home/csql/mysql-connector-odbc-
3.51.23-linux-x86-32bit/lib/libmyodbc3.so
Description     = Connector/ODBC 3.51 Driver DSN
SERVER          = localhost
PORT           = 3306
USER            = root
Password        =
Database        = test
```

OPTION = 16
SOCKET = /var/lib/mysql/mysql.sock

The above assumes that MySQL ODBC connector is installed at location
/home/csql/mysql-connector-odbc-3.51.23-linux-x86-32bit

You can check whether you have configured MySQL ODBC driver correctly using the `isql` command

```
$isql myodbc3
It should display the following
+-----+
| Connected! |
| |
| sql-statement |
| help [tablename] |
| quit |
| |
+-----+
SQL>
```

If you get the above output, it means that you have configured the ODBC driver manager and MySQL ODBC connector properly and it can connect to MySQL server.

Create the tables in MySQL server so that they shall be cached in CSQL.

In the SQL prompt enter the following statements

```
SQL>CREATE TABLE t1 (f1 integer, f2 char (196),
primary key (f1));
SQL>CREATE TABLE t2 (f1 integer, f2 integer, primary
key (f1));
SQL>CREATE TABLE t3 (f1 integer, f2 integer, primary
key (f1));
SQL>INSERT INTO t1 (f1, f2) values (100, '100');
SQL>INSERT INTO t2 (f1, f2) values (102, 102);
SQL>INSERT INTO t3 (f1, f2) values (103, 103);
SQL>quit;
```

The above statements create tables namely t1, t2 and t3, which will be cached in CSQL later.

Modify the `csql.conf` file and set the DSN parameter to `myodbc3`

9.3.2 Postgres Configuration

Make sure that postgres and unixODBC is already installed in your system before your proceed.

Once you install postgres, start the server by using the following command after logging in with user 'postgres'

The following commands assume that postgres is installed under /usr/local/pgsql directory.

```
$export PGDATA=/usr/local/pgsql/data
```

```
$/usr/local/pgsql/bin/postmaster \  
>/tmp/postgreslog 2>postgreslog &
```

Copy the following lines into /etc/odbcinst.ini

```
[ODBC]  
Trace = yes  
TraceFile = /tmp/sql.log  
  
[postgres]  
Driver = /usr/lib/libodbcpsql.so  
Setup = /usr/lib/libodbcpsqlS.so  
Description = Connector Driver DSN  
SERVER = localhost  
Database = test  
ReadOnly = No
```

Copy the following lines into /etc/odbc.ini

```
[psql]  
Description = CP resources db  
Driver = PostgreSQL  
Trace = No  
TraceFile = sql.log  
Database = test  
Servername = localhost  
UserName = postgres  
Password =  
Port = 5432  
Protocol = 6.4  
ReadOnly = No  
RowVersioning = No  
ShowSystemTables = No  
ShowOidColumn = No  
FakeOidIndex = No  
ConnSettings =
```


***ExtraSysTablePrefixes* =**

Note: You need to have root privileges to modify /etc/odbc.ini and /etc/odbcinst.ini file.

You can check whether you have configured postgres ODBC driver correctly using the `isql` command

```
$isql psql
It should display the following
+-----+
| Connected! |
|          |
| sql-statement |
| help [tablename] |
| quit      |
|          |
+-----+
SQL>
```

If you get the above output, it means that you have configured the ODBC driver manager properly and it can connect to postgres server.

Create the tables in postgres server so that they shall be cached in CSQL.

In the SQL prompt enter the following statements

```
SQL>CREATE TABLE t1 (f1 integer, f2 char (196),
primary key (f1));
SQL>CREATE TABLE t2 (f1 integer, f2 integer, primary
key (f1));
SQL>CREATE TABLE t3 (f1 integer, f2 integer, primary
key (f1));
SQL>INSERT INTO t1 (f1, f2) values (100, '100');
SQL>INSERT INTO t2 (f1, f2) values (102, 102);
SQL>INSERT INTO t3 (f1, f2) values (103, 103);
SQL>quit;
```

The above statements create tables namely t1, t2 and t3, which will be cached in CSQL later.

Modify the `csql.conf` file and set the DSN parameter to `psql`

9.4 Starting the csqlserver

`csqlserver` supports `-c` option which loads the cached tables into CSQL during the startup. Before you run make sure that filename pointed to by `csql.conf` parameter `TABLE_CONFIG_FILE` contains the table names which needs to be loaded into CSQL.

Run the below command to cache the tables in CSQL server

```
$ csqserver -c
```

9.5 Working with CSQL gateway

In another terminal run csq tool with -g option. This creates an isql session which acts as gateway to csq and target database.

```
$ csq -g
```

It will show the CSQL prompt as follows

```
CSQL>
```

To retrieve records from table t1, enter the following statement in CSQL prompt.

```
CSQL>select * from t1;
```

It will display

```
-----
      f1      f2
-----
      100     100
-----
```

It displays the values inserted into target database (Mysql or Postgres) from CSQL.

You can also perform any DML operations on these cached tables, for example

```
CSQL>insert into t1 values (200, '200');
```

This will insert one record in CSQL as well as in target database. You can verify this by accessing target database through isql tool by specifying its DSN name (myodbc3 for MySQL and psql for Postgres)

```
$isql myodbc3;
SQL>select * from t1;
+-----+-----+
| f1      | f2      |
|         |         |
+-----+-----+
```

```

-----+
| 100          | 100
|
| 200          | 200
|
+-----+-----
-----
-----
-----+

```

It displays both the records from target database.

You can also retrieve records in tables, which are not cached in CSQL and are present in target database

```

$ csql -g
CSQL>select * from t3;

```

It will display

```

24083:3086153424:DatabaseManagerImpl.cxx:599:Table not
exists t3
24083:3086153424:SelStatement.cxx:245:Unable to open
the table:Table not exists
-----
---
          f1          f2
-----
---
          103          103

```

First it displays that the table is not present in CSQL, then it checks with target database whether the table is present there and if present it retrieves the records from target database.

9.6 Programming with CSQL gateway

You can find a JDBC sample program `gwexample.java` under the `examples/jdbc` directory. It will demonstrate how to use the CSQL gateway through Java programs.

Note: You shall also use gateway through JDBC and SQLAPI interfaces. The ODBC interface does not support gateway in this release.

From SQL API, use `CSqlGateway` to create connection and statement objects in `SqlFactory` class like below:

```
AbsSqlConnection *con = SqlFactory::
    createConnection(CSqlGateway);

rv = con->connect("root", "manager");

if (rv != OK) return 1;
AbsSqlStatement *stmt = SqlFactory::
    createStatement(CSqlGateway);

stmt->setConnection(con);

//---other statements follow---
```

9.7 Configuring Bi-Directional Cache

The default caching in CSQL is unidirectional caching, which means all updates (INSERT, UPDATE, DELETE) on cached tables will be automatically propagated to target database. CSQL also supports bi-directional caching in which, direct updates on target database are propagated to CSQL cache automatically.

Bi-directional caching is implemented using triggers of the target database. This requires additional tables and triggers, which needs to be installed on the tables in target database, which needs to be cached in bi-directional mode.

Sample trigger code is available in the file `trigger.sql` under the CSQL root directory.

9.7.1 MySQL Configuration

First create the table in MySQL to hold the log records using the SQL statement below using `mysql` tool or `isql` tool.

```
CREATE TABLE csql_log_int (tablename CHAR(64), pkid INT, operation INT, id INT NOT NULL UNIQUE AUTO_INCREMENT) engine='innodb';
```

Lets say for cached table 'p1' having primary key field 'f1', following is the format for creating the triggers in MySQL database. A sample file (`trigger.sql`) is available at the CSQL root directory; you shall modify that with your cached table name and its primary key field.

```
use test;
drop trigger if exists triggerinsertp1;
drop trigger if exists triggerupdatep1;
drop trigger if exists triggerdeletep1;
```

```

DELIMITER |
create trigger triggerinsertp1
AFTER INSERT on p1
FOR EACH ROW
BEGIN
Insert into csqlog_int (tablename, pkid, operation)
values ('p1', NEW.f1, 1);
End;

create trigger triggerupdatep1
AFTER UPDATE on p1
FOR EACH ROW
BEGIN
Insert into csqlog_int (tablename, pkid, operation)
values ('p1', OLD.f1, 2);
Insert into csqlog_int (tablename, pkid, operation)
values ('p1', NEW.f1, 1);
End;
create trigger triggerdeletep1
AFTER DELETE on p1
FOR EACH ROW
BEGIN
Insert into csqlog_int (tablename, pkid, operation)
values ('p1', OLD.f1, 2);
End;
|

```

Note: Trigger name ends with the table name. Replace ‘p1’ in the above script to the cached table name and ‘f1’ to the primary key fieldname of the cached table.

After editing the trigger.sql file as per your need, you shall execute it by

```
$ mysql -u root -p <trigger.sql
```

Apart from the triggers, `ENABLE_BIDIRECTIONAL_CACHE` parameter in `csqlog.conf` should be set to true for bi-directional caching of tables.

9.7.2 Postgres Configuration

First create the table in Postgres to hold the log records using the SQL statements given below using `mysql` tool or `isql` tool.

```
CREATE TABLE csqlog_int(tablename varchar(64), pkid int, operation int);
ALTER TABLE csqlog_int add id serial;
```

Lets say for cached table ‘t1’ having primary key field ‘f1’, following is the format for creating the triggers in postgres database. A sample file (`trigger.psql`) is available at the

CSQL root directory; you shall modify that with your cached table name and its primary key field.

```
CREATE LANGUAGE plpgsql;
CREATE FUNCTION log_insert_t1() RETURNS trigger AS
$triggerinsertt1$
BEGIN
insert into csql_log_int (tablename, pkid, operation)
values ('t1', NEW.f1, 1);
RETURN NEW;
END;
$triggerinsertt1$ LANGUAGE plpgsql;
```

```
create trigger triggerinsertt1
AFTER INSERT on t1
FOR EACH ROW
EXECUTE PROCEDURE log_insert_t1();
```

```
CREATE FUNCTION log_update_t1() RETURNS trigger AS
$triggerupdatet1$
BEGIN
insert into csql_log_int (tablename, pkid, operation)
values ('t1', OLD.f1, 2);
insert into csql_log_int (tablename, pkid, operation)
values ('t1', NEW.f1, 1);
RETURN NEW;
END;
$triggerupdatet1$ LANGUAGE plpgsql;
```

```
create trigger triggerupdatet1
AFTER UPDATE on t1
FOR EACH ROW
EXECUTE PROCEDURE log_update_t1();
```

```
CREATE FUNCTION log_delete_t1() RETURNS trigger AS
$triggerdeletet1$
BEGIN
insert into csql_log_int (tablename, pkid, operation)
values ('t1', OLD.f1, 2);
RETURN NEW;
END;
$triggerdeletet1$ LANGUAGE plpgsql;
```

```
create trigger triggerdeletet1
AFTER DELETE on t1
FOR EACH ROW
```

```
EXECUTE PROCEDURE log_delete_t1();
```

Note: Trigger name ends with the table name. Replace ‘t1’ in the above script to the cached table name and ‘f1’ to the primary key fieldname of the cached table.

After editing the trigger.psql file as per your need, you shall execute it by running the below command under ‘postgres’ login

```
$ psql test -f /tmp/trigger.psql
```

Note: The above command assumes that trigger.psql file is modified at /tmp directory after copying it from the CSQL root directory.

10. Configuration

CSQL subsystem requires some system parameters that need to be set before starting CSQL database. Hence CSQL defines some of the system configuration variables that need to be defined. These configuration variables are defined in a file called `csql.conf`. Some of the parameters mentioned in this file may have to be tweaked based on the requirements.

The lines starting with # are ignored as comments and the rest are treated as configuration variables. All the other lines are read from this file during server start up.

These configuration variables are divided logically into following classes.

- **Server section variables**
- **Client section variables**
- **Cache section variables**

10.1 Server section variables

It is very important to note that for Server section parameters, the value should be the same for the server process and all the CSQL client processes, which connects to it. Otherwise, behavior is undefined.

10.1.1 PAGE_SIZE

Each database is logically divided into pages and allocation happens in this unit of pages. Increasing this value will reduce frequent allocation of pages. This value should be a multiple of 1024 bytes. This value may be set to the OS page size which is usually 8192 bytes.

10.1.2 MAX_PROCS

This is the number of process that can connect and work with the database concurrently. This value can be set anywhere between 10 and 8192 depending on the number of users who may access the database.

10.1.3 MAX_SYS_DB_SIZE

This is the maximum size of the system database where the metadata are stored. The value can be anywhere between 1 MB and 1GB. It should be multiples of PAGE_SIZE.

10.1.4 MAX_DB_SIZE

This is the maximum size of the user database where user data are stored. This value can be set anywhere between 1MB and 2GB. It should be multiples of PAGE_SIZE.

10.1.5 SYS_DB_KEY

Shared memory key to be used by the system to create and locate system database. The value can be anywhere between 10 and 8192.

10.1.6 USER_DB_KEY

Shared memory key to be used by the system to create and locate user database. The value can be anywhere between 10 and 8192. This should not be the same as SYS_DB_KEY.

10.1.7 LOG_FILE

Full path of the directory where the important CSQL system specific log files are created. Make sure that this directory exists before you start the server.

10.1.8 DATABASE_FILE

Full path of the directory where the database records are stored. This data is used during recovery after a graceful shutdown. Make sure that this directory exists before you start the server.

10.1.9 MAP_ADDRESS

This is the virtual memory start address at which the shared memory segment will be created and attached.

10.2 Client section variables

10.2.1 MUTEX_TIMEOUT_SECS

Mutex timeout interval in seconds. When requesting for mutex, if it is acquired by anybody else, then the requester will wait for this specified time interval before it checks whether it is released.

10.2.2 MUTEX_TIMEOUT_USECS

Mutex timeout interval in microseconds. The cumulative of the seconds and microseconds set will be used for the mutex timeout.

10.2.3 MUTEX_TIMEOUT_RETRIES

Number of retries before csq1 gives mutex timeout error.

10.2.4 LOCK_TIMEOUT_SECS

Lock timeout interval in seconds. When requesting for lock, if it is acquired by anybody else, then the requester will wait for this interval before it checks whether it is released.

10.2.5 LOCK_TIMEOUT_USECS

Lock timeout interval in microseconds. The cumulative of the seconds and microseconds set will be used for the lock timeout.

10.2.6 LOCK_TIMEOUT_RETRIES

Number of retries before csq1 gives lock timeout error.

10.3 Cache section variables

10.3.1 CACHE_TABLE

Enables the caching of tables from the target database. Default value is false.

10.3.2 DSN

DSN Name to connect to the target database. This name should be present in the `odbc.ini` file with the respective ODBC library specified in it.

10.3.3 TABLE_CONFIG_FILE

File name where the cached table information is stored. Specify the file name with full path.

10.3.4 ENABLE_BIDIRECTIONAL_CACHE

Enables the bi-directional caching for cached tables. Direct updates to MySQL will be brought into CSQL cache table automatically making cache coherent.

10.3.5 CACHE_RECEIVER_WAIT_SECS

Interval it waits if there is no update logs from the target database for bi-directional caching.

11. Tool reference

CSQL provides certain tools to access data from CSQL database. These are

- `csql`
- `catalog`
- `csqldump`

11.1 `csql`

CSQL provides a tool called `csql`, which is a sub-shell used to access the CSQL database. It supports most of the standard SQL statements.

Type `csql` to run CSQL sub shell. Make sure that `csqlserver` is running prior to running this tool.

```
$ csql
CSQL>
```

Once you have the CSQL prompt the tool is ready to access the database.

11.2 `catalog`

`Catalog` is a tool, which provides the information about system metadata and user metadata of tables stored in the CSQL database.

```
catalog [-u username] [-p password] [-l]
        [-i] [-d] [-T table]
        [-I index] [-D <lock|trans|proc>]
```

Options:

- | | |
|---------------------------------|---|
| <code>-u <u>username</u></code> | username of the user |
| <code>-p <u>password</u></code> | password of the user |
| <code>-l</code> | lists all the tables with field information |
| <code>-i</code> | reinitialize catalog tables dropping all the tables |
| <code>-d</code> | prints the database usage statistics for system and user database |
| <code>-T <u>table</u></code> | prints the table information |

```
-I index           prints the index information  
-D lock | trans | proc prints debug information for system tables
```

If the username is not mentioned then it will list all the tables with only their names.

If multiple options are specified then only the last option is considered for processing.

Let us understand some of the outputs of the command.

You create two tables in the database as follows with the help of CSQL tool.

```
$ csql  
  
CSQL> create table t1(f1 int, f2 char(20), f3  
float);  
Statement Executed  
  
CSQL>create table emp(eid int, name char(20), sal  
float);  
statement Executed  
  
CSQL>quit;
```

We have two tables, t1 and emp created. Let us see how the catalog tool displays the details of the two tables.

```
$ catalog -l  
<TableNames>  
  <TableName> t1 </TableName>  
  <TableName> emp </TableName>  
</TableNames>
```

This is a default behavior as mentioned before since there is no username provided.

```
$ catalog -u root -p manager -l
```

This will list all the tables with field information and index information

```
$ catalog -u root -p manager -d
```

This will print the database usage statistics

```
$ catalog -u root -p manager -T <table-name>
```

This will list Field and Index information of the table specified.

```
$ catalog -u root -p manager -I <index-name>
```

This will list index information of the index specified.

```
$ catalog -u root -p manager -D proc
```

This will list process table information

```
$ catalog -u root -p manager -D lock
```

This will list lock table information

```
$ catalog -u root -p manager -D trans
```

This will list transaction table information

```
$ catalog -u root -p manager -ild
```

This is same as -d option

```
$ catalog -u root -p manager -i
```

This will drop all the tables from the database

11.3 csqldump

csqldump is a tool that generates a standard file readable by the csql tool and dumps on the standard output. This file is fed into the csql tool to build the database that was present at the time of the file generation. This can be viewed as a back up mechanism where one can close down the server by generating this file and rebuild the database next time when the server is started again.

```
csqldump [-u username] [-p password] [-c] [-n  
numberOfStmtPerCommit] [-T tableName]
```

```
$ csqldump -?
```

```
Usage: csqldump [-u username] [-p passwd] [-c] [-n  
noOfStmtsPerCommit]
```

```
n -> number of statements per commit  
Default value is 100. If system database size is  
bigger, then it shall be increased.
```

```
T-> Will dump only the table specified with this  
option.
```

```
c -> includes all the cache tables in the dump output
```

Note: csqldump does not output cache tables by default. Use c option to include cache tables.

Now let us create some tables and insert some of the tuples into those tables.

Run the `csqlserver` in one terminal.

Open another terminal.

Run `csql` tool.

```
$ csql
CSQL> set autocommit off;
AUTOCOMMIT Mode is set to OFF
CSQL> create table t1(f1 int, f2 char(30), primary key(f1));
Statement Executed
CSQL> insert into t1 values(1, 'Lakshya');
Statement Executed: Rows Affected = 1
CSQL> insert into t1 values(10, 'Uttara');
statement Executed: Rows Affected = 1
CSQL> commit;
CSQL> create table emp(empId int, empName char(40), empSal
float, primary key (empId));
Statement Executed
CSQL> insert into emp values(1001, 'Jitendra', 1000.00);
Statement Executed: Rows Affected = 1
CSQL> insert into emp values(1002, 'Dharmendra', 2000.00);
Statement Executed: Rows Affected = 1

CSQL> commit;
CSQL> quit;
```

Two tables are now created in the `csql` database with each table having two tuples.

Now running `csqldump` will dump the file on the standard output.

```
$ csqldump
CREATE TABLE t1 (f1 INT NOT NULL , f2 CHAR (30));
CREATE INDEX t1_idx1_Primary on t1 ( f1 ) UNIQUE;
CREATE TABLE emp (empId INT NOT NULL , empName CHAR (40),
empSal FLOAT );
CREATE INDEX emp_idx1_Primary on emp ( empId ) UNIQUE;
SET AUTOCOMMIT OFF;
INSERT INTO t1 VALUES(1, 'Lakshya');
INSERT INTO t1 VALUES(10, 'Uttara');
COMMIT;
INSERT INTO emp VALUES(1001, 'Jitendra',1000.000000);
INSERT INTO emp VALUES(1002, 'Dharmendra',2000.000000);
COMMIT;
```

Again run `csqldump` and redirect the output to `backup.sql`.

```
$ csqldump > backup.sql
```

Close the server by hitting `Ctrl + C`.

Open the server again. At this point server is in virgin state.
Now run `csql` tool with the file as input as follows.

```
$ csql -u root -p manager -s backup.sql
Statement Executed
Statement Executed
Statement Executed
Statement Executed
AUTOCOMMIT Mode is set to OFF
Statement Executed: Rows Affected = 1
Statement Executed: Rows Affected = 1
Statement Executed: Rows Affected = 1
Statement Executed: Rows Affected = 1
```

The first 4 lines are for creation of table and index for both tables. Then the `autocommit` mode is set OFF by the `csqldump` tool and the last 4 rows are the output resulting from the insertion of 2 tuples each for each table.

In order to check whether the tuples are loaded appropriately or not, do the following

Start the `csql` tool.

```
$ csql
CSQL>show tables;
=====TableNames=====
      t1
      emp
=====
```

```
CSQL>select * from t1;
-----
      f1      f2
-----
      1      Lakshya
      10      Uttara
-----
```

```
CSQL>select * from emp;
-----
      empId   empName  empSal
-----
      1001    Jitendra 1000.000000
      1002    Dharmendra 2000.000000
-----
```

```
CSQL>quit;
```

```
$ csqldump -T t1
CREATE TABLE t1 (f1 INT NOT NULL , f2 CHAR (30));
CREATE INDEX t1_idx1_Primary on t1 ( f1 ) UNIQUE;
SET AUTOCOMMIT OFF;
INSERT INTO t1 VALUES(1, 'Lakshya');
INSERT INTO t1 VALUES(10, 'Uttara');
```

COMMIT;

11.4 cachetable

`cachetable` is a tool to cache the table from the target database into CSQL. This needs to be invoked when the `csqlserver` process is running.

Syntax:

```
cachetable [-U username] [-P password] -t tablename
           [-R] [-s] [-r]
```

`tablename` -> table name to be cached in `csql` from target db.

`R` -> recover all cached tables from the target database.

`s` -> load only the records from target db. Assumes table is already created in `csql`

`r` -> reload the table. get the latest image of table from target db

`u` -> unload the table. if used with `-s` option, removes only records and preserves the schema

For the below command to work, table `t1` should exist in the target database and should not exist in CSQL.

To create the table and insert records in MySQL,

```
$ isql myodbc3
+-----+
| Connected!                                |
|                                            |
| sql-statement                             |
| help [tablename]                          |
| quit                                       |
|                                            |
+-----+
SQL> create table emp (empId int, empName char(40), empSal
float, primary key (empId));
SQLRowCount returns 0
SQL> insert into emp values(1001, 'Jitendra', 1000.00);
SQLRowCount returns 1
SQL> insert into emp values(1002, 'Dharmendra', 2000.00);
SQLRowCount returns 1
SQL> insert into emp values(1003, 'Rajendra', 3000.00);
SQLRowCount returns 1
SQL> insert into emp values(1004, 'Narendra', 4000.00);
SQLRowCount returns 1
```

Now 4 rows are added into the MySQL database, which is our target database.

If you want to cache table `emp`, then run the following command

```
$ cachetable -u root -p manager -t emp
```

After loading you can check whether the records are loaded into CSQL using csq tool

```
$ csq
```

```
CSQL>select * from emp;
```

It will display all the records inserted into MySQL table 'emp'

11.5 cacheverify

cacheverify is a tool that will display the missing records in the specified cached table either in csq or in target database, if any.

This tool should be used when CACHE_TABLE option is set in the csq.conf file.

```
cacheverify [-U username] [-P password] -t tableName [-p] [-f]
```

```
$ cacheverify -?
```

```
Usage: cacheverify [-U username] [-P passwd] -t tablename  
      [-p] [-f]  
      username -> username to connect with csq.  
      password -> password for the above username.  
      tablename -> cached table name in csq from target db.  
      p -> verification at primary key field level  
      f -> verification at record level  
      ? -> help
```

Table name must be specified and it must be a cached table having a primary key field.

By default, that is without -p or -f switch the tool will display only the count of the records in both the databases.

With -p switch, the tool will display only the missing records in either of the database with primary key field value.

The switch -f, will display the missing records in either of the database and the mismatching fields in the records present in both the databases having same primary key field value.

Now let us see how the tool works.

Create a table t1 in target database as follows. We used mysql as the target database at our end.

```
$ isql myodbc3
+-----+
| Connected!
|
| sql-statement
| help [tablename]
| quit
|
+-----+
SQL> create table emp(empId int, empName char(40), empSal
float, primary key (empId));
SQLRowCount returns 0
SQL> insert into emp values(1001, 'Jitendra', 1000.00);
SQLRowCount returns 1
SQL> insert into emp values(1002, 'Dharmendra', 2000.00);
SQLRowCount returns 1
SQL> insert into emp values(1003, 'Rajendra', 3000.00);
SQLRowCount returns 1
SQL> insert into emp values(1004, 'Narendra', 4000.00);
SQLRowCount returns 1
```

Now 4 rows are added into the mysql database which is our target database.

Let us cache this table into csql.

To cache this table into csql add an entry

1:emp

on a new line in csqtable.conf.

Now run

```
$ csqserver -c
ConfigValues
  getPageSize 8192
  getMaxProcs 100
  getMaxSysDbSize 1048576
  getMaxDbSize 10485760
  getSysDbKey 1222
  getUserDbKey 4555
  getLogFile /tmp/log/csql/log.out
  getMapAddress 400000000
  getMutexSecs 0
  ...
  ...
  ...
  getTableConfigFile /tmp/csql/csqtable.conf
  isTwoWayCache 1
  getCacheWaitSecs 10
Sysdb size 1048576 dbsize 10485760
System Database initialized
Database server recovering cached tables...
```

```
Recovering table emp
Cached Tables recovered
Starting Cache Recv Server
filename is /home/csql/install/bin/csqlcacheserver
Cache Recv Server Started pid=6414
Database server started
Cache server started
```

Now the table emp is cached into the csql server.

Let us now check the output of cacheverify for the default option and for the -p option.

Open another terminal and setup the environment by moving into csql root directory and entering the following command

```
$ . ./setupenv.ksh
$ cacheverify -t emp
```

Data	In CSQL	In TargetDb
Number of Tuples	4	4

```
$ cacheverify -t emp -p
Primary key field name is 'empId'
The data in both the servers is consistent
```

Let us delete one row with primary key field value 1002 from target database and one row with primary key value 1004 from csql as follows.

```
$ isql myodbc3
+-----+
| Connected! |
|           |
| sql-statement |
| help [tablename] |
| quit      |
|           |
+-----+
```

```
SQL> delete from emp where empId = 1002;
SQLRowCount returns 1
SQL> quit;
```

```
$ csql
CSQL>delete from emp where empId = 1004;
Statement Executed: Rows Affected = 1
CSQL>quit;
```

Now one row each is deleted from each of the databases. Let us see what happens to the output of cacheverify.

```
$ cacheverify -t emp
```

Data	In CSQL	In TargetDb
Number of Tuples	3	3

```
$ cacheverify -t emp -p
Primary key field name is 'empId'
```

empId	not in csql	not in targetdb
1002		X
1004	X	

The first output shows number of tuples present in both the databases. And the second output shows the missing records based on the primary key field values 1002 and 1004 in the sorted increasing order.

Now let us check how -f switch works. There are two tuples that are present in both the databases. One with primary key value 1001 and another with 1003. Let us update record with primary key value 1001 in csql and update record with primary key value 1003 in target database.

```
$ csql
CSQL>update emp set empName = 'Ganesh' where empId = 1001;
Statement Executed: Rows Affected = 1
CSQL>quit;
$ isql myodbc3
```

```
+-----+
| Connected! |
| |
| sql-statement |
| help [tablename] |
| quit |
| |
+-----+
```

```
SQL> update emp set empName = 'Mahesh' where empId = 1003;
SQLRowCount returns 1
SQL> quit;
```

Check the out put with the -f field now using cacheverify tool. The output will display all the differences in records at every level.

```
$ cacheverify -t emp -f
```

```
Number of Records:
```

```
-----+
```

Data	In CSQL	In TargetDB
No. Of Records	3	3

Primary key field name is 'empId'

Missing Records: Marked by 'X'

Primary Key	In CSQL	In Target DB
1002		X
1004	X	

Inconsistent Records for the same key:

Primary Key	Field Name	In CSQL	In Trgt DB
1001	empName	Ganesh	Jitendra
1003	empName	Rajendra	Mahesh

12. Troubleshooting

12.1 Errors while building CSQL

12.1.1 Please set JDK_HOME

```
$ ./build.ksh
Please set JDK_HOME
```

This error is thrown when `JDK_HOME` is not set. CSQL requires Java Development Toolkit (JDK) Version 1.6 or higher. If it is not present in the system then please install it.

Set up the `JDK_HOME` by following the steps below –

```
$ export JDK_HOME=/home/csql/jdk1.6.0_04
```

In the above example it is assumed that the JDK is based in the `/home/csql` directory, you might change the path based on your system. Now run `./build.ksh` and it should work.

12.1.2 Cannot find -lodbc

```
$ make
...
/usr/lib/gcc/i586-suse-
linux/4.2.1/../../../../i586-suse-linux/bin/ld:
cannot find -lodbc

collect2: ld returned 1 exit status

make[3]: *** [libcsqlobcadapter.la] Error 1
...
...
```

This error is thrown when `unixodbc` module is not installed in the system. Please download and install `unixodbc` rpm package from <http://rpm.pbone.net/index.php3/stat/3/srodzaj/1/search/unixODBC>

To install the rpm

```
$ rpm -ivh <rpm-package-file>
```

Now run `make` and it should work.

12.2 Errors while running `csqlserver`

12.2.1 – `bash: csqlserver Command not found`

```
$ csqlserver
-bash: csqlserver: command not found
```

This error is thrown when the `$PATH` environmental variable may not have been set for `CSQL`. Run the following command from `CSQL_ROOT` directory

```
$ . ./setupenv.ksh
```

Now run `csqlserver` and it should work.

12.2.2 Unable to create the log file

```
$ csqlserver

4822:3086075584:Logger.cxx:101:Unable to create
log file. Check whether server started
Unable to start the logger
```

This error is thrown when `csqldbserver` is not able to create the log file. Please create the directory defined for `LOG_FILE` in `csqldb.conf` file, which is present in `CSQL_ROOT` directory.

For example if `LOG_FILE=/tmp/log/csqldb/log.out` then create the directory as follows.

```
$ mkdir -p /tmp/log/csqldb
```

If you wish to create the log file in different directory then create that directory and change the value of `LOG_FILE` in `csqldb.conf` file.

Now run `csqldbserver` and it should work.

13. Getting Support

The primary mechanism for CSQL communication is through its forums. Anyone who is using this product shall participate in user forum. You can search for the archive of past discussions before you post your question. Post your issues/questions at http://sourceforge.net/forum/forum.php?forum_id=562614

As is usual with forums, be prepared to wait for an answer.

Please summarize any off-list knowledge gained and post it for the benefit of all. For example, if a user asks a question and gets a response privately, post that to user forums.

Any development related queries, should be posted on development forums at http://sourceforge.net/forum/forum.php?forum_id=562615

14. How to contribute

For beginners, items, which come first in the list, are a good starting point. It is ordered on the basis of complexity

- Check out, build the code and run the tests.
- Add functional, stress and scalability Tests.
- Fix bugs.
- Add new test cases for user exposed interfaces
- Develop test cases, run them and create bugs
- Review bug fixes, new feature's design and its code

- Test the Documentation
 - Review the manual and test all the examples. If you find something that looks wrong, create bug and specify "Documentation" as category
- Develop New Features
- Testing New Features
- Read Architecture in csql wiki page and update missing links
- Suggesting new features
- Improve subsystems (code reorganization, performance improvement, etc)

If you find any issues or any queries on CSQL please get back to us at feedback@csqldb.com.

Appendix – A (Benchmark Results)

All times are in microseconds and benchmarking is done against leading open source database. Read operation is point lookup on the primary key field, F1.SQLAPI in the native C++ interface of CSQL for its SQL Engine.

The benchmark is done on most frequently used database operations

- INSERT one record
- SELECT on the primary key field with equality predicate
- UPDATE one field with equality predicate on primary key field
- DELETE with equality predicate on the primary key field

Machine Configuration

Dell OPTIPLEX 320, Intel Pentium D 800 MHz Dual core, 1GBRAM, Linux 2.6 Kernel

Schema Definition

```
CREATE TABLE T1 (F1 INTEGER, F2 CHAR (200), PRIMARY KEY (F1));
```

CSQL MMDB Benchmark Results

For the above said operations, time taken is measured in microsecond for leading traditional database system and for CSQL Main Memory Database System. The benchmarking application and the database server runs in the same machine/host and table fully cached in RAM during the test.

Operation	Other DB ODBC	CSQL ODBC	ODBC Times Faster	CSQL SQLAPI	SQLAPI Times Faster
Insert	139	10	13.90	6.9	20.14
Read =	167	7.5	22.27	3.2	52.19
Update	163	9	18.11	7.9	20.63
Delete	148	11	13.45	8.2	18.05

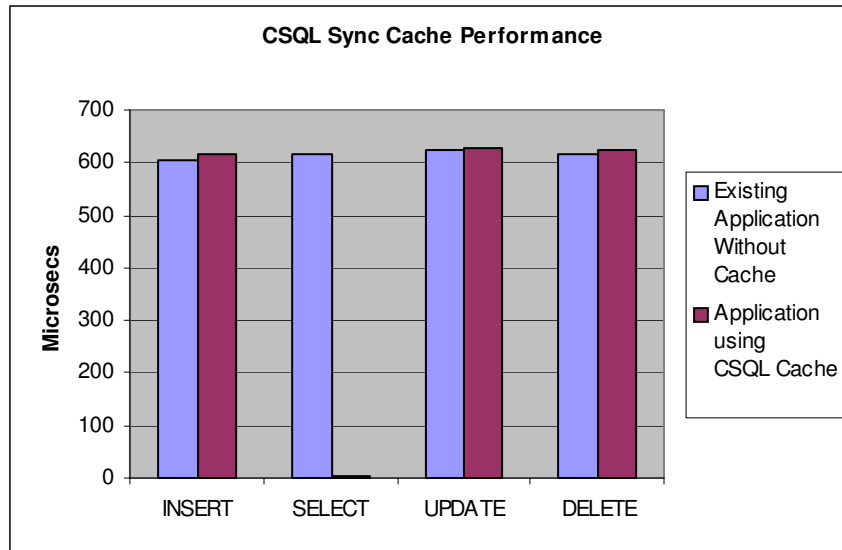
from the above results, it is evident that CSQL is 22 times faster than leading database with standard ODBC interface and 52 times faster with proprietary C++ SQL API. This demonstrates CSQL's ability to meet the most demanding service levels which traditional disk based database systems cannot deliver.

CSQL Cache Benchmark Results

Time taken for the above operations in microsecond granularity for an application, which access the table in the target database directly (Column-2 of the below table) and after it caches the table using CSQL cache (Column-3 of the below table). The fourth column in the table tells how many times performance of the above said operations increase after employing CSQL Cache.

Network: Application host and target db host connected through 10/100Mb Fast Ethernet switch

Database Operation	Existing Application Without Cache	Application using CSQL Cache	Times Faster
INSERT	603	616	0.98
SELECT	618	4	154.50
UPDATE	623	628	0.99
DELETE	617	625	0.99



From the above results, for sync cache update propagation mode, it is evident that CSQL queries are **150** times faster than the existing database system and there is no degradation of performance in case of write operations.