



Programmer's Guide

**CSQL**  
Main Memory Database Cache

# Table of Contents

<b>1. BEFORE YOU START .....</b>	<b>5</b>
1.1. WHAT IS CSQL.....	5
1.2. WHO SHOULD READ THIS BOOK .....	5
1.3. INFORMATION IN THIS GUIDE .....	5
1.4. CONVENTIONS.....	5
1.4.1. TYPOGRAPHIC RULE.....	5
1.4.2. SYNTAX ANNOTATION .....	6
1.5. TECHNICAL SUPPORT .....	6
<b>2. MAKING CLIENT APPLICATION.....</b>	<b>8</b>
2.1. WHAT IS A CLIENT?.....	8
2.2. PASSING QUERY TO THE SERVER .....	8
2.3. SUPPORTED INTERFACES .....	8
<b>3. SQLAPI (PROPRIETARY INTERFACE) .....</b>	<b>9</b>
3.1. WHAT IS SQL API.....	9
3.2. LIBCSQLSQL – C++ LIBRARY .....	9
3.3. IMPORTANT CLASSES AND HEADER FILES.....	9
3.4. ERROR HANDLING ENUM .....	10
3.5. DATA TYPE .....	10
3.6. CONNECT TO THE DATABASE.....	10
3.7. CREATE AND SET THE STATEMENT FOR THE CONNECTION .....	11
3.8. CREATE THE TABLE.....	11
3.8.1. PREPARE STATEMENT .....	12
3.8.2. EXECUTE THE STATEMENT AND RELEASE THE MEMORY .....	12
3.9. INSERT RECORDS INTO THE TABLE .....	12
3.9.1. PREPARE THE STATEMENT.....	12
3.9.2. START THE TRANSACTION .....	13
3.9.3. PARAMETERIZE THE FIELDS .....	13
3.9.4. EXECUTE THE INSERT STATEMENT.....	14
3.9.5. COMMIT THE TRANSACTION .....	14
3.10. READ THE RECORDS FROM THE TABLE.....	14
3.10.1. READ RECORDS FROM THE TABLE. ....	14
3.10.2. PREPARE THE STATEMENT.....	14
3.10.3. BIND THE FIELDS .....	15
3.10.4. BEGIN TRANSACTION AND EXECUTE THE STATEMENT.....	15

3.10.5.	FETCH THE RECORDS.....	15
<b>3.11.</b>	<b>UPDATE SOME RECORDS.....</b>	<b>16</b>
<b>3.12.</b>	<b>DELETE RECORDS.....</b>	<b>16</b>
<b>3.13.</b>	<b>SAMPLE PROGRAMS.....</b>	<b>17</b>
3.13.1.	CONNECT AND DISCONNECT TO CSQL.....	17
3.13.2.	INSERT AND SELECT.....	18
3.13.3.	UPDATE AND SELECT.....	20
3.13.4.	DELETE.....	22

## **4. ODBC AND CSQL ..... 25**

<b>4.1.</b>	<b>HEADER FILES AND LIBRARY.....</b>	<b>25</b>
<b>4.2.</b>	<b>DATA TYPES.....</b>	<b>25</b>
4.2.1.	TYPE IDENTIFIERS.....	25
4.2.2.	SQL DATA TYPES.....	26
4.2.3.	C DATA TYPES.....	26
<b>4.3.</b>	<b>FUNCTION RETURN CODES.....</b>	<b>27</b>
<b>4.4.</b>	<b>HANDLES.....</b>	<b>27</b>
4.4.1.	ENVIRONMENT.....	28
4.4.2.	CONNECTION.....	28
4.4.3.	STATEMENT.....	28
<b>4.5.</b>	<b>CONNECT TO THE DATABASE.....</b>	<b>29</b>
4.5.1.	ALLOCATE ENVIRONMENT HANDLE.....	29
4.5.2.	SETTING UP DRIVER VERSION.....	29
4.5.3.	ALLOCATE CONNECTION HANDLE.....	29
4.5.4.	CONNECT TO DATABASE USING.....	29
4.5.4.1.	SQLCONNECT().....	30
4.5.4.2.	SQLDRIVERCONNECT().....	30
<b>4.6.</b>	<b>DISCONNECT TO THE DATABASE.....</b>	<b>31</b>
4.6.1.	DISCONNECT USING SQLDISCONNECT.....	31
4.6.2.	FREING THE CONNECTION HANDLE AND ENVIRONMENT HANDLE.....	31
4.6.3.	A SAMPLE PROGRAM FOR CONNECT AND DISCONNECT TO THE DATABASE.....	32
<b>4.7.</b>	<b>USING ODBC TO SQL.....</b>	<b>33</b>
4.7.1.	CREATING AND DROPPING TABLE.....	33
4.7.2.	SAMPLE PROGRAM FOR CREATE AND DROP THE TABLE.....	34
4.7.3.	DATA MANIPULATION.....	36
4.7.3.1.	USING SQLPREPARE AND SQLEXECUTE.....	36
4.7.3.2.	USING SQLBINDPARAMETER FUNCTION.....	37
4.7.3.3.	INSERT.....	37
4.7.3.4.	UPDATE.....	40
4.7.3.5.	DELETE.....	43
4.7.4.	ISSUING A QUERY AND PROCESSING A RESULT.....	47
4.7.4.1.	SQLBINDCOL,SQLFETCH FUNCTION & SQLCLOSECURSOR.....	47
4.7.4.2.	A SAMPLE PROGRAM.....	48
4.7.5.	CLOSING TRANSACTION USING COMMIT AND ROLLBACK.....	50

<b>4.8. APIs IN DETAIL.....</b>	<b>51</b>
4.8.1. SQLALLOCHANDLE .....	51
4.8.2. SQLFREEHANDLE .....	52
4.8.3. SQLCONNECT .....	52
4.8.4. SQLDISCONNECT .....	53
4.8.5. SQLBINDCOL .....	53
4.8.6. SQLBINDPARAMETER .....	54
4.8.7. SQLPREPARE .....	55
4.8.8. SQLEXECUTE.....	55
4.8.9. SQLFETCH.....	56
4.8.10. SQLTRANSACTION .....	56

## **5. CSQL AND JDBC ..... 57**

<b>5.1. OVERVIEW OF JDBC .....</b>	<b>57</b>
<b>5.2. BACKGROUND .....</b>	<b>57</b>
<b>5.3. WHAT IS JDBC DRIVER.....</b>	<b>58</b>
<b>5.4. CLASSES, INTERFACES AND METHODS IN DETAILS.....</b>	<b>58</b>
5.4.1. INTERFACES .....	58
5.4.2. CLASSES.....	59
5.4.3. EXCEPTION CLASSES .....	59
<b>5.5. DATA TYPES .....</b>	<b>60</b>
<b>5.6. USING CSQL JDBC DRIVER.....</b>	<b>60</b>
5.6.1. GETTING STARTED .....	60
5.6.2. SETTING UP THE CLASSPATH .....	61
5.6.3. SETTING THE DEVELOPMENT ENVIRONMENT .....	61
5.6.4. LOADING THE DRIVER .....	61
5.6.5. CONNECTING TO THE DATABASE .....	62
5.6.6. CLOSING THE CONNECTION.....	62
<b>5.7. RUNNING SQL STATEMENTS WITH JDBC .....</b>	<b>63</b>
5.7.1. CREATE AND DROP.....	63
5.7.2. INSERT.....	63
5.7.3. SELECT (USING THE RESULTSET INTERFACE).....	64
5.7.4. UPDATE .....	65
5.7.5. DELETE.....	66
<b>5.8. TRANSACTION AND AUTO COMMIT MODE .....</b>	<b>66</b>
<b>5.9. CODE EXAMPLES .....</b>	<b>67</b>
<b>5.10. NETBEANS IDE CONFIGURATION FOR JDBC DRIVER.....</b>	<b>71</b>
5.10.1. CONNECTION THROUGH DRIVERMANAGER .....	71
5.10.1.1. CLASSPATH SETTING AND RUN THE APPLICATION: .....	72
5.10.2. CONNECTION THROUGH DATASOURCE.....	72
5.10.2.1. CLASSPATH SETTING AND RUN THE APPLICATION .....	73

# 1. Before you Start

Before you dip into this Guide, this chapter gives a introduction such as what is CSQL, Information in this manual, and Conventions used in this manual.

## 1.1. What is CSQL

CSQL is a main memory relational database cache system developed in SourceForge.net site, (world's largest development and download repository of open source code and applications.) CSQL includes main memory database as well as cache for traditional database, which both employ the same SQL interface, and a high availability option.

Primarily CSQL is meant for client side caching mechanism for any disk based database with undistruptive performance more than 20 times faster than traditional options. Using the JDBC and ODBC client interface application can increase its throughput it is suitable for real time acces of data .

## 1.2. Who should read this Book

The reader is expected to have the basic knowledge of following subjects

- SQL( Structured Query Language)
- C,C++ Programming language
- Relational database technology, CSQL server in particular
- JDBC and ODBC in general

## 1.3. Information in this guide

This guide is for application developers who use CSQL drivers and libraries to access data in the database programmatically.

## 1.4. Conventions

The below typographic and syntax conventions has been maintained throughout this guide.

### 1.4.1. Typographic rule

This manual uses the following typographic rule.

Table 1.1. Typographic Rule

<b>FORMAT</b>	<b>USED FOR</b>
Main memory database	This font is used for ordinary text.
<code>SELECT * FROM T1</code>	This font is used for SQL Statements and Program code.
<code>UNIQUE</code>	This font with uppercase letter indicates SQL keywords and data types.
<code>csql.conf</code>	This font indicates file name and path.
<code>build.ksh</code>	This font is used for command lines
<code>SQLFetch()</code>	This font is used for function names
<code>Java.Sql.Statement</code>	This font is used for interface, classes and header files

#### 1.4.2. Syntax Annotation

This manual uses the following syntax annotation conventions

Table 1.2. Syntax Annotation

<b>FORMAT</b>	<b>USED FOR</b>
[ ]	This square bracket indicates that item in command line is optional
{ }	Curly braces indicates that one must choose one of the items separated by a vertical bar(   ) in a command line.
	A vertical bar separates arguments
...	An ellipsis indicates that arguments can be repeated several times
.	This indicates that continuation of previous lines of code
\$	This dollar sign indicates the Linux prompt.
#	The pound sign indicates the Linux root prompt.

### 1.5. Technical support

Mainly CSQL provides the flexible support programs by Forums and Mailing. By providing the Developer's support, user can get the idea to implementation faster, eliminates defects that can derail your project and exceed your application's requirements.

Click the forum link: [http://sourceforge.net/forum/?group\\_id=165396](http://sourceforge.net/forum/?group_id=165396)

-----

## 2. Making Client Application

This Section gives you an overview of how to create a client application that will work with CSQL.

### 2.1. What is a client?

A client is a program that makes service requests (SQL Queries) to the server and get results back from the server. A client and Server is two different programs and in many cases client is running on a separate computer.

The client program cannot call functions in server directly; it uses shared memory communication mechanisms to communicate with the server. A specific library file need to link to your application so that your application can communicate with the server.

### 2.2. Passing query to the server

Mainly queries are written using SQL language. The client program sends the required queries to the server and after processing the server returns results back to the client. For example, Suppose the below query is send to the server by client

```
SELECT name FROM EMP WHERE eid=1001;
```

and server responses and send back the string **“Sumuthi.jain”**

Communication between client and server is usually done via a Driver, Such as ODBC Driver and JDBC Driver. If your client program follows the ODBC conventions, then your client program will be able to talk with Database server that follows the same conventions. The ODBC standard is generally used by programs written in the C and C++ programming language. And in the same manner the JAVA applications use JDBC Driver to connect to the database server that follows the JDBC Conventions. *In the section 4 and 5 , the ODBC and JDBC have been discussed in details.*

### 2.3. Supported Interfaces

The following is a list of interfaces currently supported by CSQL.

- SQL API (Proprietary Interface)
- ODBC Driver
- JDBC Driver

These interfaces enable applications to establish connections and process SQL statements simultaneously. In the above three APIs, SQL API is the Proprietary Interface. ODBC and JDBC are standard interfaces supported by CSQL.

### 3. SQLAPI (Proprietary Interface)

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.

#### 3.1. What is SQL API

SQLAPI is CSQL's Proprietary C++ interface. It is a highly efficient and convenient interface for CSQL main memory database. SQLAPI directly talks to the SQL Engine, It takes care of parsing the SQL statement and executing it using the storage manager which is used by DBAPI interface (Proprietary interface) . Primary sub modules of SQL engine are parser, executor, optimizer, etc.

CSQL's implementation says that SQL API is placed in below of the ODBC and JDBC, so that ODBC and JDBC driver will use SQLAPI to access the SQL Engine. Applications shall also use SQLAPI directly, which is a proprietary C++ interface.

#### 3.2. libcsqsql – C++ Library

The application should link to the libcsqsql.so library to access database. This library is available with the installation package of CSQL and is placed in 'lib' directory.

#### 3.3. Important classes and Header Files

The following are some important classes and header files for SQLAPI interface.

##### Classes

- SqlFactory - Create appropriate implementation for SQLAPI.
- AbsSqlConnection- It represents a database connection to SQL engine.
- AbsSqlStatement - Handle to the SQL statement.

##### Header Files

- AbsSqlStatement.h
- SqlFactory.h
- Info.h

### 3.4. Error Handling enum

DbRetVal is one such enum which is defined in ErrorType.h file and used by SQL API. It contains all the error codes returned by all the functions.

The following are some sample error codes :

```
OK = 0,  
ErrSysFatal = -1,  
ErrSysInit = -2,  
ErrNoPrivilege = -3,  
ErrSysInternal = -4,  
ErrNoExists = -5,  
ErrNoMemory = -6,  
. . .  
SplCase = -100
```

### 3.5. Data Type

The SQLAPI supports all primitive and non-primitive data types along with Date, Time and TimeStamp .

Table 1.3. Data Type

<b>Primitive</b>	<b>Non-Primitive</b>
typeInt	typeByteInt
typeLong	typeDate
typeLongLong	typeTime
typeFloat	typeTimeStamp
typeDouble	typeBinary
typeString	

### 3.6. Connect to the Database

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

```
AbsSqlConnection *con = NULL;
```

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

CSQL provides different modes to connection to it. Applications can use appropriate modes as mentioned below.

- **CSql** – provides connection to CSQL database locally.
- **CSqlAdapter** – provides connection to the target database locally.
- **CSqlGateway** – provides connection to the CSQL Gateway locally.
- **CSqlNetwork** – provides remote connection to CSQL database through network.
- **CSqlNetworkAdapter** – provides remote connection to target database through network.
- **CSqlNetworkGateway** – provides connection to the CSQL Gateway through network.

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

### 3.8. Create the table

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

```
Create 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 records in the table and how to drop the

Now you should prepare the SQL statement to create the table into the CSQL database. *Refer Section 3.13 for Sample Program.*

### 3.8.1. Prepare Statement

The SQL statements is prepared by the `prepare()` function before it executed by the `execute()` method.

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

Where `statement` is a memory location pointing to the string holding the SQL statement that needs to be prepared before 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);");
```

### 3.8.2. 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 the below function.

```
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.

## 3.9. Insert records into the table

Now let us insert some records into the table EMP.

### 3.9.1. 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 “parameterize” the fields of the table in the order mentioned during the definition of the table.

### 3.9.2. 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();
```

The default isolation level of CSQL is READ\_COMMITTED, if you need to start the transaction in other modes specify it as argument to beginTrans() method.

### 3.9.3. 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 the requirement in the example. It is done as shown below –

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

The 1<sup>st</sup> argument is the position of the '?' in the INSERT statement starting from 1 and the 2<sup>nd</sup> argument is the value to be picked up for respective field.

In our example, the 1<sup>st</sup> parameter to be inserted is integer, 2<sup>nd</sup> one is String and 3<sup>rd</sup> 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.

### 3.9.4. 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 for insert statement. Since this is an insert statement and always one row is inserted at a time.

### 3.9.5. 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. [Refer Section 3.13.2. for Source Code.](#)

## 3.10. Read the records from the table

Executing select queries using the `fetch()` functions in the database. Below sub sections shows the functions to do this and sample programs.

### 3.10.1. Read records from the table.

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

### 3.10.2. Prepare the statement

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

### 3.10.3. Bind the fields

Using `bindField()` function, the select statement able to produce the table records in a database. When this function is called the field values will be copied from the database to the buffer in memory .Using `fetch()` function the field value is generated.

```
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. Suppose 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 parameter position 1 and 2 respectively.

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

Suppose 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 parameter position 1 and 2 respectively.

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

### 3.10.4. 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 records and initialize the appropriate iterator for picking up the records.

### 3.10.5. Fetch the records

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

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

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

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

### 3.11. Update some records

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

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

Now call,

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

To execute the prepared statement, commit the transaction.

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

After committed the transaction, the 'rows' variable's value gives the number of updated rows in the table.

*Refer the Section 3.13.3. for the Source Code*

### 3.12. Delete records

For source code which deletes the tuples where EID > 1006. The statement for this one would be - Delete from EMP WHERE EID > 1006;

This statement is prepared using `prepare()` function. The transaction would have been started using `beginTrans()` to execute the statement using `execute()` function in the database. After executing the statement commit the transaction using `commit()` function.

```
stmt->prepare(statement);
con->beginTrans();
stmt->execute(rows);
conn->commit();
```

All the required fields are deleted based on the condition. Here 'rows' variable's value gives the number of deleted rows from the table.

Check for the deleted values by creating another select statement after freeing that statement. *Refer the Section 3.13.4 for example program.*

### 3.13. Sample Programs

Illustrating sample programs using SQL APIs.

#### 3.13.1. Connect and Disconnect to CSQL

```
/*
 * Sample SQL API application
 *
 * This Simple C++ application does the
 * following using CSQL SQL APIs.
 *
 * 1. Connects to the CSQL
 * 2. Disconnect from the database
 *
 */

#include<SqlStatement.h>
int main()
{
    DbRetVal rv = OK;
    SqlConnection *con = new SqlConnection();
    rv = con->connect("root", "manager");

    if (rv != OK) return 1;
    printf("Connection established\n");

    delete con;
```

```

    printf("Connection closed");

    return 0;
}

```

### 3.13.2. Insert and select

```

/*
 * Sample SQL API application
 *
 * This Simple C++ application does the
 * following using CSQL SQL APIS.
 *
 * 1. Connects to the CSQL
 * 2. Created table EMP(EID INT,ENAME CHAR(20),SALARY FLOAT
 * 3. Performing INSERT statement,
 * 4. using parameterized value
 *
 * 5. Creates SELECT query.
 * 6. Fetches and dumps all the rows.
 */

#include<SqlFactory.h>

int main()
{
    DbRetVal rv = OK;

    AbsSqlConnection *con =
    SqlFactory::createConnection(CSql);

    rv = con->connect("root", "manager");
    if(rv != OK) return 1;

    AbsSqlStatement *stmt =
    SqlFactory::createStatement(CSql);

    stmt->setConnection(con);
    char statement[200];

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

    int rows = 0;

```

```

rv = stmt->prepare(statement);
if(rv != OK) { delete stmt; delete con; return 2; }

rv = stmt->execute(rows);
if(rv!=OK) { delete stmt; delete con; return 3; }
stmt->free();
printf("Table Created\n");

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

int eid = 1000;
char ename[20] = "fasd";

char ename1[10][20]={"praba", "kishor", "jitu",
"biswa", "suman", "arin", "arabi", "sanjit", "sanjay", "rajesh"}
;

float salary = 0;
int count = 0;

rv = stmt->prepare(statement);
if(rv!=OK) { delete stmt; delete con; return 4; }

rv = con->beginTrans();
for(int i = 0; i < 10; i++)
{
    eid++;
    salary = salary + 1000;
    strcpy(ename, ename1[i]);
    stmt->setIntParam(1, eid);
    stmt->setStringParam(2, ename);
    stmt->setFloatParam(3, salary);

    rv = stmt->execute(rows); if(rv != OK) break;
    count++;
}
con->commit();
stmt->free();

printf("Total rows Inserted = %d\n", count);

//fetching the rows from EMP Table

strcpy(statement, "SELECT * FROM EMP;");
rv = stmt->prepare(statement);
if(rv!=OK) { delete stmt; delete con; return 5; }

```

```

stmt->bindParam(1, &eid);
stmt->bindParam(2, ename);
stmt->bindParam(3, &salary);
count=0;

rv = con->beginTransaction(); if(rv != OK) return 6;
stmt->execute(rows);

printf("\ninserted values are as follows\n");
printf("EmpId | name\t| salary\n");
printf("-----\n");
while (stmt->fetch() != NULL)

printf("%d | %s\t| %6.2f\n", eid, ename, salary);

rv = con->commit();
if(rv != OK){delete stmt; delete con; return 7; }

stmt->close();
stmt->free();
delete stmt;
delete con;
return 0;
}

```

### 3.13.3. Update and Select

```

/*
 * Sample SQL API application
 *
 * This Simple C++ application does the
 * following using CSQL SQL APIS.
 *
 * 1. Conencts to CSQL using SQL API
 * 2. Open the 'EMP' table
 *
 * 3. Performing UPDATE statement,
 *    Where EID = 1001, 1003, 1005
 *
 * 4. Creates SELECT query.
 * 5. Fetches and dumps all the rows.
 */
#include<SqlFactory.h>

```

```

#include <AbsSqlStatement.h>

int main()
{

DbRetVal rv = OK;
AbsSqlConnection *con = SqlFactory ::

rv = con->connect("root","manager");
if(rv!=OK)return 1;

AbsSqlStatement*stmt= SqlFactory :: createStatement(CSql);

stmt->setConnection(con);
char statement[200];

int rows = 0;

// UPDATE EMP WITH WHERE CLAUSE(EID<1006)
strcpy(statement, "UPDATE EMP SET SALARY=?, ENAME=? WHERE
EID=?;" );

rv = stmt->prepare(statement);
if(rv != OK) { delete stmt; delete con; return 3; }

int eid = 1001;
char ename[20];
float salary;
strcpy(ename, "Mani");

char *name[20]= { "Ravi", "Kiran", "Ganesh", "Yogesh",
"Vishnu" };

int i = 0;
while(true)
{
eid = 1001 + 2 * i;
if (eid > 1005) break;
strcpy(ename, name[i]);
salary = 1111.00 * (1 + i);
rv = con->beginTrans();
stmt->setFloatParam(1,salary);
stmt->setStringParam(2,ename);
stmt->setIntParam(3, eid);
if(rv != OK){ delete stmt; delete con;return 4; }

rv = stmt->execute(rows);

```

```

rv = con->commit();
if(rv != OK){delete stmt; delete con; return 5;}
i++;
}
stmt->free();

//fetching the rows from EMP Table

strcpy(statement, "SELECT * FROM EMP;");
rv = stmt->prepare(statement);
if(rv != OK) { delete stmt; delete con; return 6; }

int count=0;

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

printf("updated values are as follows\n");
printf("EmpId | name\t| salary\n");
printf("-----\n");

rv = con->beginTransaction();
if(rv != OK) { delete stmt; delete con; return 6; }
stmt->execute(rows);
while(stmt->fetch() != NULL)

printf("%d | %s\t| %6.2f\n", eid, &ename, salary);
rv = con->commit();

if(rv != OK) { delete stmt; delete con; return 7; }
stmt->close();
stmt->free();
delete stmt;
delete con;
return 0;
}

```

### 3.13.4. delete

```

/*
 * Sample SQL API application
 *
 * This Simple C++ application does the

```

```

* following using CSQL SQL APIS.
*
* 1. Conencts to CSQL using Driver
* 2. Open the 'EMP' table
* 3. Performing DELETE statement
* 4. Where EID > 1005
*
* 5. Creates SELECT query.
* 6. Fetches and dumps all the rows.
*/

#include<SqlFactory.h>
int main()
{
    DbRetVal rv = OK;
    AbsSqlConnection *con = SqlFactory ::
createConnection(CSql);

    rv = con->connect("root","manager");
    if(rv!=OK)return 1;

    AbsSqlStatement *stmt = SqlFactory ::
createStatement(CSql);

    stmt->setConnection(con);
    if(rv!=OK)return 2;
    char statement[200];
    int rows = 0;

    // deleteing tuples from EMP;

    int eid = 1000;
    char ename[20]="smith";
    float salary = 2000;

    strcpy(statement,"DELETE FROM EMP WHERE EID > 1005;");

    rv = stmt->prepare(statement);
    if(rv!=OK) { delete stmt; delete con; return 3; }
    rv = con->beginTrans();
    if(rv!=OK) { delete stmt; delete con; return 4; }
    rv = stmt->execute(rows);
    if(rv!=OK) { delete stmt; delete con; return 5; }
    printf("Rows Deleted successfully\n");
    rv = con->commit();
    if(rv!=OK) { delete stmt; delete con; return 6; }
    stmt->free();
}

```

```

// fetching the rows from EMP Table
strcpy(statement, "SELECT * FROM EMP;");
rv = stmt->prepare(statement);
if(rv!=OK) { delete stmt; delete con; return 7; }

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

printf("EmpId | name\t| salary\n");
printf("-----\n");

int count=0;
rv = con->beginTrans();
if(rv!=OK) return 6;
stmt->execute(rows);
while(stmt->fetch() !=NULL)

printf("%d | %s\t| %6.2f\n", eid, &ename, &salary);
rv = con->commit();
if(rv!=OK) { delete stmt; delete con; return 7; }
stmt->close();
stmt->free();

strcpy(statement, "DROP TABLE EMP;");
rv = stmt->prepare(statement);
if(rv!=OK) { delete stmt; delete con; return 9; }
rv = stmt->execute(rows);
if(rv!=OK) { delete stmt; delete con; return 10; }
stmt->free();
printf("Table dropped\n");
delete stmt;
delete con;
return 0;
}

```

## 4. ODBC and CSQL

This chapter is intended as a guide to ODBC Programming and help to get you started.

ODBC (Open Database Connectivity) is a standard set of interfaces to SQL-compliant databases for ODBC application and is one of the standard interfacing subsystems in CSQL database.

CSQL Provides ODBC Level 3 Driver and it indicates that most of the ODBC 3 APIs is supported and along with all-primitive data types, with the date, time and timestamp.

### 4.1. Header files and Library

The ODBC header files define the ODBC Functions to which your program calls in the ODBC Driver, and also the data types and constants that are used with ODBC Functions. The header files are standard files provided by Microsoft. The CSQL ODBC Driver implements the functions that are specified in these header files.

The following are the listed Header files.

- `Sql.h` - which contains most of the definitions you'll need
- `Sqltext.h` - which contains mostly additions for ODBC3
- `Sqlucode.h` - It is automatically included by `sqltext.h`
- `Sqltypes.h` - It is automatically included by `sql.h`

The CSQL provides a library `libcsqldb`. so with the install package. Your application directly link to this library to access CSQL Database and this makes as a interface between an application and CSQL.

### 4.2. Data Types

In general ODBC uses two set of data types: SQL data types and C data types. SQL data types are used in data source and C data types are used in C code in the application.

#### 4.2.1. Type Identifiers

To describe SQL and C data types, ODBC defines two sets of type identifiers and it describes the type of an SQL column or a C buffer. It is a macro is generally passed as a function argument.

The `SQLBindParameter` binds a variable of type `SQL_DATE_STRUCT` to a date parameter in an SQL statement. The C type identifier `SQL_C_TYPE_DATE` specifies the type of the date variable, and the SQL type identifier `SQL_TYPE_DATE` specifies the type of the dynamic parameter.

Following code is the example ,

```
SQLBindParameter ( stmt, 3, SQL_PARAM_INPUT,
SQL_C_TYPE_DATE, SQL_TYPE_DATE, 196, 0, &Date, sizeof (
Date ), &slen ) ;

SQL_DATE_STRUCT Date;
SQLINTEGER slen = SQL_NTS;
```

## 4.2.2. SQL Data types

SQL data types are the types in which data is stored in the data source.

Each database defines its own SQL data types. ODBC defines type identifier and describes the general characteristics of the SQL data types that might be mapped to each type identifier. It is completely driver-specific how each data type is mapped to an SQL type identifier of ODBC.

For Example, SQL\_CHAR is the type identifier for a character column with a fixed length (between 1 and 254 characters).SQL\_INTEGER for a integer column etc.

## 4.2.3. C Data Types

The CSQL ODBC driver application uses C data types to store values in application variables with their corresponding type identifiers. These are used as buffers that are bound to result set columns and statement parameter.

For Example, An application wants to retrieve the data from a result set column in character format. It have to declares a variable with the SQLCHAR \* data type and binds this variable to the result set column with a type identifier of SQL\_C\_CHAR.

The following table describes the C data types and SQL data types that CSQL ODBC Driver provides.

Table 1.4. CSQL ODBC Data Types

<b>SQL Definition</b>	<b>SQL Type Identifier</b>	<b>C Type Identifier</b>
CHAR(n)	SQL_CHAR	SQL_C_CHAR
SMALLINT	SQL_SMALLINT	SQL_C_SSHORT
INTEGER	SQL_INTEGER	SQL_C_SLONG
REAL	SQL_REAL	SQL_C_FLOAT
FLOAT	SQL_FLOAT	SQL_C_FLOAT
DOUBLE	SQL_DOUBLE	SQL_C_DOUBLE
TINYINT	SQL_TINYINT	SQL_C_TINYINT

BIGINT	SQL_BIGINT	SQL_C_SBIGINT
DATE	SQL_TYPE_DATE	SQL_C_TYPE_DATE
TIME	SQL_TYPE_TIME	SQL_C_TYPE_TIME
TIMESTAMP	SQL_TYPE_TIMESTAMP	SQL_C_TYPE_TIMESTAMP

### 4.3. Function Return Codes

Every ODBC function returns a return code that indicates whether the function invocation succeeded or failed.

The following table lists all possible return codes for CSQL ODBC Functions.

Table 1.5. CSQL ODBC function return codes

Return Code	Explanation
SQL_SUCCESS	The function completed successfully
SQL_SUCCESS_WITH_INFO	The function completed successfully with a warning.
SQL_NO_DATA_FOUND	The function returned successfully but no relevant data was found.
SQL_ERROR	The function failed
SQL_INVALID_HANDLE	The function failed due to an invalid input handle.

If the function returns SQL\_SUCCESS\_WITH\_INFO or SQL\_ERROR, the application can call SQLError to retrieve additional information about the error.

### 4.4. Handles

ODBC application uses a small set of handles to define basic features such as database connections and SQL statements. A handle is a 32-bit value.

The following handles are used in essentially all ODBC applications and are allocated by SQLAllocHandle() function.

The handle types required for ODBC programs are as follows:

Item	Handle Type
Environment	SQLHENV
Connection	SQLHDBC
Statement	SQLHSTMT

*Refer the Section 4.8.* to know details about SQLAllocHandle() function and its argument.

### 4.4.1. Environment

The environment handle provides a global context in which to access data. Every ODBC application must allocate exactly one environment handle upon starting, and must free it at the end.

The following code illustrates how to allocate an environment handle:

```
SQLHENV  env;  
SQLRETURN rc;  
  
rc=SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&env);
```

*Please Refer the Secton:4.8.* To know details about SQLAllocHandle function and its arguments.

### 4.4.2. Connection

A connection is specified by an ODBC Driver and a data source. An application can have several connections associated with its environment. Allocating a connection handle does not establish a connection; a connection handle must be allocated first and then used when the connection is established.

The following code illustrates how to allocate a connection handle:

```
SQLHDBC  dbc ;  
SQLRETURN rc ;  
rc = SQLAllocHandle ( SQL_HANDLE_DBC, env, &dbc) ;
```

### 4.4.3. Statement

A statement handle provides access to a SQL statement and any information associated with it, such as result sets and parameters. Each connection can have several statements. Statements are used both for cursor operations (fetching data) and for single statement execution (e.g. INSERT, UPDATE, and DELETE).

The following code illustrates how to allocate a statement handle:

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

## 4.5. Connect to the Database

Follow the basic steps to connect to CSQL

### 4.5.1. Allocate Environment Handle

An environment handle provide access to global information. To request an environment handle in your application, call `SQLAllocHandle( )` with the `HandleType` argument set to `SQL_HANDLE_ENV` and the `InputHandle` argument set to `SQL_NULL_HANDLE`. CSQL ODBC allocates the environment handle and passes the value of the associated handle to the `*OutputHandlePtr` argument. After allocating Environment handle, the driver version needs to be set.

### 4.5.2. Setting up Driver Version

After allocating an environment handle, an application must call `SQLSetEnvAttr` on the environment handle to set the `SQL_ATTR_ODBC_VERSION = SQL_OV_ODBC3` environment attribute. This requires that our application specify which version of the ODBC API it is using before you go on to allocate Connection handles. *Refer the Section 4.8.* to know details about `SQLSetEnvAttr( )` function.

### 4.5.3. Allocate Connection Handle

A connection handle provides access to information such as the valid statement handles on the connection and an indication of whether a transaction is currently open. Using `SQLAllocHandle( )` with the `HandleType` argument set to `SQL_HANDLE_DBC` and set the `InputHandle` argument to the current environment handle.

This function will be called only after setting up the driver version using `SQLSetAttr` function.

### 4.5.4. Connect to Database using

For the connection to CSQL database through ODBC Driver, you can prefer any one of the following methods.

- `SQLConnect()`
- `SQLDriverConnect()`

This method needs to be called after allocating connection handle and before allocate statement handle. Before starting connection you need to know connection sting to connect CSQL

The general form of connection string is

```
DSN=<DSN Name>;MODE=<CSQL Mode>;SERVER=<Server Name>;PORT=<Port No>;
```

CSQL supports different modes in connection string which is mentioned below.

MODE	USES
Csql	To connect CSQL database
Adapter	To connect CSQL through adapter for any operation in target database directly.
Gateway	To connect CSQL through gateway for any operation in target database as well as csql
Network	To connect CSQL database with network.
NetworkAdapter	To connect CSQL through adapter for any operation in target database directly with network
NetworkGateway	To connect CSQL through gateway for any operation in target database as well as csql with network

#### 4.5.4.1. SQLConnect()

SQLConnect( ) establishes a connection to the target database. The application must supply a target SQL database.

This function needs to be call after allocating Connection handle and before allocates Statement Handle.

The following code illustrates how to connect to CSQL.

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

**Note:** CSQL also accepts connection string as 2<sup>nd</sup> argument of SQLConnect( )

#### 4.5.4.2. SQLDriverConnect()

The following code illustrates how to connect to CSQL through SQLDriverConnect.

```
ret =SQLDriverConnect (dbc,
SQLHWND)WindowHandle,
    (SQLCHAR *)csqlConnectionString,
    (SQLSMALLINT)StringLength,
    (SQLCHAR *)OutConnectionString,
    (SQLSMALLINT)BufferLength,
    (SQLSMALLINT *)StringLength2Ptr,
    (SQLUSMALLINT)DriverCompletion
```

*Refer section 4.9.* to know details about the SQLConnect( ) function and arguments.

## 4.6. Disconnect to the Database

Follow the steps to disconnect from CSQL

### 4.6.1. Disconnect using SQLDisconnect

SQLDisconnect( ) closes the connection that is associated with the database connection handle. Before you call SQLDisconnect(), you must call SQLEndTran( ) if an outstanding transaction exists on this connection.

The following code illustrates how to connect to CSQL.

```
ret = SQLDisconnect (dbc);
```

*Refer the Section 4.9.* to know about SQLDisconnect and its arguments.

### 4.6.2. Freeing the Connection handle and Environment handle

SQLFreeHandle( ) frees an environment handle, a connection handle, or a Statement handle.

The following code illustrates how to freeing the Connection and Environment handle

```
ret = SQLFreeHandle (SQL_HANDLE_DBC, dbc);
ret = SQLFreeHandle (SQL_HANDLE_DBC, env);
```

*Refer section 4.9.* to know in details about SQLFreeHandle.

### 4.6.3. A sample program for Connect and Disconnect to the database.

```
/*
 * Sample ODBC application
 *
 * This Simple ODBC application does the
 * following using CSQL ODBC Driver.
 *
 *     1. Loading the Driver
 *     2. Connects to CSQL using Driver
 *     3. Closes the connection
 */

#include<stdio.h>
#include<stdlib.h>
#include<CSql.h>
#include<sql.h>
#include<sqlext.h>

int main()
{
    SQLHSTMT stmt;
    SQLRETURN ret;
    SQLCHAR outstr[1024];
    SQLSMALLINT  outstrlen;

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

    ret
    =SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODB
    C3, 0);
    checkrc(ret, __LINE__);

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

    ret = SQLConnect (dbc,
        (SQLCHAR *) "test",
        (SQLSMALLINT) strlen ("test"),
        (SQLCHAR *) "root",
        (SQLSMALLINT) strlen ("root"),
        (SQLCHAR *) "manager",
        (SQLSMALLINT) strlen (""));
}
```

```

if(SQL_SUCCEEDED(ret))
{
    printf("\nConnect to the data source successfully\n");
}
else
{
    printf("Failed to connect\n");
    return 2;
}

ret = SQLDisconnect(dbc);
checkrc(ret, __LINE__);

ret = SQLFreeHandle(SQL_HANDLE_DBC, dbc);
checkrc(ret, __LINE__);

ret = SQLFreeHandle(SQL_HANDLE_ENV, env);
checkrc(ret, __LINE__);
return 0;
}

```

## 4.7. Using ODBC to SQL

ODBC is a widely accepted application programming interface (API) for database access and uses Structured Query Language (SQL) as its database access language.

### 4.7.1. Creating and Dropping table

SQLExecDirect( ) prepares and executes the SQL statement in one step. It uses the current values of the parameter marker variables, if any parameter exists in the statement. The statement can only be executed once. The following code illustrates how to create a table

#### Creation of emp Table

```

SQLHSTMT  stmt ;
SQLRETURN ret ;

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

ret = SQLExecDirect(stmt, table, SQL_NTS);
if (SQL_SUCCEEDED(ret))
{
    printf("Table 'emp' is created");
}

```

## Dropping of emp Table

```
SQLCHAR table[100]= "drop table emp";
ret = SQLExecDirect(stmt,table,SQL_NTS);
if (SQL_SUCCEEDED(ret))
{
    printf("Table 'emp' is dropped");
}
```

*Refer the section 4.9.* to know details about SQLExecDirect function.

### 4.7.2. Sample Program for Create and Drop the table

```
/*
 * Sample ODBC application
 *
 * This Simple ODBC application does the
 * following using CSQL ODBC Driver.
 *
 *     1. Loading the Driver
 *     2. Connects to CSQL using Driver
 *     3. Create table 'emp' with 3 fields
 *     4. Drop the table
 *     5. Close the Connection.
 */

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sql.h>
#include<sqlext.h>

inline void checkrc(int rc, int line)
{
    if(rc)
    {
        printf("ERROR %d at line %d\n",rc,line);
        exit(1);
    }
}
```

```

int main()
{
    SQLHENV env;
    SQLHDBC dbc;
    SQLHSTMT stmt;
    SQLRETURN ret;
    SQLCHAR outstr[1024];
    SQLSMALLINT outstrlen;

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

    ret=SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_
    ODBC3, 0);
    checkrc(ret, __LINE__);

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

    ret=SQLConnect(dbc,
                   (SQLCHAR*)"test",
                   (SQLSMALLINT) strlen("test"),
                   (SQLCHAR *) "root",
                   (SQLSMALLINT) strlen("root"),
                   (SQLCHAR *) "manager",
                   (SQLSMALLINT) strlen(""));

    if(SQL_SUCCEEDED(ret))
    {
        printf("\nConnect to the data source successfully\n");
    }
    else
    {
        printf("Failed to connect\n");
        return 2;
    }
    ret = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
    checkrc(ret, __LINE__);

    // Table creation

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

    ret = SQLExecDirect(stmt, table, SQL_NTS);
    checkrc(ret, __LINE__);

```

```

printf("Table 'emp' created \n");
SQLCHAR table1[50] = "drop table emp;" ;
ret = SQLExecDirect(stmt, table1,SQL_NTS);

if(SQL_SUCCEEDED(ret))
{
printf("\ndrop the 'emp' table \n");
}
ret = SQLFreeHandle(SQL_HANDLE_STMT,stmt);
checkrc(ret,__LINE__);

ret = SQLDisconnect(dbc);
checkrc(ret,__LINE__);

ret = SQLFreeHandle(SQL_HANDLE_DBC,dbc);
checkrc(ret,__LINE__);

ret = SQLFreeHandle(SQL_HANDLE_ENV,env);
checkrc(ret,__LINE__);
return 0;
}

```

### 4.7.3. Data Manipulation

#### 4.7.3.1. Using SQLPrepare and SQLExecute

##### SQLPrepare

This function compiles a SQL Statement and stores the information in the provided statement handle. The application can reference this prepared statement by passing the statement handle to other functions.

If the statement handle is previously used with a query statement, SQLFreeStmt( ) must be called before calling SQLPrepare( ) .

The SQL statement string might contain parameter markers. A parameter marker represented by a '?' character and is used to indicate a position in the statement in which an application-supplied value is to be substituted during the execution.

##### SQLExecute

SQLExecute() executes a statement, that is successfully prepared using SQLPrepare(), one or multiple times. The statement is executed using the current value of any application variables that were bound to parameter markers by SQLBindParameter() .

All parameters must be bound before calling SQLExecute() .

#### 4.7.3.2. Using SQLBindParameter Function

SQLBindParameter() is used to bind parameter markers in an SQL statement to application variables, for all C data types. In this case data is transferred from the application to the DBMS when SQLExecute() is called.

A parameter marker is represented by a ‘?’ character in an SQL statement and is used to indicate a position in the statement where an application –supplied value is to be substituted when the statement is executed. This value can be obtained from an application variable.

*Refer the section 4.9.* to know details about SQLBindParameter() , SQLPrepare() and SQLExecute() and its arguments.

#### 4.7.3.3. INSERT

Using functions SQLPrepare,SQLBindParameter and SQLExecute , you can insert records into the table.

```
/*
 * Sample ODBC application
 *
 * This Simple ODBC application does the
 * following using CSQL ODBC Driver.
 *
 *     1. Loading the Driver
 *     2. Connects to CSQL using Driver
 *     3. Insert 10 rows in it.
 *     4. Close the Connection.
 */

#include<stdio.h>
#include<stdlib.h>
#include<sql.h>
#include<sqlext.h>
#include<string.h>
inline void checkrc(int rc, int line)
```

```

{
    if(rc)
    {
printf("ERROR %d at line %d\n",rc,line);

exit(1);
    }
}

int main()
{
    SQLHENV env;
    SQLHDBC dbc;
    SQLHSTMT stmt;
    SQLRETURN ret;
    SQLCHAR outstr[1024];
    SQLSMALLINT outstrlen;

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

ret=
SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC
3,0);
checkrc(ret, __LINE__);

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

ret = SQLConnect (dbc,
(SQLCHAR*)"test", (SQLSMALLINT) strlen ("test"),
                (SQLCHAR *) "root",
                (SQLSMALLINT)strlen ("root"),
                (SQLCHAR *) "manager",
                (SQLSMALLINT) strlen (""));

if(SQL_SUCCEEDED(ret))
{
    printf("\nConnect to the data source successfully\n");
}

else
{
    printf("Failed to connect\n");
}
}

```

```

        return 2;
    }

    ret = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
    checkrc(ret, __LINE__);

    // inserting records into the table.

    int eid1 = 1000;
    char ename1[20]="jitu";
    float salary1 = 5500;

    char ename2[10][20]=
    {"Praba", "Kishor", "Jitu", "Sanjit", "Sanjay", "Bisi", "Suman",
    "Vikrant", "Eti", "Suba"};

    SQLINTEGER slen=SQL_NTS;

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

    ret=
    SQLBindParameter(stmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_IN
    TEGER, 0, 0, &eid1, 0, NULL);
    checkrc(ret, __LINE__);

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

    ret=
    SQLBindParameter(stmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_RE
    AL, 0, 0, &salary1, 0, NULL);
    checkrc(ret, __LINE__);

    int i;
    int count=0;
    for( i=0;i<10;i++)
    {
        eid1++;
        salary1 = salary1 + 1000;
        strcpy(ename1, ename2[i]);
        ret = SQLExecute(stmt);
        checkrc(ret, __LINE__);
    }

```

```

    ret = SQLTransact(env, dbc, SQL_COMMIT);
    checkrc(ret, __LINE__);

    count++;
}
printf("%d rows inserted in 'emp' table\n", count);

ret = SQLFreeHandle(SQL_HANDLE_STMT, stmt);
checkrc(ret, __LINE__);

ret = SQLDisconnect(dbc);
checkrc(ret, __LINE__);

ret = SQLFreeHandle(SQL_HANDLE_DBC, dbc);
checkrc(ret, __LINE__);

ret = SQLFreeHandle(SQL_HANDLE_ENV, env);
checkrc(ret, __LINE__);
return 0;
}

```

#### 4.7.3.4. UPDATE

Using the above three functions, you can update rows.

```

/*
 * Sample ODBC application
 *
 * This Simple ODBC application does the
 * following using CSQL ODBC Driver.
 *
 *     1. Loading the Driver
 *     2. Connects to CSQL using Driver
 *     3. Update 5 rows in it.
 *     4. Close the Connection.
 */
#include<stdio.h>
#include<stdlib.h>
#include<sql.h>
#include<sqlext.h>
#include<string.h>
inline void checkrc(int rc, int line)
{
    if(rc)
    {

```

```

        printf("ERROR %d at line %d\n",rc,line);
        exit(1);
    }
}

int main()
{
    SQLHENV env;
    SQLHDBC dbc;
    SQLHSTMT stmt;
    SQLRETURN ret;
    SQLCHAR outstr[1024];
    SQLSMALLINT outstrlen;

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

    ret=
    SQLSetEnvAttr(env,SQL_ATTR_ODBC_VERSION,(void*)SQL_OV_ODBC
    3,0);
    checkrc(ret,__LINE__);

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

    ret = SQLConnect (dbc,
        (SQLCHAR *) "test",
        (SQLSMALLINT) strlen ("test"),
        (SQLCHAR *) "root",
        (SQLSMALLINT) strlen ("root"),
        (SQLCHAR *) "manager",
        (SQLSMALLINT) strlen (""));

    if(SQL_SUCCEEDED(ret))
    {
        printf("\nConnect to the data source successfully\n");
    }
    else
    {
        printf("Failed to connect\n");
        return 2;
    }

    ret = SQLAllocHandle(SQL_HANDLE_STMT,dbc,&stmt);
    checkrc(ret,__LINE__);

```

```

// update rows in the table

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

    int eid1;
    int eid2;
    float salary1;

ret=
SQLBindParameter(stmt,1,SQL_PARAM_INPUT,SQL_C_SLONG,SQL_IN
TEGER,0,0,&eid1,0,NULL);
checkrc(ret,__LINE__);

ret=
SQLBindParameter(stmt,2,SQL_PARAM_INPUT,SQL_C_FLOAT,SQL_RE
AL,0,0,&salary1,0,NULL);
checkrc(ret,__LINE__);

ret=
SQLBindParameter(stmt,3,SQL_PARAM_INPUT,SQL_C_SLONG,SQL_IN
TEGER,0,0,&eid2,0,NULL);

int i,count=0;
eid1=5000;
for(i=0;i<5;i++)
{
    eid2 = 1001 + i;
    eid1++;
    salary1= 20500+(500*i);

    ret=SQLExecute(stmt);
    checkrc(ret,__LINE__);

    ret = SQLTransact(env,dbc,SQL_COMMIT);
    checkrc(ret,__LINE__);
    count++;
}

printf("%d rows updated in 'emp' table\n",count);

//fetch updated rows

```

```

char ename1[10]="jitu";

ret = SQLPrepare(stmt,(unsigned char*)"select * from emp
where salary > 20000;",SQL_NTS);
checkrc(ret,__LINE__);

ret = SQLBindCol(stmt,1,SQL_C_SLONG,&eid1,0,NULL);
checkrc(ret,__LINE__);

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

ret = SQLBindCol(stmt,3,SQL_C_FLOAT,&salary1,0,NULL);
checkrc(ret,__LINE__);

count=0;
ret = SQLExecute(stmt);
checkrc(ret,__LINE__);

printf("select updated rows\n");
while(SQL_SUCCEEDED(ret=SQLFetch(stmt)))
{
    printf("eid=%d\tename=%s\tsal=%f\n",eid1,ename1,salary);
    count++;
}

ret = SQLCloseCursor(stmt);
checkrc(ret,__LINE__);

ret = SQLFreeHandle(SQL_HANDLE_STMT,stmt);
checkrc(ret,__LINE__);

ret = SQLDisconnect(dbc);
checkrc(ret,__LINE__);

ret = SQLFreeHandle(SQL_HANDLE_DBC,dbc);
checkrc(ret,__LINE__);

ret = SQLFreeHandle(SQL_HANDLE_ENV,env);
checkrc(ret,__LINE__);
return 0;

}

```

#### 4.7.3.5. DELETE

The following codes delete the rows in the table

```
/*
 * Sample ODBC application
 *
 * This Simple ODBC application does the
 * following using CSQL ODBC Driver.
 *
 *     1. Loading the Driver
 *     2. Connects to CSQL using Driver
 *     3. Update 5 rows in it.
 *     4. Close the Connection.
 */

#include<stdio.h>
#include<stdlib.h>
#include<sql.h>
#include<sqlext.h>
#include<string.h>
inline void checkrc(int rc, int line)
{
    if(rc)
    {
        printf("ERROR %d at line %d\n",rc,line);
        exit(1);
    }
}

int main()
{
    SQLHENV env;
    SQLHDBC dbc;
    SQLHSTMT stmt;
    SQLRETURN ret;
    SQLCHAR outstr[1024];
    SQLSMALLINT outstrlen;

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

    ret = SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC
    3, 0);
    checkrc(ret, __LINE__);

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

```

checkrc(ret, __LINE__);

ret = SQLConnect (dbc,
                 (SQLCHAR *) "test",
                 (SQLSMALLINT) strlen ("test"),
                 (SQLCHAR *) "root",
                 (SQLSMALLINT) strlen ("root"),
                 (SQLCHAR *) "manager",
                 (SQLSMALLINT) strlen (""));

if (SQL_SUCCEEDED(ret))
{
    printf("\nConnect to the data source successfully\n");
}
else
{
    printf("Failed to connect\n");
    return 2;
}

ret = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
checkrc(ret, __LINE__);

// delete 5 rows from the table with parameter.

int eid1;

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

ret=
SQLBindParameter(stmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, 0, 0, &eid1, 0, NULL);
checkrc(ret, __LINE__);

int i, count=0;
eid1=5000;

for(i=0; i<5; i++)
{
    eid1++;
    ret = SQLExecute(stmt);
    checkrc(ret, __LINE__);

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

```

```

    checkrc(ret, __LINE__);

    ret = SQLTransact(env, dbc, SQL_COMMIT);
    count++;
}
printf("Deleted first %d rows in 'emp' table\n", count);

// delete all the rows from the table

ret = SQLPrepare(stmt, (unsigned char*)"delete          from
emp;", SQL_NTS);
checkrc(ret, __LINE__);

    ret = SQLExecute(stmt);
    checkrc(ret, __LINE__);

    ret = SQLTransact(env, dbc, SQL_COMMIT);
    checkrc(ret, __LINE__);

printf("All the rows are deleted \n");

// drop the table from the database

SQLCHAR drop[30]="drop table emp;";

ret = SQLExecDirect(stmt, drop, SQL_NTS);
checkrc(ret, __LINE__);

printf("Table 'emp' dropped successfully\n");

ret = SQLFreeHandle(SQL_HANDLE_STMT, stmt);
checkrc(ret, __LINE__);

ret = SQLDisconnect(dbc);
checkrc(ret, __LINE__);

ret = SQLFreeHandle(SQL_HANDLE_DBC, dbc);
checkrc(ret, __LINE__);

ret = SQLFreeHandle(SQL_HANDLE_ENV, env);
checkrc(ret, __LINE__);
return 0;

}

```

#### 4.7.4. Issuing a Query and processing a result

When you want to issue SQL statements to the database, you require select statements which could be Prepared by the SQLPrepare .Once you have a prepare statement, you can issue a query.

##### 4.7.4.1. SQLBindCol,SQLFetch function & SQLCloseCursor

###### SQLBindCol

The application binds a column by calling SQLBindCol .This function binds one column at a time.

SQLBindCol( ) bind columns in a result set to application variables(storage buffers), for all data types. Data is transferred from the DBMS to the application when SQLFetch( ) is called.

This function is also used to specify any data conversion required. It is called once for each column in the result set that the application needs to retrieve.

SQLPrepare( ) is usually called before this function and SQLBindCol( ) must be called before SQLFetch( ) to transfer data to the storage buffers specified by this call.

###### SQLFetch

This function advances the cursor to the next row of the result set, and retrieves any bound columns.

This function can be used to receive the data directly into variables you specify with SQLBindCol( ) .

###### SQLCloseCursor

This function closes the open cursor on a statement handle. When SQLExecute function is called, it creates a result set and opens the cursor in case of SELECT statement. After finished working with result set, we should close the cursor and free the memory by using this Function. [Refer the Section 4.9.](#) to know about this two Functions and their arguments.

#### 4.7.4.2. A sample Program

```
/*
 * Sample ODBC application
 *
 * This Simple ODBC application does the
 * following using CSQL ODBC Driver.
 *
 *     1. Loading the Driver
 *     2. Connects to CSQL using Driver
 *     3. Select all the rows from 'emp'table
 *     4. Close the Connection.
 */

#include<stdio.h>
#include<stdlib.h>
#include<sql.h>
#include<sqlext.h>
#include<string.h>

inline void checkrc(int rc, int line)
{
    if(rc)
    {
        printf("ERROR %d at line %d\n",rc,line);
        exit(1);
    }
}

int main()
{
    SQLHENV env;
    SQLHDBC dbc;
    SQLHSTMT stmt;
    SQLRETURN ret;
    SQLCHAR outstr[1024];
    SQLSMALLINT outstrlen;

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

    ret = SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC
    3, 0);
    checkrc(ret, __LINE__);
}
```

```

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

ret = SQLConnect (dbc,
(SQLCHAR *) "test", (SQLSMALLINT) strlen ("test"),
(SQLCHAR *) "root",
(SQLSMALLINT) strlen ("root"),
(SQLCHAR *) "manager",
(SQLSMALLINT) strlen (""));

if (SQL_SUCCEEDED(ret))
{
printf("\nConnect to the data source successfully\n");
}
else
{
printf("Failed to connect\n");
return 2;
}

ret = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
checkrc(ret, __LINE__);

/* fetch the rows from the table */

int eid1=100;
char ename1[20]="jitu";
float salary1 = 2000;
int j, count=0;

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

ret =
SQLBindCol(stmt, 1, SQL_C_SLONG, &eid1, 0, NULL);
checkrc(ret, __LINE__);

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

ret = SQLBindCol(stmt, 3, SQL_C_FLOAT, &salary1, 0, NULL);
checkrc(ret, __LINE__);

```

```

ret = SQLExecute(stmt);
checkrc(ret, __LINE__);

printf("Fetching starts on 'emp' table\n ");

while(SQL_SUCCEEDED(ret = SQLFetch(stmt)))
{
    printf("eid=%d\tename=%s\tsalary=%f\n", eid1, ename1,
        salary1);

    count++;
}

ret = SQLCloseCursor(stmt);
checkrc(ret, __LINE__);

ret = SQLTransact(env, dbc, SQL_COMMIT);
checkrc(ret, __LINE__);

printf("Total row fetched=%d\n", count);

ret = SQLFreeHandle(SQL_HANDLE_STMT, stmt);
checkrc(ret, __LINE__);

ret = SQLDisconnect(dbc);
checkrc(ret, __LINE__);

ret = SQLFreeHandle(SQL_HANDLE_DBC, dbc);
checkrc(ret, __LINE__);

ret = SQLFreeHandle(SQL_HANDLE_ENV, env);
checkrc(ret, __LINE__);
return 0;
}

```

#### 4.7.5. Closing transaction using COMMIT and ROLLBACK

For completing a transaction, commit or rollback is used. Issuing Commit end the transaction permanently which is opposite of Rollback.

SQLTransact function is used to complete the commit or rollback the operations performed. If a transaction is active on a connection, the application must call SQLTransact( ) before it can disconnect from the database.

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

*Refer section 4.9.* to know in details about SQLTransact and its arguments.

## 4.8. APIs in Detail

This section documents the ODBC functions that exhibit CSQL specific behaviors when used with the CSQL Native Client ODBC Driver.

### 4.8.1. SQLAllocHandle

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

This function replaces the old allocation functions for each individual handle types (SQLAllocEnv, SQLAllocConnection and SQLAllocStmt).

Syntax :

```
RETCODE          SQLAllocHandle(  handleType,  inputHandle,
outputHandle )
```

Arguments :

The arguments for SQLAllocHandle are listed in this Table

Name	Type	Description
HandleType	SQLSMALLINT	The type of handle to allocate
InputHandle	SQLHANDLE	The handle to base on the new handle. This is either an environment or connection handle.
OutputHandle	SQLHANDLE*	Pointer to the storage for the newly create handle

Usage :

An application allocates different handles to use with different API functions. The handle provides a context for each function. The supported handle types are listed in below table.

Handle	Type	Description
Environment	SQL_TYPE_ENV	This handle is used to create an environment

Connection	SQL-TYPE_DBC	A connection handle is used to open a connection to a specific CSQL Database
Statement	SQL_TYPE_STMT	The statement handle contains information about the compiled SQL statement and its result set.

Returns : SQLAllocHandle returns SQL\_SUCCESS if it is successful. Otherwise it returns SQL\_ERROR.

### 4.8.2. SQLFreeHandle

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

Syntax :

```
RETCODE SQLFreeHandle ( handleType, handle ) ;
```

Arguments :

The arguments fro SQLFreeHandle are listed in below table.

Type	Name	Description
SQLSMALLINT	HandleType	The type of handle to free
SQLHANDLE	Handle	The handle to free

Returns :SQLFreHandle returns SQL\_SUCCESS if it is successful else SQL\_ERROR.

### 4.8.3. SQLConnect

SQLConnect connects to a database and saves information about the connection in the provided connection handle. The handle must be previously allocated using the SQLAllocHandle function.

Syntax :

```
RETCODE SQLConnect ( hConn, dbName, dbNameLen, userName,
userNameLen, auth, authLen ) ;
```

Arguments :

The arguments for SQLConnect are listed in below table.

Type	Name	Description
SQLHDBC	hConn	Newly allocated connection handle.
SQLCHAR *	dbName	Name of the database to connect.
SQLSMALLINT	dbNameLen	Length of the database name.
SQLCHAR *	Username	Name of the database user.
SQLSMALLINT	UserNameLen	Length of the username.
SQLCHAR*	Auth	Database encryption password.
SQLSMALLINT	authLen	Password length.

Returns :SQLConnect returns SQL\_SUCCESS if it is successful else SQL\_ERROR.

#### 4.8.4. SQLDisconnect

This function disconnects and closes a previously connected database. If the environment used to make the connection is not committed before the connection is closed, committing afterwards fails.

Syntax :

```
RETCODE SQLDisconnect ( hDbc ) ;
```

#### 4.8.5. SQLBindCol

This binds a buffer to a column in the result set. The buffer is updated when SQLFetch is called. New columns from the result set are then read in.

SQLBindCol can be called after or before the statement is prepared and executed, as long as it is called before SQLFetch is called.

Syntax :

```
RETCODE SQLBindCol ( hStmt, columnNo, targetType, targetValue, targetSize, actualSize )
```

Arguments :

The arguments fro SQLBindCol are listed in below table.

Type	Name	Description
SQLHSTMT	hStmt	Statement handle
SQLUSMALLINT	columnNo	The number of the column of the result set to bind to.
SQLSMALLINT	targetType	The C data type of the buffer
SQLPOINTER	targetValue	Pointer to buffer to hold the column data
SQLINTEGER	targetSize	Size of the buffer in bytes
SQLINTEGER *	actualSize	Pointer buffer to hold the size of the data.

Returns :This function returns SQL-SUCCESS if it is successful , else SQL\_ERROR.

#### 4.8.6. SQLBindParameter

SQLBindParameter binds a data buffer to a parameter marker in a SQL statement .Parameter markers are denoted by “?” in the SQL statement.

Syntax :

```
RETCODE SQLBindParameter ( hStmt, paramNo, paramType,
cType, sqlType, colDef, scale, value, valueMaxSize,
valueSize )
```

Arguments :

The arguments for SQLBindParameter are listed in below table.

Type	Name	Description
SQLHSTMT	hStmt	Statement handle
SQLUSMALLINT	ParamNo	The number of the parameter marker to bind to. Starts from 1, counted from left to right.
SQLSMALLINT	paramType	Parameter type
SQLSMALLINT	cType	The C data type of the parameter
SQLSMALLINT	sqlType	The SQL data type of the parameter
SQLINTEGER	colDef	The precision of the parameter
SQLSMALLINT	Scale	The scale of the parameter

SQLPOINTER	Value	Pointer to the buffer where the parameter value is stored
SQLINTEGER	valueMaxSize	The size of the parameter marker.
SQLINTEGER *	valueSize	Actual size of the parameter value.

Returns : This function returns SQL-SUCCESS if it is successful else SQL-ERROR.

#### 4.8.7. SQLPrepare

SQLPrepare compiles a SQL statement and stores the information in the provided statement handle.

Syntax :

```
RETCODE SQLPrepare ( hStmt, statement, statementLen )
```

Arguments :

Type	Name	Description
SQLHSTMT	hStmt	Statement handle
SQLCHAR *	Statement	SQL Statement string
SQLINTEGER	StatementLen	Length of the SQL statement

#### 4.8.8. SQLExecute

This executes the prepared SQL using SQLPrepare.

Syntax :

```
RETCODE SQLExecute ( hStmt )
```

Arguments :

The arguments for SQLExecute are listed in below table.

Type	Name	Description
SQLHSTMT	hStmt	Statement handle

Returns : This function returns SQL-SUCCESS upon successful, else SQL\_ERROR.

#### 4.8.9. SQLFetch

This function reads in a row of data from the result set. After calling the function, the cursor is positioned to the next row to be read.

If the application called SQLBindCol to bind columns, SQLFetch stores data from the row in the specified buffers.

Syntax :

```
RETCODE SQLFetch ( hStmt )
```

Arguments :

Type	Name	Description
SQLHSTMT	hStmt	Statement handle

Returns :SQLFetch returns SQL\_SUCCESS if a new row of data is read successfully.

#### 4.8.10. SQLTransact

This function requests a commit or rollback for all active operations on all statements associated with an environment.

Syntax :

```
RETCODE SQLTransact ( hEnv, hDbc, completionType )
```

Arguments :

The arguments for SQLTransact are listed in below table.

Type	Name	Description
SQLHENV	hEnv	Environment handle
SQLHDBC	hDbc	Connection handle
SQLUSMALLINT	completionType	The transaction action, which could be either SQL_COMMIT or SQL-ROLLBACK.

## 5. CSQL and JDBC

This chapter gives an overview on writing applications on CSQL JDBC Driver. The CSQL JDBC Driver 2.0 is a JDBC type 2 driver for CSQL database. This chapter intends to provide reader sufficient knowledge for connecting CSQL server through JDBC and completing the basic database operations.

This document contains a general description of the services provided by JDBC and some code example using JDBC explained.

### 5.1. Overview of JDBC

- Standard Java Interface
- Important Packages
  - java.sql
  - javax.sql
- Important classes
  - java.sql.Date
  - java.lang.DriverManager
  - java.sql.Time
- Important Interfaces
  - java.sql.Connection
  - java.sql.PreparedStatement
  - java.sql.ResultSet
  - java.sql.Statement
  - java.sql.SQLException
  - java.sql.ParameterMethod
  - java.sql.ResultSetMetadata
  - java.sql.SQLException
  - javax.sql.DataSource
- Library
  - libcsqldb.so
- Driver
  - CSqldbDriver.jar

### 5.2. Background

The JDBC API defines Java classes to represent database connections, SQL statements, Result sets, etc. It allows a Java Programmer to issue SQL statements and process the

results. JDBC is the primary API for database access in Java. Because JDBC is a standard specification, a Java program that uses the JDBC API can connect to any database management system (DBMS) for which there is a JDBC Driver.

CSQL JDBC driver is implemented using native methods to bridge to database access libraries.

### 5.3. What is JDBC Driver

The JDBC API defines the Java Interfaces and classes that programmers use to connect to database and send queries. A Java program that uses the JDBC API loads the specified driver for a particular DBMS before it actually connects to a database. The JDBC DriverManager class then sends all JDBC API calls to the loaded driver.

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

### 5.4. Classes, Interfaces and Methods in Details

This section details CSQL's implementation of the following `java.sql` classes, interfaces and methods. Using these standard interfaces and classes, programmers can write applications that connect to databases, send queries written in structured query language (SQL), and process the results.

Majority of JDBC API is located in `java.sql` package. The following are core JDBC `java.sql` Classes, Interfaces and Methods.

#### 5.4.1. Interfaces

Below Interfaces and classes are supported.

<b>Interface</b>	<b>Description</b>
<code>java.sql.Connection</code>	Interface used to establish a connection to a database. SQL statements run within the context of a connection.
<code>java.sql.Driver</code>	Interface used to locate the driver for a particular database management system.
<code>java.sql.PreparedStatement</code>	Interface used to send precompiled SQL statements to the database server and obtain result.

java.sql.ResultSet	Interface used to process the results returned from executing an SQL statement.
java.sql.ResultSetMetaData	Interface used to return information about the columns in a ResultSet object.
java.sql.Statement	Interface used to send static SQL statements to the database server and obtain results.
java.sql.ParameterMetadata	It describes the number, type, and properties of parameters to prepared statements.
javax.sql.DataSource	An alternative to the DriverManager facility, a DataSource object is the preferred means of getting a connection.

#### 5.4.2. Classes

Classes	Description
java.sql.Date	Subclass of java.util.Date used for the SQL DATE data type.
java.lang.DriverManager	Class used to manage a set of JDBC Drivers
java.sql.Time	Subclass of java.util.Date used for the SQL TIME data type
java.sql.TimeStamp	Subclass of java.util.Date used for the SQL TIMESTAMP data type
java.sql.Types	Class used to define constants that are used to identify standard SQL data types, such as CHAR and INTEGER.
java.sql.String	Class used to identify text data types such as CHAR.

#### 5.4.3. Exception Classes

Exception Class	Description
java.sql.SQLException	Exception that provides information about a database error.

## 5.5. Data types

Data Types of CSQL and Java Programming are not same; it needs some mechanism for transferring data between database using CSQL data types and application.

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

The Below table represent the Java mapping for CSQL data types.

<b>CSQL Type</b> (java.sql.types)	<b>Java Type</b>
TINYINT	Byte
SMALLINT	short
INTEGER	Int
LONG	bigint
BIGINT	Long
FLOAT	Double
DOUBLE	
CHAR	Java.lang.String
DATE	Java.sql.Date
TIME	Java.sql.Time
TIMESTAMP	Java.sql.TimeStamp
BINARY	Byte []

## 5.6. Using CSQL JDBC Driver

### 5.6.1. Getting Started

The following steps for getting started with CSQL JDBC Driver.

- Setting the CLASSPATH
- Setting the development environment
- Loading the Driver
- Registering JDBC Driver
- Connecting to the database

## 5.6.2. Setting up the CLASSPATH

After installing CSQL, the JDBC Driver should be found in `csql/lib/CsqlJdbcDriver.jar`. The library `libcsqljdbc.so` is required because CSQL JDBC Driver is implemented using native method to access the database server. Both of these files are available with the CSQL Installation Package.

Below path is for CSQL JDBC Driver.

```
export CLASSPATH=$CSQL_INSTALL_ROOT/lib/CsqlJdbcDriver.jar:.
```

The `setupenv.ksh` file contains the class path for the JDBC Driver and is present in CSQL's root directory and at the time of building the CSQL it is automatically set.

*Refer CSQL User Manual* to know how to install CSQL and Setting up Required JDK.

## 5.6.3. Setting the development environment

CSQL JDBC Driver expects for JDK version 1.5. JDBC interface is included in the `java.sql` Package. Any source that uses JDBC needs to import the `java.sql.package`. The application always need to import this package.

```
import java.sql.* ;
```

## 5.6.4. Loading the driver

The JDBC driver should be loaded before making the connection. The below method will load the driver, and while loading, the driver will automatically register itself with JDBC. After execution of this code, the driver registers itself in the `DriverManager` which handles loading and unloading driver and interfacing connection requests with the appropriate driver.

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

In the above method the driver class name as an argument and once the driver is loaded it creates an instance of itself.

**Note :** The `forName( )` method can throw a `ClassNotFoundException` if the driver is not available.

### 5.6.5. Connecting to the database

Once the driver is successfully registered with the driver manager a connection is established by creating a Java Connection object with the following code.

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

The argument “jdbc : csql “ is the URL (Uniform Resource Locator ) which represents Database. Another two arguments “root” and “manager” corresponds to user name and password of the database. The getConnection method returns a new connection object to the database.

The CSQL database maintains a list of JDBC URLs for different connections for different purposes.

Below are the URLs and their uses in brief:

- jdbc:csql  
Connect to database without network.
- jdbc:adapter  
Connect to database through adapter without network for any operation in target database.
- jdbc:gateway  
Connect to database through gateway without network
- jdbc:csql://<Host Name>:<Port>/csql  
Connect to database with network
- jdbc:adapter://<Host Name>:<port>/csql  
Connect to database through adapter with network for any operation on target database.
- jdbc:gateway://<Host Name>:<port>/csql  
Connect to database through gateway with network

### 5.6.6. Closing the Connection

To close the connection, call the close method of the Connection. This method is belongs to Connection interface.

```
con.close( );
```

## 5.7. Running SQL Statements with JDBC

This section outlines some important methods for SQL operations which execute in the database.

### 5.7.1. CREATE and DROP

Create statement object using,

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

This cStmt object is used to send and execute SQL Statements to a database.

#### Create Table:

After creating the statement object, we can create a table in database using `execute()` function.

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

The `execute ( )` method is for execution of SQL statements in the database. The return type of this Statement object is Boolean value. After execution of this statement the 'T1' table will be created in database.

#### Drop Table

Using the same statement object we can drop the table from the database.

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

### 5.7.2. INSERT

#### Simple Statement

Following code expects that a Connection object con is established before calling the code.

```
cStmt.execute("INSERT INTO T1 ( f1, f2 ) VALUES ( 1, 'ABC' ) ;" ) ;
```

Note that the insert is not committed by the code unless the database is in autocommit mode .

### Statement with parameters

The code below creates a `PreparedStatement` object for a query, assigns values for its parameters and executes the query. In order to bind the input parameter, `PreparedStatement` provides `setInt( )` and `setString( )` method to set the IN parameters.

After bind the parameters value the SQL statement is executed by the `executeUpdate( )` method. It returns the number of records affected by the insert.

The below code expects a `Connection` object `con` to be established.

```
PreparedStatement stmt = null ;
int count = 0 ;

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

for ( int i=0 ; i<10 ; i++ )
{
    stmt . setInt ( 1, i ) ;
    stmt.setString ( 2, String . valueOf ( i + 100 ) ) ;
    stmt . executeUpdate( ) ;
    count ++ ;
}
```

After executing the above code, 10 records will be inserted into the table 'T1' . Note that the insert is not committed by the code unless the database is in autocommit mode.

### 5.7.3. SELECT (Using the ResultSet Interface )

`ResultSet` Interface.

- After traversal, the `ResultSet` must be closed by calling `close( )`.
- `ResultSet` is currently read only.

The below code loops through the result set and prints the data in each column in each row by using `getString` and `getInt` method. Before that the query needs to be executed using `executeQuery( )` method.

The below code expects a `Connection` object `con` to be established.

```

preparedStatement    selStmt = null ;
ResultSet    rs = null ;
int count =0;

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

for ( int  i = 0; i <10 ; i++ )
{
    selStmt . setInt( 1, i ) ;
    rs = selStmt . executeQuery( ) ;

    while ( rs . next( ) )
    {
        System.out.println("Value Is " + rs.getInt(1)+ " "+
rs.getString(2));
        count ++ ;
    }
    rs.close( ) ;
}

```

**NOTE :** CSQL supports only read only forward cursors.

#### 5.7.4. UPDATE

To update records we use the `executeUpdate( )` method. This method is similar to the `executeQuery( )` used to issue a select, however it does not return a `ResultSet`, instead it returns the number of records affected by the update

The below code will issue a update with where clause (Parameterized value) and assumes the connection object is to be established.

```

PreparedStatement    stmt = null;
int count = 0;

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();
    count++;
}

```

### 5.7.5. DELETE

To delete data performing INSERT statement you can use the `executeUpdate( )` method. This method returns the number of records affected by the DELETE statement.

Below code is for delete statement and assumes that connection object is to be established. *Refer a sample example in the Section 5.10*

```
PreparedStatement stmt = null ;
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++;
}
```

### 5.8. Transaction and AutoCommit mode

When a Transaction is created using JDBC, by default it is in auto-commit mode. This means that each SQL statement is treated as a transaction and will be automatically committed immediately after it is executed. Sometimes, you want a group a statements to execute together or fail together. Transactions are used to group a set of statements so that they all execute successfully, or all fail. The way to allow two or more statements to be grouped into a transaction is to disable auto-commit mode. The following line of code will do this.

```
con.setAutoCommit(false) ;
```

#### Transaction Methods

- `setAutoCommit(boolean flag )`  
-If set true, every executed statement is committed immediately.
- `commit()`  
-Relevant only if `setAutoCommit(false)`

- `rollback()`  
-Relevant only if `setAutoCommit(false)`

The following lines of code will for disable auto commit mode and assumes the connection object is to be established.

```
con . setAutoCommit( false ) ;
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();
```

**Note :** The autoCommit state can be monitored by `connection. getAutoCommit()` function.

## 5.9. Code Examples

```
/*
 * Sample JDBC application
 *
 * This Simple JDBC application does the
 * following using CSQL JDBC Driver.
 *
 * 1. Loading the Driver
 * 2. Connects to CSQL using Driver
 * 3. Created table T1(f1 int,f2 char(20)
 * 4. Performing INSERT,UPDATE,DELETE statement
 * using parameterized value
 *
 * 5. Create SELECT query.
 * 6. Fetches and dumps all the rows.
 * 7. Drop the T1 table
 * 8. Closes the connection
```

```

*
*
*   To build and run the application
*
*
*   1.  Make sure you have a working JDK.
*   2.  Install and start CSQL to connect.
*
*   3.  Build and run the application
*       ensure that the server is up.
*
*   For more information follow the link
*       csql/example/jdbc/jdbcexample.java
*
*/

import java.sql.*;
public class jdbcexample {
public static void main(String[] args) {

try {

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

con.setAutoCommit();

Statement cStmt = con.createStatement();
cStmt.execute("CREATE TABLE T1 (f1 integer, f2 char
(20));");

System.out.println("Table t1 created");
cStmt.execute("CREATE INDEX IDX ON T1 ( f1);");
System.out.println("Index created on T1 (f1) ");
cStmt.close();
con.commit();

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.out.println("Total Rows inserted " + count);

    count =0;
    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();
    System.out.println("Total Rows updated ");

    count =0;
    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();

    System.out.println("Total Rows deleted " + count);
    count =0;
    selStmt = con.prepareStatement("SELECT *
    from T1 where f1 = ?;");

```

```

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();
System.out.println("Total Rows selected " + count);

cStmt.execute("DROP TABLE T1;");
System.out.println("Dropped table T1");
cStmt.close();
con.close();

    }catch(Exception e){
        System.out.println("Exception in Test:    "+e);
        e.printStackTrace();
    }

}
}

```

### Java Code Example output

```

Table t1 created
Index created on T1 (f1)
Total Rows inserted 10
Total Rows updated 5
Total Rows deleted 4
Tuple value is 1 101
Tuple value is 2 202
Tuple value is 4 204
Tuple value is 5 105
Tuple value is 7 107
Tuple value is 8 208
Total Rows selected 6
Dropped table T1

```

## 5.10. Netbeans IDE Configuration for JDBC Driver

CSQL JDBC driver supports connection through driver manager, data sources object, connection pooling data source object. For Netbeans IDE, you need to configure properly for CSQL database.

### 5.10.1. Connection Through DriverManager

For the connection through DriverManager object,

- You need to start the CSQL server in one terminal.
- In another terminal run, `./setupenv.ksh` in csql root directory ,
- Find the Netbeans `/bin` directory and run `./netbeans`.
- After executing this script Netbeans editor will open up for you.

Now the next step is to create a java application using the below code.

```
public class Main{
public static void main(String[] args) {

try {
Class.forName("csql.jdbc.JdbcSqlDriver");

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

if(con!=null)
{
    System.out.println("Connection Exstablished");
}
else
{
    System.out.println("Connection failed");
}

Statement cStmt = con.createStatement();
cStmt.execute("CREATE TABLE T1(f1 integer,f2 char
(20));");

System.out.println("Table T1 is created");
cStmt.close();
con.close();

}catch(Exception e){
    System.out.println("Exception in Test: "+e);
}
```

```
        e.printStackTrace();
    }
}
```

#### 5.10.1.1. CLASSPATH setting and Run the application:

After writing the above java code, find the below points,

- Find the path projects ->libraries, and right click on it,
- Add JAR/Folders... It will show you a dialog box.

In the dialog box specify your jar file path and click on OK button. Now run applications.

#### 5.10.2. Connection Through DataSource

For DataSource configuration, you will be setting it up as mentioned above with a web application. For example use the following JSP code.

```
<html>
  <head>
    <meta http-equiv="Content-Type"
content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <%
      try
      {
out.println("Table created on csql ");
javax.naming.Context cxc= new
javax.naming.InitialContext();

javax.sql.DataSource ds = (javax.sql.DataSource)
cxc.lookup("jdbc/bijaya");

java.sql.Connection conn=ds.getConnection("root",
"manager");

out.println("Table created on csql ");

java.sql.Statement stmt=conn.createStatement();
```

```

stmt.execute("CREATE TABLE papu (f1 int,f2 int);");

out.println("Table created on csql ");
conn.close();

    }catch (java.sql.SQLException e)
    {
        out.println("An error occurred.");
    }
    %>
    </body>
</html>

```

### 5.10.2.1. CLASSPATH Setting and Run the application

Find the below points for CLASSPATH setting and to run the application. Now Start application server, Follows the below points to do this,

- Find the 'application sever admin console' for connection pool setup.
- Find the application sever, click on JVM settings, from their in Path Settings
- Set Classpath Prefix and Native Library Path Suffix
- Find the Resources->JDBC->Connection Pools path,
- Create new connection pool with additional properties with URL from the CSQL's available jdbc urls. *Refer the section 5.6.5. for available URLs.*
- User, Password properties and Datasource Classname as csqldb.jdbc.JdbcSqlDataSource.
- Save and ping for successful connection.
- Now you would be creating a JDBC Resources name.
- Run the web application.