Mini SQL
A Lightweight Database Server

Version 1.1

HTML version of the manual produced by Thomas R. Kimpton

Table of Contents

Introduction and History

Mini SQL, or mSQL, is a lightweight database engine designed to provide fast access to stored data with low memory requirements. As its name implies, mSQL offers a subset of SQL as its query interface. Although it only supports a subset of SQL (no views, sub-queries etc.), everything it supports is in accordance with the ANSI SQL specification. The mSQL package includes the database engine, a terminal "monitor" program, a database administration program, a schema viewer, and a C language API. The API and the database engine have been designed to work in a client/server environment over a TCP/IP network.

The decision to write yet another database package was made due to the hole in the range of "free" or "freely available" databases. At the time of writing, there are no other database packages available that support SQL as the query language. The most notable database package for research work, Postgres from the University of California at Berkeley, offers a superset of the original Ingres QUEL known as PostQUEL as its query language.

mSQL has been developed as the database backend for the Minerva Network Management Environment. Originally, Minerva utilised Postgres as its database and generated PostQUEL queries to access it. During initial alpha testing of Minerva, a comment was made that if Minerva generated SQL queries, sites with an existing database installation, such as Ingres or Oracle, could use their commercial databases rather than have to support Postgres as well. To accommodate that wish, mSQL was written initially as an SQL to PostQUEL translator so that sites without commercial database could still use Postgres (seeing as there were no "free" SQL engines available).

As time passed, and Minerva developed further, it became apparent that Postgres was too resource hungry to support the evolving mechanisms provided by Minerva. To gain speed, Minerva was extended to perform monitoring and data acquisition in parallel. Unfortunately, each process that communicated with the database forced another copy of the Postgres backend to be spawned. The fact that each Postgres backend consumes close to 1.5 megabytes of memory soon put a stop to the parallel data acquisition operations.

Although Postgres is a very large and capable package, it is supported on only a handful of platforms. This proved to be a problem as a couple of the original Minerva alpha testers wished to run Minerva on Silicon Graphics machines. Unfortunately, Postgres did not support the SGI machines so they could not participate in the testing. The fact that Minerva itself utilised only a fraction of the features of Postgres and needed to be portable to most platforms proved that tying Minerva to Postgres was not the best option. From that decision Mini SQL was developed.

It should be noted that Postgres is an excellent database package offering a vast array of powerful features and that the above comments in no way try to detract from its success. The fact that Minerva utilises very few database features (it doesn’t even need a relational join) showed that a database as capable and advanced as Postgres was overkill.

Mini SQL Specification

The mSQL language offers a significant subset of the features provided by ANSI SQL. It allows a program or user to store, manipulate and retrieve data in table structures. It does not support relational capabilities such as table joins, views or nested queries. Although it does not support all the relational operations defined in the ANSI specification, it does provide the capability of "joins" between multiple tables.

Although the definitions and examples below depict mSQL key words in upper case, no such restriction is placed on the actual queries.

The Create Clause

The create clause as supported by mSQL can only be used to create a table. It cannot be used to create other definitions such as views. It should also be noted that there can only be one primary key field defined for a table. Defining a field as a key generates and implicit "not null" attribute for the field.

```
CREATE TABLE table_name (
        col_name                col_type                [ not null | primary
        [ , col_name            col_type                [ not null | primary
)
```

for example

```
CREATE TABLE emp_details(
        first_name              char(15) not null,
        last_name               char(15) not null,
        dept            char(20),
        emp_id          int primary key,
        salary          int
)
```

The available types are:-

- char (len)
- int
- real

The Drop Clause

Drop is used to remove a table definition from the database:

```
DROP TABLE table_name
```

for example

```
DROP TABLE emp_details
```

The Insert Clause

Unlike ANSI SQL, you cannot nest a select within an insert (i.e. you cannot insert the data returned by a select). Currently, you must also specify the names of the fields into which the data is to be inserted. You cannot specify the values without the field name and expect the server to insert the data into the correct fields by default.

```
INSERT INTO table_name ( column [ , column ]** )
VALUES (value [, value]** )
```

for example

```
INSERT INTO emp_details ( first_name, last_name, dept, salary)
VALUES ('David', 'Hughes', 'I.T.S.','12345')
```

The number of values supplied must match the number of columns.

The Delete Clause

The syntax for mSQL's delete clause is

```
DELETE FROM table_name
WHERE column OPERATOR value
[ AND | OR column OPERATOR value ]**

OPERATOR can be <, >, =, <=, >=, <>, or like
```

for example

```
        DELETE FROM emp_details WHERE emp_id = 12345
```

The Select Clause

The select offered by mSQL lacks some of the features provided by the SQL spec:

- No nested selects
- No implicit functions (e.g. count(), avg())

It does however support:

- Joins
- DISTINCT row selection
- ORDER BY clauses
- Regular expression matching
- Column to Column comparisons in WHERE clauses

So, the formal syntax for mSQL's select is:-

```
        SELECT [table.]column [ , [table.]column ]**
        FROM table [ , table]**
        [ WHERE [table.] column OPERATOR VALUE
                [ AND | OR [table.]column OPERATOR VALUE]** ]
        [ ORDER BY [table.]column [DESC] [, [table.]column [DESC] ]

        OPERATOR can be <, >, =, <=, >=, <>, or like
        VALUE can be a literal value or a column name
```

A simple select may be

```
        SELECT first_name, last_name FROM emp_details
        WHERE dept = 'finance'
```

To sort the returned data in ascending order by last_name and descending order by first_name the query would look like this

```
        SELECT first_name, last_name FROM emp_details
        WHERE dept = 'finance'
        ORDER BY last_name, first_name DESC
```

And to remove any duplicate rows, the DISTINCT operator could be used:

```
        SELECT DISTINCT first_name, last_name FROM emp_details
```

```
WHERE dept = 'finance'
ORDER BY last_name, first_name DESC
```

The regular expression syntax supported by LIKE clauses is that of standard SQL:

- '_' matches any single character
- '%' matches 0 or more characters of any value
- '\' escapes special characters (e.g. '\%' matches % and '\\' matches \ )
- all other characters match themselves

So, to search for anyone in finance who's last name consists of a letter followed by 'ughes', such as Hughes, the query could look like this:

```
SELECT first_name, last_name FROM emp_details
WHERE dept = 'finance' and last_name like '_ughes'
```

The power of a relational query language starts to become apparent when you start joining tables together during a select. Lets say you had two tables defined, one containing staff details and another listing the projects being worked on by each staff member, and each staff member has been assigned an employee number that is unique to that person. You could generate a sorted list of who was working on what project with a query like:

```
SELECT  emp_details.first_name, emp_details.last_name,
                project_details.project
FROM emp_details, project_details
WHERE emp_details.emp_id = project_details.emp_id
ORDER BY emp_details.last_name, emp_details.first_name
```

mSQL places no restriction on the number of tables "joined" during a query so if there were 15 tables all containing information related to an employee ID in some manner, data from each of those tables could be extracted, albeit slowly, by a single query. One key point to note regarding joins is that you must qualify all column names with a table name. mSQL does not support the concept of uniquely named columns spanning multiple tables so you are forced to qualify every column name as soon as you access more than one table in a single select.

The Update Clause

The mSQL update clause cannot use a column name as a value. Only literal values may by used as an update value

```
UPDATE table_name SET column=value [ , column=value ]**
WHERE column OPERATOR value
        [ AND | OR column OPERATOR value ]**

OPERATOR can be <, >, =, <=, >=, <>, or like
```

for example

```
UPDATE emp_details SET salary=30000 WHERE emp_id = 1234
```

The Database Engine

The mSQL daemon, msqld, is a standalone application that listens for connections on a well known TCP socket. It is a single process engine that will accept multiple connections and serialise the queries received. It utilises memory mapped I/O and cache techniques to offer rapid access to the data stored in a database. It also utilises a stack based mechanism that ensures that INSERT operations are performed at the same speed regardless of the size of the table being accessed. Preliminary testing performed by a regular user of mSQL has shown that for simple queries, the performance of mSQL is comparable to or better than other freely available database packages. For example, on a set of sample queries including simple inserts, updates and selects, mSQL performed roughly 4 times faster than University Ingres and over 20 times faster than Postgres on an Intel 486 class machine running Linux.

The server may be accessed either via a well known TCP socket or via a UNIX domain socket with the file system (/dev/msqld). The availability of the TCP socket allows client software to access data stored on machine over the network. Use of the TCP socket should be limited to client software on remote machines as communicating with the server via a TCP socket rather than the UNIX socket will result in a substantial drop in performance. See the details on the programming API and also the command line options to standard programs for details on selecting the server machine.

The engine includes debugging code so that its progress can be monitored. There are currently 8 debugging modules available in the engine. Debugging for any of the available modules can be enabled at runtime by setting the contents of the MINERVA_DEBUG environment variable to acolon separated list of debug module names. A list of available debug modules is given below:

- cache Display the workings of the table cache
- query Display each query before it is executed
- error Display error message as well as sending them to the client
- key Display details of key based data lookups
- malloc Display details of memory allocation
- trace Display a function call trace as the program executes
- mmap Display details of memory mapped regions
- general Anything that didn’t fit into a category above

For example, to make the server display the queries before they are processed and also show details of the memory allocation that takes place during the query execution, the following value would be set

```
setenv MINERVA_DEBUG query:malloc
```

By default, the software is installed into /usr/local/Minerva and the server will use space within that directory for the storage of the databases and also temporary result tables during operations such as joins and ordering.

C Programming API

Included in the distribution is the mSQL API library, libmsql.a. The API allows any C program to communicate with the database engine. The API functions are accessed by including the msql.h header file into your program and by linking against the mSQL library (using -lmsql as an argument to your C compiler). The library and header file will be installed by default into /usr/local/Minerva/lib and /usr/local/Minerva/include respectively.

Like the mSQL engine, the API supports debugging via the MINERVA_DEBUG environment variable. Three debugging modules are currently supported by the API: query, api, and malloc. Enabling "query" debugging will cause the API to print the contents of queries as they are sent to the server. The "api" debug modules causes internal information, such as connection details, to be printed. Details about the memory used by the API library can be obtained via the "malloc" debug module. Information such as the location and size of malloced blocks and the addresses passed to free() will be generated. Multiple debug modules can be enabled by setting MINERVA_DEBUG to a colon separated list of module names. For example

```
        setenv MINERVA_DEBUG api:query
```

msqlConnect()

```
int msqlConnect(char * host)
```

msqlConnect() forms an interconnection with the mSQL engine. It takes as its only argument the name or IP address of the host running the mSQL server. If NULL is specified as the host argument, a connection is made to a server running on the localhost using the UNIX domain socket /dev/msqld. If an error occurs, a value of -1 is returned and the external variable msqlErrMsg will contain an appropriate text message. This variable is defined in "msql.h".

If the connection is made to the server, an integer identifier is returned to the calling function. This values is used as a handle for all other calls to the mSQL API. The value returned is in fact the socket descriptor for the connection. By calling msqlConnect() more than once and assigning the returned values to separate variables, connections to multiple database servers can be maintained simultaneously.

In previous versions of mSQL, the MSQL_HOST environment variable could be used to specify a target machine if the host parameter was NULL. This is no longer the case.

msqlSelectDB()

```
int msqlSelectDB(sock,dbName)
        int     sock;
        char    *dbName;
```

Prior to submitting any queries, a database must be selected. msqlSelectDB() instructs the engine which database is to be accessed. msqlSelectDB() is called with the socket descriptor returned by msqlConnect() and the name of the desired database. A return value of -1 indicates an error with msqlErrMsg set to a text string representing the error. msqlSelectDB() may be called multiple times during a program’s execution. Each time it is called, the server will use thespecified database for future accesses. By calling msqlSelectDB() multiple times, a program can switch between different databases during its execution.

msqlQuery()

```
int msqlQuery(sock, query)
        int     sock;
        char    *query;
```

Queries are sent to the engine over the connection associated with sock as plain text strings using msqlQuery(). As usual, a returned value of -1 indicates an error and msqlErrMsg will be updated. If the query generates output from the engine, such as a SELECT statement, the data is buffered in the API waiting for the application to retrieve it. If the application submits another query before it retrieves the data using msqlStoreResult(), the buffer will be overwritten by any data generated by the new query.

msqlStoreResult()

```
m_result *msqlStoreResult()
```

Data returned by a SELECT query must be stored before another query is submitted or it will be removed from the internal API buffers. Data is stored using the msqlStoreResult() function which returns a result handle to the calling routines. The result handle is a pointer to a m_result structure and is passed to other API routines when access to the data is required. Once the result handle is allocated, other queries may be submitted. A program may have many result handles active simultaneously.

msqlFreeResult()

```
void msqlFreeResult(result)
        m_result                *result;
```

When a program no longer requires the data associated with a particular query result, the data must be freed using msqlFreeResult(). The result handle associated with the data, as returned by msqlStoreResult() is passed to msqlFreeResult() to identify the data set to be freed.

msqlFetchRow()

```
m_row msqlFetchRow(result)
        m_result                *result;
```

The individual database rows returned by a select are accessed via the msqlFetchRow() function. The data is returned in a variable of type m_row which contains a char pointer for each field in the row. For example, if a select statement selected 3 fields from each row returned, the value of the 3 fields would be assigned to elements [0], [1], and [2] of the variable returned by msqlFetchRow(). A value of NULL is returned when the end of the data has been reached. See the example at the end of this sections for further details.

msqlDataSeek()

```
void msqlDataSeek(result, pos)
        m_result                *result;
        in              pos;
```

The m_result structure contains a client side "cursor" that holds information about the next row of data to be returned to the calling program. msqlDataSeek() can be used to move the position of the data cursor. If it is called with a position of 0, the next call to msqlFetchRow() will return the first row of data returned by the server. The value of pos can be anywhere from 0 (the first row) and the number of rows in the table. If a seek is made past the end of the table, the next call to msqlFetchRow() will return a NULL.

msqlNumRows()

```
int msqlNumRows(result)
        m_result                *result;
```

The number of rows returned by a query can be found by calling msqlNumRows() and passing it the result handle returned by msqlStoreResult(). The number of rows of data sent as a result of the query is returned as an integer value. If a select query didn't match any data, msqlNumRows() will indicate that the result table has 0 rows (note: earlier versions of mSQL returned a NULL result handle if no data was found. This has been simplified and made more intuitive by returning a result handle with 0 rows of result data)

msqlFetchField()

```
m_field *msqlFetchField(result)
        m_result                *result;
```

Along with the actual data rows, the server returns information about the data fields selected. This information is made available to the calling program via the msqlFetchField() function. Like msqlFetchRow(), this function returns one element of information at a time and returns NULL when no further information is available. The data is returned in a m_field structure which contains the following information:-

```
typedef struct {
        char    *name,          /* name of field */
                *table;         /* name of table */
        int     type,           /* data type of field */
                length,         /* length in bytes of field */
                flags;          /* attribute flags */
} m_field;
```

Possible values for the type field are defined in msql.h as INT_TYPE, CHAR_TYPE and REAL_TYPE. The individual attribute flags can be accessed using the following macros:-

```
IS_PRI_KEY(flags)                               /* Field is the primary key */
IS_NOT_NULL(flags)                              /* Field may not contain a NULL valu
```

msqlFieldSeek()

```
void msqlFieldSeek(result, pos)
        m_result                *result;
        int             pos;
```

The result structure includes a "cursor" for the field data. It's position can be moved using the msqlFieldSeek() function. See msqlDataSeek() for further details.

msqlNumFields()

```
int msqlNumFields(result)
        m_result                *result;
```

The number of fields returned by a query can be ascertained by calling msqlNumFields() and passing it the result handle. The value returned by msqlNumFields() indicates the number of elements in the data vector returned by msqlFetchRow(). It is wise to check the number of fields returned before, as with all arrays, accessing an element that is beyond the end of the data vector can result in a segmentation fault.

msqlListDBs()

```
m_result *msqlListDBs(sock)
        int             sock;;
```

A list of the databases known to the mSQL engine can be obtained via the msqlListDBs() function. A result handle is returned to the calling program that can be used to access the actual database names. The individual names are accessed by calling msqlFetchRow() passing it the result handle. The m_row data structure returned by each call will contain one field being the name of one of the available databases. As with all functions that return a result handle, the data associated with the result must be freed when it is no longer required using msqlFreeResult().

msqlListTables()

```
m_result *msqlListTables(sock)
        int     sock;;
```

Once a database has been selected using msqlInitDB(), a list of the tables defined in that database can be retrieved using msqlListTables(). As with msqlListDBs(), a result handle is returned to the calling program and the names of the tables are contained in data rows where element [0] of the row is the name of one table in the current database. The result handle must be freed when it is no longer needed by calling msqlFreeResult().

msqlListFields()

```
m_result *msqlListFields(sock,tableName);
        int     sock;
        char    *tableName
```

Information about the fields in a particular table can be obtained using msqlListFields(). The function is called with the name of a table in the current database as selected using msqlSelectDB() and a result handle is returned to the caller. Unlike msqlListDBs() and msqlListTables(), the field information is contained in field structures rather than data rows. It is accessed using msqlFetchField(). The result handle must be freed when it is no longer needed by calling msqlFreeResult().

msqlClose()

```
int msqlClose(sock)
        int     sock;
```

The connection to the mSQL engine can be closed using msqlClose(). The function must be called with the connection socket returned by msqlConnect() when the initial connection was made.

The mSQL Terminal Monitor

Like all database applications, mSQL provides a program that allows a user to interactively submit queries to the database engine. In the case of mSQL, it is a program simply called ‘msql’. It requires one command line argument, being the name of the database to access. Once started, there is no way to swap databases without restarting the program.

The monitor also accepts two command line flags as outlined below:

- -h Host Connect to the mSQL server on Host.
- -q Process one query and quit returning an exit code.

The monitor has been modelled after the original Ingres (and the subsequent Postgres) monitor program. Commands are distinguished from queries due to their being prefixed with a backlashes. To obtain help from the monitor prompt, the \h command is used. To exit from the program, the \q command or an EOF (^D) must be entered.

To send a query to the engine, the query is entered followed by the \g command. \g tells the monitor to "Go" and send the query to the engine. If you wish to edit your last query, \e will place you inside vi so that you can modify your query. If you wish to use an editor other than vi to perform query editing, msql will honour the convention of using the contents of the VISUAL environment variable as an alternate editor. When you have completed your editing, exiting the editor in the usual manner will return you to msql with the edited query placed in the buffer. The query can then be submitted to the server by using the \g "Go" command as usual.

The query buffer is maintained between queries to no only enable query editing, but to also allow a query to be submitted multiple times. If \g is entered without entering a new query, the last query to be submitted will be resubmitted. The contents of the query buffer can also be displayed by using the \p "Print" command of the monitor.

To enable convenient access to database servers running on remote hosts, the mSQL terminal monitor supports the use of an environment variable to indicate the machine running the server (rather than having to specify "-h some.host.name" everytime you execute mSQL). Note that this is a function provided by the mSQL terminal monitor NOT the mSQL API library and as such is not available for use with other programs. To use this feature set the environment variable MSQL_HOST to the name or address of the desired machine.

mSQL Database Administration

mSQL databases are administered using the msqladmin command. Several administrative tasks, such as creating new databases and forcing a server shutdown are performed using msqladmin. Like all mSQL programs, msqladmin accepts the '-h Host' command line flag to specify the desired machine. The commands available via msqladmin are:

- create DataBase Create a new database called DataBase
- drop DataBase Delete the entire database called DataBase
- shutdown Tell the server to shut itself down
- reload Tell the server to reload its access control information
- version Display various version information from the server

It should be noted that the server will only accept create, drop, shutdown, and reload commands if they are sent by the root user (as defined at installation time) and are sent from the machine running the server. An attempt to perform any of these commands from a remote client or as a non-root user will result in a "permission denied" error. The only command you can execute over the network or as a non-root user is version.

mSQL Schema Viewer

mSQL provides the relshow command for display the structure of a database. If executed with no arguments, relshow will list the available database. If it is executed with the name of a database, relshow will list the tables that have been defined for that database. If given both a database and table name, relshow will display the structure of the table including the field names, types, and sizes. Like all mSQL programs, relshow honours the ‘-h Host’ command line flag to specify a remote machine as the database server.

mSQL Database Dumper

A program is provided that will dump the contents and structure of a table or entire database in an ASCII form. The program, msqldump, produces output that is suitable to be read by the mSQL terminal monitor as a script file. Using this tool, the contents of a database can be backed-up or moved to a new database. By virtue of the ‘-h Host’ option, the contents of a remote database may be sucked over the net. This can be used as a mechanism for mirroring the contents of an mSQL database onto multiple machines.

msqldump started life as a user contributed program called msqlsave written by Igor Romanenko (igor@frog.kiev.ua). Thanks Igor.

mSQL Access from Script Languages

ESL

Another development that has arisen from the development of Minerva has been the Extensible Scripting Language, ESL (pronounced Easel). ESL is a C styled scripting language that offers automatic memory allocation, strict typing, associative arrays (both in-core and bound the ndbm files), full SNMP support and much, much more. ESL resembles C so closely that any C programmer will be able to code in ESL within a minute or two of scanning the manual. Because both ESL and mSQL have been developed as part of the Minerva project (well, both were developed in the spare bedroom I call an office as part of my Ph.D.), ESL provides full support for the mSQL API. This includes every aspect of the C API as well as the client server mode of operation.

Access to mSQL from other scripting languages is available using user-contributed extensions to the respective languages. Currently, the following languages are supported:

Perl 5

Andreas Koenig <k@franz.ww.TU-Berlin.DE> has developed MsqlPerl, a Perl 5 adapter for mSQL. It was written against the mSQL 0.2 Patch 1 API but should still work with the 1.0 release as the API hasn’t changed (although a couple of semantics have). MsqlPerl is available via ftp from Bond.edu.au in /pub/Minerva/msql/contrib/MsqlPerl-a1.tgz

Python

Anthony Baxter <anthony.baxter@aaii.oz.au> has developed PymSQL, a Python module for mSQL. It was written using the msql 0.2 Patch 2 API but should still work well. PymSQL can be found on Bond.edu.au in /pub/Minerva/msql/contrib/PymSQL.tar.gz

Tcl

Brad Pepers <pepersb@cuug.ab.ca> has developed tcl_msql, a Tcl interface to mSQL. I can't recall which version of the API Brad was using when he wrote tcl_msql (Brad's been hacking on mSQL since the early days). I'm pretty sure that it'll work against release 1.0 and am even more sure that Brad will fix it if it doesn't (that's because he's _such_ a nice guy and Tcl/Tk users can be quire persuasive when they need to be).

Access Control

Access control is managed by the msql.acl file in the installation directory. This file is split into entries for each database to be controlled. If the file doesn't exist or details for a particular database aren't configured, access reverts to global read/write. An example ACL entry is included below:

```
# Sample access control for mSQL
database=test
read=bambi,paulp
write=root
host=*.Bond.edu.au,-student.it.Bond.edu.au
access=local,remote
```

Using this definition, database 'test' can be accessed by both local and remote connections from any host in the Bond.edu.au domain accept for the machine student.it.Bond.edu.au. Read access is only granted to bambi and paulp. Nobody else is allowed to perform selects on the database. Write access is only available to root.

Control is based on the first match found for a given item. So, a line such as "read=-*,bambi" would not do the desired thing (i.e. deny access to everyone other than bambi) because -* will also match bambi. In this case the line would have to be "read=bambi,-*" although the -* is superfluous as that is the default action.

Note that if an entry isn't found for a particular configuration line (such as read) it defaults to a global denial. For example, if there is no "read" line (i.e. there are no read tokens after the data is loaded) nobody will be granted read access. This is in contrast to the action taken if the entire database definition is missing in which case access to everything is granted.

Another thing to note is that a database's entry _must_ be followed by a blank line to signify the end of the entry. There may also be multiple config lines in the one entry (such as "read=bambi,paulp" "read=root"). The data will be loaded as though it was concatenated onto the same "read" line (i.e. "read=bambi,paulp,root").

Wildcards can be used in any configuration entry. A wildcard by itself will match anything whereas a wildcard followed by some text will cause only a partial wildcard (e.g. *.Bond.edu.au matches anything that ends in Bond.edu.au). A wildcard can also be set for the database name. A good practice is to install an entry with database=* as the last entry in the file so that if the database being accessed wasn't covered by any of the other rules a default site policy can be enforced.

The ACL information can be reloaded at runtime using "msqladmin reload". This will parse the file before it sends the reload command to the engine. Only if the file is parsed cleanly is it reloaded. Like most msqladmin commands, it will only be accepted if generated by the root user (or whoever the database was installed as) on the localhost.

Author's Details

Mini SQL was written by:-


```
David J. Hughes
Senior Network Programmer (and Ph.D. lunatic)
Bond University
Australia
```

E-Mail: bambi@Bond.edu.au
HTTP://Bond.edu.au/People/bambi
Fax: +61 75 951456

Archive Location

The mSQL software is available from :-

- Current version of mSQL.
- User contributed code.
- Monthly archive of the mailing list.

Mailing List

I have setup a mailing list for discussing mSQL. To subscribe, send a message to:-
msql-list-request@Bond.edu.au
To send a message to the entire list, address it to:-
msql-list@Bond.edu.au