Mini SQL Version 4

Users Guide and Reference

Last Updated for mSQL 4.0 release (January 2017)

Hughes Technologies

Preface

Intended Audience

This document has been prepared as a manual for the use of the Mini SQL database system. It is not a general purpose tutorial or text for learning every aspect of the Structured Query Languages (SQL). The reader is expected to have at least an introductory knowledge of SQL and the concepts of a relational database system.

The mSQL API section of this document covers the programming interface provided by mSQL. It is described in the native language of the API library, C. It is assumed that the reader has a good understanding of programming in the C language and that s/he is familiar with the basic functionality provided by the standard C library.

Readers interested in using a scripting language with mSQL should look to Ember, our scripting language with mSQL API bindings. Ember is suitable for use as a stand-alone scripting language or for embedding scripting capabilities in other applications. Details can be found at www.Hughes.com.au.

Document Conventions

This manual has been designed to be printed on US Letter paper. While many parts of the world utilise the A4 paper size (Australia included), it is not possible to print A4 formatted documents on US Letter paper without loss of information. However, printing of US Letter formatted documents on A4 will result in a correct representation of the document with somewhat larger margins than normal.

Throughout this manual, parts of the text have been flagged with the symbol that appears in the margin opposite this paragraph. Such pieces of text are viewed as being important. The reader should ensure that paragraphs marked as important are read even if the entire manual section is only being skimmed. Important sections will include information such as areas in which mSQL may deviate from other SQL implementations, or tips on improving the performance of your database applications.

Table of Contents

Introduction

Mini SQL, or mSQL as is it often called, is a light weight relational database management system. It has been designed to provide rapid access to data sets with as little system overhead as possible. The system itself is comprised of a database server and various tools that allow a user or a client application to communicate with the server.

Although mSQL uses the Structured Query Language (SQL) as its query language, it does not provide a complete implementation of the ANSI standard SQL. Several features of SQL that are found in more recent versions of the ANSI standard and in more sophisticated database systems are not provided by mSQL. The incorporation of such features would be in conflict with the basic concept of mSQL (i.e. a Mini database system) and would also increase the load and system requirements needed to run the software.

The philosophy of mSQL has been to provide a database management system capable of rapidly handling simple tasks. It has not been developed for use in critical financial environments (banking applications for example). The software is capable of performing the supported operations with exceptional speed whilst utilising very few system resources. Some database systems require high-end hardware platforms and vast quantities of memory before they can provide rapid access to stored data. mSQL has been designed to provide exceptional data access performance on “small hardware” platforms (such as PC class hardware). Because of these characteristics, mSQL is well suited to the vast majority of data management tasks.

Although the mSQL software distribution is made available over the Internet (and other mechanisms) it is not public domain software or FreeWare. mSQL is a commercial, supported software package developed by Hughes Technologies Pty Ltd in Australia. Use of this software in any commercial environment requires the purchase of a commercial use license from Hughes Technologies. Free licenses are provided to organisations such as Universities, schools and registered charities in an attempt to maintain the ethos of the original Internet. For more information on purchasing a license or determining whether you qualify for a free license, please see the Hughes Technologies Web site at http://Hughes.com.au/.

Development of mSQL and its associated tools is an ongoing project. Current releases of the mSQL package and applications that use mSQL are always available from the Hughes Technologies web site. If you require product support, a new version of the software, or some ideas about using mSQL then please visit our web site.

Installing mSQL

Mini SQL is generally distributed in source code form to enable the widest possible use of the software. The software and the installation tools have been made as portable as possible. In general, the software will automatically configure itself to the capabilities of the operating system on which it is being compiled. Typing four commands can complete the process of compiling and installing mSQL on most UNIX platforms.

For those that are familiar with installing software of this nature, a summary of the installation process is provided in the ***Express Setup*** section of this document.

Getting Ready to Compile

Before the software can be compiled, the contents of the archive file must be extracted. This involves uncompressing the archive file with gunzip and then using the tar utility to extract the file. If, for example, the file containing the mSQL distribution is called msql-x.x.tar.gz then the following commands would extract the files (two methods are outlined)

```
gunzip msql-x.x.tar.gz
tar –xvf msq-x.x.tar

Or
gzcat msql-x.x.tar.gz | tar –xvf -
```

This process would create a new directory called msql-x.x in the current directory. Within that directory you will find the entire mSQL distribution. Along with various other files and directories there will be directories containing the source code (src directory) and the documentation (the doc directory). Although it is tempting to just enter the src directory and type "make" it is not the correct way to compile mSQL and doing so will cause problems.

The configuration process is totally automatic and will determine what system calls, library functions, and header files your operating system provides. To configure the compilation process simply type

```
./setup
```

While the setup utility is executing you will see various pieces of information about your operating system being displayed as it is determined. This output is informative only. The results are automatically placed in files used by mSQL.

Once the automatic configuration is complete you may either compile the software using the default configuration settings or change the configuration settings from their default value. There are two configuration items located in the src/site.mm file in the targets directory that you may consider modifying. The configuration utility will try to determine the best C compiler to use on your system. If you have multiple C compilers (a system compiler and gcc for example) you may wish to modify the CC entry in src/site.mm.

The only other option in src/site.mm that may require modification is the INST_DIR entry. This entry defines the default installation directory. This setting is not only used during installation of the software but also as the directory containing the run-time configuration file. If you intend running mSQL from a directory other than the default /opt/msql directory then modify the INST_DIR entry in src/site.mm to reflect your installation directory.

Compilation and Installation

Once the setup utility has completed you may compile the software by typing

```
make all
```

The compilation process will traverse all the directories of the mSQL distribution and compile the C source code in those directories. Status information is displayed to you as the compilation process proceeds. If the compilation process stops with an error at any stage then please see the Installation Troubleshooting section below. If the compilation has completed properly you will see a message on your screen informing you that you are ready to install mSQL.

Installation of mSQL can also be achieved using a single command, although you may need to have special permissions on your UNIX system (usually root access). By default, mSQL will be installed in a directory called /usr/local/msql3 on your system. If /usr/local is root owned on your system (as it is on most systems) then you will either need root access or you will have to get your system administrator to complete the installation for you. If you are using a non-default installation directory then ensure that you have the required permissions to create the directory you specified. To complete the installation simply type

```
make install
```

Express Setup

Below is a rough outline of the process of compiling, installing and configuring mSQL. It is intended as a guide for those who are familiar with installing software on a UNIX machine. If you are not familiar with any of the steps mentioned below then please read the complete installation guide from the start of this manual section.

Step 1 Unpack the software distribution using gunzip and tar

```
gunzip msql-x.x.tar.gz
tar –xvf  msql-x.x.tar
```

Step 2 Configure the compilation process

```
./setup
```

Step 3 Check the default values of INST_DIR and CC in the src/site.mm file

Step 4 Compile the software

```
make all
```

Step 5 Install the software

```
make install
```

Step 6 Configure the software by editing the msql.conf file in the installation directory

Configuring mSQL

mSQL 3 utilises an external run-time configuration file for definition of all system configuration values. The file is called msql.conf and is located in the installation directory (usually /usr/local/msql3). An application can choose to use a different configuration file by calling the new msqlLoadConfigFile( ) API function. All standard mSQL applications and utilities provide a command line flag, -f ConfFile , that allows you to specify a non-standard configuration file. When an application first calls the mSQL API library, a check is made to see if a configuration file has been loaded via a call to the msqlLoadConfigFile( ) function. If no such call has been made, the API library loads the default config file. Any values that are specified in that file will over-ride the normal operating parameters used by mSQL.   If no configuration file is found (or certain items are not set) then the default values listed below will be used.

Structure of the config file

The configuration file is a plain text file organised into sections. The file can contain blank lines and comments. A comment is a line that begins with the '#' character. Each section of the configuration file has a section header, which is written as the section name enclosed in square brackets (for example [general]).

Configuration values within a section are presented using the config parameter name followed by an equals sign and then the new value. There can only be one entry per line and if an entry is defined multiple times in the one config file the last value defined will be used. If a parameter is not defined in the config file then an internal default value will be used at run-time.

Elements of the General section

The following configuration parameters are available in the general section of the config file. Please note that %I may be used in configuration entries to signify the mSQL installation directory (e.g. /opt/msql).

Parameter	Default Value	Definition
Inst_Dir	/opt/msql	The full path to the installation directory. This is the directory in which all the mSQL files are located (such as the program files, the database files etc).
DB_Dir	%I/msqldb	The full path of the directory in which the server will store the database files. %I/msqldb is expanded to /usr/local/msql3/msqldb by default.
mSQL_User	msql	The user that the mSQL server should run as. If a user other than this user starts the server (e.g. it is started as root from a boot script) it will attempt to change UID so that it runs as the specified user.
Admin_User	root	The user that is allowed to perform privileged operations such as server shutdown, creation of databases etc.
Pid_File	%I/msqld.pid	The full path of a file in which the PID of the running mSQL server process will be stored.
TCP_Port	1114	The TCP port number on which the mSQL server will accept client/server connections over a TCP/IP network. If this value is modified it must be modified on the machine running the client software also.
UNIX_Port	%I/msqld.sock	The full path name of the UNIX domain socket created by the mSQL server for connections from client applications running on the same machine.

Elements of the System section

The following configuration parameters are available in the System section of the configuration file and determine the values of various system level configuration items.

Parameter	Default Value	Definition
Msynch_Timer	30	Defines the interval in seconds at which the memory mapped data regions maintained in the mSQL server process will be synched with the on-disk images. Setting this value to 0 will disable forced synchronisation of the data and rely on the kernel's synch'ing of the mmap regions.
Host_Lookup	True	Determines whether ip address to hostname lookups are required. If set to true, connections by hosts that do not resolve to a hostname will be rejected.
Read_Only	False	Forces the server to operate in read-only mode. Any attempts to modify the database will be rejected (i.e. the only commands accepted are *select* queries).
Remote_Access	False	Allow access to the database server from remote machines over a TCP/IP network
Local_Access	True	Allow access to the database server from applications running on the same physical machine as the server.
Query_Log	False	Generate a log file containing all queries received and processed by the server.
Query_Log_File		If Query_Log is set to True, this item contains the full path of the file in which the log data will be stored. It is often defined as %I/query.log
Update_Log	False	Generate a log file containing all queries received and processed by the server that modify the database (INSERT, DELETE, UPDATE etc). SELECT queries are not logged.
Update_Log_File		If Update_Log is set to True, this item contains the full path of the file in which the log data will be stored. It is often defined as %I/update.log
Force_Munmap	False	Rather than relying on the operating system to synchronise memory mapped files properly, the option will force the server to unmap and remap the regions. If set to true, this will impose significant performance penalties but will ensure data integrity on buggy systems.
Num_Children	2	The number of backend process started by the multi-process (broker based) server implementation
Table_Cache	8	The number of entries that can be held in the table cache. Each entry consumes memory and file descriptor. On embedded systems, this value can be set to a lower value to consume resources. A minimum site of 2 is enforced as the server cannot operate without at least 2 entries.
Sort_Max_Mem	1000	The max amount of memory that an ORDER BY or a DISTINCT operation will consume.

Example configuration file

Below is a sample configuration file. This file just sets the parameters to their default values.

```
#
# msql.conf  -  Configuration file for Mini SQL Version 3
#
# This configuration sets all options to their default values.
# Note : %I is expanded to the value of the Inst_Dir element is included in a value.
#

[general]
Inst_Dir = /opt/msql
mSQL_User = daemon
Admin_User = root
Pid_File = %I/msqld.pid
TCP_Port = 1114
UNIX_Port = %I/msqld.sock

[system]
Msynch_Timer = 30
Host_Lookup = True
Read_Only = False
Num_Children = 3
Table_Cache = 8
Sort_Max_Mem = 1000
Force_Munmap = False
Query_Log = False
Update_Log = False
Local_Access = True
Remote_Access = False
```

The mSQL Query Language

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 some relational capabilities such as views and nested queries. Although it does not support all the relational operations defined in the ANSI specification, mSQL provides a significant subset of the ANSI SQL standard and is capable of supporting the vast majority of applications.

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

The Create Clause

The create clause as supported by mSQL can be used to create tables, indices, and sequences. The three valid constructs of the create clause are shown below:

```
CREATE TABLE table_name (
        col_name col_type [ not null ]
        [ , col_name col_type [ not null ] ]**
)

CREATE [ UNIQUE ] INDEX index_name ON table_name (
        field_name
        [ , field_name ] **
)

CREATE SEQUENCE ON table_name [ STEP step_val ] [ VALUE initial_val ]
```

An example of the creation of a table is shown below:

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

mSQL supports a wide range of data types for storing numeric, textual, or other more specialised data. Numeric data types are available for integer and floating point values. And integer type may be defined as either signed or unsigned, and can range in size from 8 bits (int8) to 64 bits (int64). Special data types are included for supporting efficient storage of network related information such as IP Addresses and Internet Address Blocks (CIDR network definitions). The complete list of available types is provided in the table below.

Type	Description
char (len)	String of characters (or other 8 bit data)
text (len)	Variable length string of characters. The defined length is used to indicate the expected average length of the data. Any data longer than the specified length will be split between the data table and external overflow buffers. **Note** : text fields cannot be used in an index.
int	Signed integer values. Sub types include int8, int16, int32 and int64 for 8, 16, 32 and 64 bit values respectively.
real	Decimal or Scientific Notation real values
uint	Unsigned integer values. Sub types include uint8, uint16, uint32, and uint64 for 8, 16, 32, and 64 bit values.

date	Date values in the format of 'DD-Mon-YYYY' such as '1-Jan-1997'
time	Time values stored in 24 hour notation in the format of 'HH:MM:SS'
datetime	Combined date and time in the format of 'DD-Mon-YYYY HH:MM:SS'
millitime	Time value including milliseconds represented as 'HH:MM:SS.SSSS'
millidatetime	Combined date and time including milliseconds shown as 'DD-Mon-YYYY HH:MM:SS.SSSS'
money	A numeric value with two fixed decimal places
ipv4	An Internet address in the format of 'aaa.bbb.ccc.ddd'
cidr4	An Internet network address block specified in CIDR format 'aaa.bbb.ccc.ddd/length'. When comparing cidr4 values, the standard operators are overridden so that the < operator performs a "less specific" comparison and the > operator performs a "more specific" comparison.

The table structure shown in the example would benefit greatly from the creation of some **indices**. It is assumed that the *emp_id* field would be a unique value that is used to identify an employee. Such a field would normally be defined as a unique index. Similarly, a common query may be to access an employee based on the combination of the first and last names. A compound index (i.e. constructed from more than one field) would improve performance. Naturally, such a compound index may have multiple entries with the same value (if more than one person called John Smith works for the same company) so a non-unique index would be required. We could construct these indices using :

```
CREATE UNIQUE INDEX idx1 ON emp_details (emp_id)
CREATE INDEX idx2 ON emp_details (first_name, last_name)
```

These indices will be used automatically whenever a query is sent to the database engine that uses those fields in its WHERE clause. The user is not required to specify any special values in the query to ensure the indices are used to increase performance.

Sequences provide a mechanism via which a sequence value can be maintained by the mSQL server. Sequences are a numeric value that can be used as serial numbers, staff identifiers, invoice numbers, or any other application that requires a unique numeric value. Having the server maintain the index allows for atomic operations (such as getting the next sequence value) and removes the concerns associated with performing these operations in client applications. A client application would need to send two queries (one to read the current value and one to update the value) which introduces a "race condition" and the potential for the same sequence value to be assigned to multiple items.

A sequence is associated with a table and a table may contain at most one sequence. Once a sequence has been created it can be accessed by SELECTing the _seq system variable from the table in which the sequence is defined. For example

```
CREATE SEQUENCE ON test STEP 1 VALUE 5
SELECT _seq FROM test
```

The above CREATE operation would define a sequence on the table called *test* that had an initial value of 5 and would be incremented each time it is accessed (i.e. have a step of 1). The SELECT statement above would return the value 5. If the SELECT was issued again, a value of 6 would be returned. Each time the _seq field is selected from *test* the current value is returned to the caller and the sequence value itself is incremented.

Using the STEP and VALUE options a sequence can be created that starts at any specified number and is increased or decreased by any specified value. The value of a sequence would decrease by 5 each time it was accessed if it was defined with a step of -5.

The Drop Clause

The Drop clause is used to remove a definition from the database. It is most commonly used to remove a table from a database but can also be used for removing several other constructs. In 3.x it can be used to remove the definition of an index, a sequence, or a table. It should be noted that *dropping* a table or an index removes the data associated with that object as well as the definition. Dropping a table removes any indices or sequences defined for the table.

The drop clause cannot be used to remove an entire database. Dropping a database is achieved by using the msqladmin utility program that is included in the software distribution.

The syntax of the drop clause as well as examples of its use are given below.

```
DROP TABLE table_name
DROP INDEX index_name FROM table_name
DROP SEQUENCE FROM table_name
```

Examples of the use of the drop clause for removing an entire table, an index and a sequence are shown below.

```
DROP TABLE emp_details
DROP INDEX idx1 FROM emp_details
DROP SEQUENCE FROM emp_details
```

The Insert Clause

The insert clause is used to insert or add data to the database. When inserting data you may either specify the fields for which you have provided data (if you are not providing data for every field in the data row) or you may omit the field names if you are providing data for every field. If you do not specify the field names they will be used in the order in which they were defined - you must specify a value for every field if you use this form of the insert clause. If you provide the field names then the number of data values provided must match the number of fields specified.

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

for example

```
INSERT INTO emp_details ( first_name, last_name, dept, salary)
        VALUES ( 'David', 'Hughes', 'Development',12345.00)
INSERT INTO emp_details
        VALUES ('David', 'Hughes', 'Development',12345.00)
```

The Select Clause

The select clause is used to extract data from the database. It allows you to specify the particular fields you wish to retrieve and also a condition to identify the records or rows that are of interest. The ANSI SQL standard defines two features that are not supported by mSQL. The mSQL implementation of the select clause does not support

- Nested selects
- Aggregate functions (e.g. count(), avg())

It does however support:

- Relational joins between multiple tables
- Table aliases
- DISTINCT row selection for returning unique values
- ORDER BY clauses for sorting
- Normal SQL regular expression matching
- Enhanced regular expression matching including case insensitive and soundex
- Column to Column comparisons in WHERE clauses
- Complex conditions
- Value functions (e.g. upper())
- Result tables

The formal definition of the syntax for mSQL's select clause is

```
SELECT [DISTINCT]  [table.]column [ , [table.]column ]**
        [INTO result_table]
        FROM table [ = alias] [ , table [ = alias] ]**
        [ WHERE [table.] column OPERATOR VALUE
        [ AND | OR [table.]column OPERATOR VALUE]** ]
        [ ORDER BY [table.]column [DESC] [, [table.]column [DESC] ]
        [ LIMIT num]  [OFFSET num]

OPERATOR can be <, >, =, <=, =, <>, BETWEEN, LIKE, RLIKE, CLIKE or SLIKE
VALUE can be a literal value or a column name
```

The condition used in the where statement of a select clause may contain '(' ')' to nest conditions or to focus on parts of the conditional evaluation. e.g. "where (age <20 or age >30) and sex = 'male'" .

A simple select that returns the first and last names of anybody employed in the finance department would be

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

To sort the returned data we would add an ORDER BY statement to the select clause. mSQL supports sorting on multiple values in either ascending or descending order for each value. If a direction is not specified it defaults to ascending order. To sort the data from the previous query in ascending order by last_name and descending order by first_name we could use the query below. Note that the two sorting values are separated by a comma and that the first_name field includes the DESC attribute to indicate we sorting is required in descending order.

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

A query such as the one presented above may return multiple of the same value. If for example there were two people named John Smith working in the finance department the name ‘John Smith’ would be returned twice from the query. You may remove any duplicates from the returned data by providing the DISTINCT attribute with the query. An example of using the DISTINCT attribute to remove duplicates from the above query is given below.

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

The amount of data returned by the query can be constrained using the LIMIT clause. By setting a LIMIT value of 10, only the first 10 rows of data will be returned. The OFFSET clause can also be used to specify which rows of the result data are returned to the client. By specifying “OFFSET 15”, the first 15 result rows are skipped. The example query below could be used to generate a limited result set similar to that shown by Internet search engines. In this example, the query will return the 10 result rows starting at the 21st row of matching data (i.e. LIMITed to 10 rows and OFFSETed by 20 rows)

```
SELECT first_name, last_name FROM emp_details
        ORDER BY last_name, first_name
        LIMIT 10  OFFSET 20
```

mSQL supports the use of functions for manipulating the date being retrieved from the database. It doesn't support aggregate functions such as COUNT and AVG. The functions that are provided by mSQL for the manipulation of textual values are shown below.

Function	Example	Description
upper ( )	upper(last_name)	Produces an upper case version of the value
lower( )	lower(last_name)	Produces a lower case version of the value
length( )	length(deptarment)	Produces the number of characters in the value
substr( )	substr(name, 1, 5)	Extracts a substring from the value, starting at character 1 and continuing for 5 characters.
chop( )	chop(name)	Removes the last character from the value
translate( )	translate(name,'a-z ','A-Z_')	Replaces all characters matched by the second arg with the corresponding character from the third arg. The example replaces lowercase characters with uppercase and replaces spaces with underscores.
replace( )	replace(value,'color','colour')	Rewrites 'color' to 'colour' in the value
soundex( )	soundex(value)	Produces the numeric soundex value of the string

The functions provided by Mini SQL for the manipulation of numeric values are listed in the table below.

Function	Example	Description
abs( )	abs(value)	Returns the absolute value of number
ceil( )	ceil(value)	Produces the ceiling of the value (i.e. rounded up)
floor( )	floor(value)	Produces the floor of the value (i.e. rouned down)
mod( )	mod(value)	Returns the modulus of the value
sign( )	sign(value)	Returns –1 for neg, +1 for pos and 0 for NULL values
power( )	power(value, power)	Raises the value to the specified power

mSQL provides four regular expression operators for use in *where* comparisons. These operators may be used to perform "fuzzy" matching on the data if you do not know the exact value for which you are searching. An example of such a search would be if you were looking for any employee with a last_name starting with 'Mc' such as McCormack or McDonald. In such a situation you cannot provide a complete value for the last_name field as you are only interested in part of the value.

The standard SQL syntax provides a very simplistic regular expression capability that does not provide the power nor the flexibility of which UNIX programmers or users will be accustomed. mSQL supports the "standard" SQL regular expression syntax, via the LIKE operator, but also provides further functionality if it is required. The available regular expression operators are:

- LIKE - the standard SQL regular expression operator.
- CLIKE - a standard LIKE operator that ignores case.
- RLIKE - a complete UNIX regular expression operator.
- SLIKE - a ‘soundex’ matching operator (i.e. phonetic matching)

Note : CLIKE, RLIKE, and SLIKE are not features of standard SQL and may not be available in other implementations of the language. If you choose to use them you may have problems porting your application to other database systems. They are, however, very convenient and powerful features of mSQL.

LIKE and CLIKE utilise the regular expression syntax as specified in the ANSI SQL standard. As mentioned above, the ANSI standard regular expression feature provides only a very simplistic implementation of regular expressions. It provides for only single and multiple character wildcards. It does not include enhanced features such as value ranges, value exclusions or value groups. The syntax of the LIKE and CLIKE operators is provided in the following table.

Operator	Description
_	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

Two examples of using the LIKE operator are provided below. In the first we are searching for anyone in the finance department whose last name consists of any letter followed by 'ughes', such as Hughes. The second example shows the query for the ‘Mc’ example mentioned earlier in this section.

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

SELECT first_name, last_name FROM emp_details
        WHERE dept = ‘finance’ and last_name like ‘Mc%’
```

The RLIKE operator provides access to the power of the UNIX standard regular expression syntax. The UNIX regular expression syntax provides far greater functionality than SQL's LIKE syntax. The UNIX regex syntax does not use the '_' or '%' characters in the way SQL's regex does (as outlined above) and provides enhanced functionality such as grouping, value ranges, and value exclusion. The syntax available in the RLIKE operator is shown in the table below. Tutorials for using UNIX regular expression matching are available in the manual pages of any UNIX system (such as the manual pages for grep or ed).

Because RLIKE utilises a complete UNIX regex implementation, the evaluation of a condition containing the RLIKE operator is quite complex. The performance of searches using the RLIKE operator will be slower than those using the LIKE or CLIKE operator. You should only use the RLIKE operator if you cannot achieve your desired matching using the more simplistic LIKE or CLIKE operators.

Operator	Description
.	The dot character matches any single character
^	When used as the first character in a regex, the caret character forces the match to start at the first character of the string
$	When used as the last character in a regex, the dollar sign forces the match to end at the last character of the string
[ ]	By enclosing a group of single characters within square brackets, the regex will match a single character from the group of characters. If the ']' character is one of the characters you wish to match you may specify it as the first character in the group without closing the group (e.g. '[]abc]' would match any single character that was either ']', 'a', 'b', or 'c'). Ranges of characters can be specified within the group using the 'first-last' syntax (e.g. '[a-z0-9]' would match any lower case letter or a digit). If the first character of the group is the '^' character the regex will match any single character that is **not** contained within the group.
*	If any regex element is followed by a '*' it will match **zero or more** instances of the regular expression. To match any string of characters you would use '.*' and to match any string of digits you would use '[0-9]*'

The SLIKE operator provides soundex matching of values (i.e. one value sounds like another value). It does not use an explicit syntax in the same way as the other LIKE operators. You simply provide the word you wish to match. If you wished to search for any name that sounded like 'Hughes', such as 'Hues' you could use SLIKE 'Hughes'.

Relational joining is one of the most powerful features of a relational query language. The concept of "joining" relates to "merging" multiple database tables together and extracting fields from the merged result. As an example, if you had two tables defined, one containing employee details and another containing a list of all current, you may wish to extract a list of the projects that each employee was working on. Rather than duplicating the employee details in the projects table you could simply include the employees staff ID number in the projects table and use a join to extract the first and last names.

The query below is an example of such an operation. The logic behind the query is that we want to extract the first and last names of the employee, plus the name of the project on which the employee is working. We can identify which combinations of the merged table we are looking for as they will have a common value for the employee's staff ID value. Because we are referencing multiple tables in the query, we must include the table name for each field when it is included in the query (e.g. emp_details.first_name rather than just first_name)

```
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
```

It is important to understand the inner workings of a join. If we are joining table A with table B, a merged row will be created for all possible combinations of the rows of both tables. If table A contains only two rows and table B contains 10 rows, then 20 merged rows will be generated and evaluated against the *where* condition. If no *where* condition is specified then all 20 rows will be returned. If this example is extended so that table A contained 1,000 rows and table B contained 2,500 rows then the result would be 1,000 * 2,500 merged rows (that's two and a half million rows!). Whenever a join is used there should normally be a common value in both tables (such as the employee ID in our example) and the condition must include a direct comparison between these two fields to ensure that the result set is limited to only the desired results.

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, and each table included the employee ID field to identify the employee, data from each of those tables could be extracted, by a single query. As mentioned before, a key point to note regarding joins is that you must qualify all field names with a table name. Remember that you must qualify every column name as soon as you access more than one table in a single *select*.

mSQL also supports table aliases so that you can perform a join of a table onto itself. This may appear to be an unusual thing to do but it is a very powerful feature if the rows within a single table relate to each other in some way. An example of such a table could be a list of people including the names of their parents. In such a table there would be multiple rows with a parent/child relationship. Using a table alias you could find out any grandparents contained in the table using the query below. The logic is to find any person who is the parent of someone's parent.

```
SELECT t1.parent, t2.child from parent_data=t1, parent_data=t2
        WHERE t1.child = t2.parent
```

The table aliases t1 and t2 both point to the same table (parent_data in this case) and are treated as two different tables that just happen to contain exactly the same data. Like any other join, the possible result set size is the multiplication of the number of rows in each table. If a table is joined with itself, this equates to N^2 rows where N is the number of rows in the original table. Care must be taken to ensure that the result set is limited by the condition specified otherwise the query can take a very long time to complete and has the potential to fill your disk drive with temporary data as the query is processed.

The Delete Clause

The SQL DELETE clause is used to remove one or more entries from a database table. The selection of rows to be removed from the table is based on the same *where* statement as used by the SELECT clause. In the SELECT clause, the *where* condition is used to identify the rows to be extracted from the database. In the DELETE clause, the *where* condition identifies the rows that are to be deleted from the database. As with all SQL queries, if no where condition is provided, then the query applies to every row in the table and the entire contents of the table will be deleted. The syntax for mSQL's delete clause is shown below

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

OPERATOR can be <, >, =, <=, =, <>, LIKE, RLIKE, CLIKE, or SLIKE
```

An example of deleting a specific employee (identified by the employee ID number of the person) and also deleting every employee within a particular salary range is given below.

```
DELETE FROM emp_details WHERE emp_id = 12345
DELETE FROM emp_details WHERE salary > 20000 and salary < 30000
```

The Update Clause

The SQL update clause is used to modify data that is already in the database. The operation is carried out on one or more rows as specified by the *where* construct. If the condition provided in the where construct matches multiple rows in the database table then each matched row will be updated in the same way. The value of any number of fields in the matched rows can be updated. The syntax supported by mSQL is shown below.

```
UPDATE table_name SET column=value [ , column=value ]**
WHERE column OPERATOR value
[ AND | OR column OPERATOR value ]**
OPERATOR can be <, >, =, <=, =, <>, LIKE, RLIKE, CLIKE or SLIKE
```

For example

```
UPDATE emp_details SET salary=30000 WHERE emp_id = 1234
UPDATE emp_details SET salary=35000, dept='Development' where emp_id = 1234
```

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/ Hughes/lib and /usr/local/msql3/include respectively. An example compilation of a client application (my_app.c in this case) is shown below. The header file and API library are assumed to be in the default installation directory.

```
cc –c –I/usr/local/msql3/include  my_app.c
cc –o my_app  my_app.c  -L/usr/local/msql3/lib  -lmsql
```

Some versions of UNIX, usually those derived from SystemV, do not include the TCP/IP networking functions in the standard C library. The mSQL API library includes calls to the networking functions to facilitate the client/server nature of the API. On machines that do not include the networking functions in the standard C library, the compilation illustrated above would fail due to "unresolved externals" with reference to function names such as socket( ) and gethostbyname( ). If this occurs then the networking code must also be linked with the application. This is usually achieved by adding "-lsocket –lnsl" to the link command as shown below. If you continue to have problems, please consult the "socket" and "gethostbyname" manual pages of your system to determine the libraries you have to include in your link statement.

```
cc –c –I/usr/local/msql3/include  my_app.c
cc –o my_app  my_app.c  -L/usr/local/msql3/lib  -lmsql –lsocket –lnsl
```

Like the mSQL engine, the API supports debugging via the MSQL_DEBUG environment variable. The API currently supports three debugging modules: 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 module 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 MSQL_DEBUG to a colon separated list of module names. For example setenv MSQL_DEBUG api:query

The API has changed slightly from the original mSQL API. Please ensure that you check the semantics and syntax of the functions before you use them.

Query Related Functions

msqlConnect()

```
int  msqlConnect ( host )
        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 value 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. It should also be noted that communicating with the server via UNIX sockets rather than TCP/IP sockets increases performance greatly. If you are communicating with a server on the same machine as the client software you should always specify NULL as the hostname. Using "localhost" or the name of the local machine will force the use of TCP/IP and degrade performance.

msqlSelectDB()

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

Prior to submitting any queries, a database must be chosen. 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 the specified 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 ;
```

A query in SQL terminology is not the same as a query in the English language. In English, the word query relates to asking a question whereas in SQL a query is a valid SQL command. It is a common mistake that people believe that the msqlQuery function can only be used to submit SELECT commands to the database engine. In reality, msqlQuery can be used for any valid mSQL command including SELECT, DELETE, UPDATE etc.

Queries are sent to the engine over the connection associated with sock as plain text strings using msqlQuery(). As with previous releases of mSQL, a returned value of -1 indicates an error and msqlErrMsg will be updated to contain a valid error message. 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.

In previous versions of mSQL, the return value of msqlQuery() was either -1 (indicating an error) or 0 (indicating success). mSQL3.xadds to these semantics by providing more information back to the client application via the return code. If the return code is greater than 0, not only does it imply success, it also indicates the number of rows "touched" by the query (i.e. the number of rows returned by a SELECT, the number of rows modified by an update, or the number of rows removed by a delete).

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. See also msqlFreeResult( ).

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 three 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 section for further details. Note: a NULL value in the database is represented as a NULL pointer in the row.

msqlDataSeek()

```
void msqlDataSeek ( result , pos )
        m_result    * result ;
        int         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 no data is matched by a *select* query, 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 */
        char        * table ;           /* name of table */
        int         type ;              /* data type of field */
        int         length ,            /* length in bytes of field */
        int         flags ;             /* attribute flags */
        } m_field;
```

Possible values for the type field are defined in msql.h. Please consult the header file if you wish to interpret the value of the type or flags field of the m_field structure.

msqlFieldSeek()

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

The result structure includes a "cursor" for the field data. Its 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 because, as with all arrays, accessing an element that is beyond the end of the data vector can result in a segmentation fault.

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. msqlClose() should be called by an application when it no longer requires the connection to the database. If the connection isn’t closed, valuable resources, such as network sockets or file descriptors, can be wasted.

Schema Related Functions

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(). Failure to do so will waste memory.

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(). If msqlListTables is called before a database has been chosen using msqlSelectDB an error will be generated.

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(). As with msqlListTables, msqlListFields will return an error if it is called prior to a database being chosen using the msqlSelectDB function.

msqlListIndex()

```
m_result * msqlListIndex ( sock , tableName , index ) ;
        int         sock ;
        char        * tableName;
        char        * index;
```

The structure of a table index can be obtained from the server using the msqlListIndex() function. The result table returned contains one field. The first row of the result contains the symbolic name of the index mechanism used to store the index. Rows 2 and onwards contain the name of the fields that comprise the index.

An example of the data returned by msqlListIndex is shown below. The example shows the result of calling msqlListIndex on a compound index. The index is defined as an AVL Tree index and is based on the values of the fields first_name and last_name. Currently the only valid index type is ‘avl’ signifying a memory mapped AVL tree index. Further index schemes will be added to mSQL in the future.

```
row[0]
avl
first_name
last_name
```

Date & Time Related Functions

msqlTimeToUnixTime()

```
time_t  msqlTimeToUnixTime( msqltime )
        char        * msqltime;
```

msqlTimeToUnixTime( ) converts an mSQL time value to a standard UNIX time value. The mSQL time value must be a character string in the 24 hour format of “HH:MM:SS” and the returned value will be the number of seconds since 1 Jan 1970 (the normal UNIX format).

msqlUnixTimeToTime()

```
char * msqlUnixTimeToTime( clock )
        time_t      clock;
```

msqlUnixTimetoTime( ) converts a UNIX time value (seconds since the UNIX epoch) into a character string representing the same time in mSQL time format (i.e. “HH:MM:SS” 24 hour format). The returned string is statically declared in the API so you must make a copy of it before you call the function again.

msqlDateToUnixTime()

```
time_t  msqlDateToUnixTime( msqldate )
        char        * msqldate;
```

msqlDateToUnixDate( ) converts an mSQL date format string into a UNIX time value. The mSQL date format is “DD-Mon-YYYY” (for example “12-Jun-1997”) while the returned value will be the number of seconds since the UNIX epoch. The mSQL date routines will assume the 20th century if only 2 digits of the year value are presented. Although the valid range of mSQL dates is 31st Dec 4096bc to the 31st Dec 4096, the UNIX format cannot represent dates prior to the 1st Jan 1970.

msqlUnixTimeToDate()

```
char * msqlUnixTimeToDate( clock )
        time_t    clock;
```

msqlUnixTimeToDate( ) converts a standard UNIX time value to an mSQL date string. The time value is specified as seconds since the UNIX epoch (1st Jan 1970) while the mSQL date string will contain the date formatted as "DD-Mon-YYYY" (e.g. "12-Jun-1997"). A convenient use of this function is to determine the mSQL date value for the current day using the following C code example.

```
clock = time( );
date = msqlUnixTimeToDate( clock );
```

msqlSumTimes()

```
char * msqlSumTimes ( time1, time2 )
        char      * time1,
                  *time2;
```

The msqlSumTimes( ) routine provides a mechanism for performing addition between two mSQL time formatted strings. A literal addition of the values is returned to the calling routine in mSQL time format. As an example, calling msqlSumTimes with the values "1:30:25" and "13:15:40" would return "14:46:05".

msqlDateOffset()

```
char * msqlDateOffset( date, dOff, mOff, yOff )
        char      * date;
        int       dOff,
                  mOff,
                  yOff
```

The msqlDateOffset( ) function allows you to generate an mSQL date string that is a specified period before or after a given date. This routine will determine the correct date based on the varying days of month. It is also aware of leap years and the impact they have on date ranges. The new date is calculated using the specified date and an offset value for the day, month and year. The example below would determine tomorrow's date

```
clock = time( );
today = msqlUnixTimeToDate( clock );
tomorrow = msqlDateOffset( today , 1 , 0 , 0 );
```

msqlDiffTimes()

```
char * msqlDiffTimes( time1, time2 )
        char      * time1,
                  *time2;
```

To determine the time difference between two time values, the msqlDiffTimes( ) function can be used. The two time values must be mSQL time formatted text strings and the returned value is also an mSQL time string. A restriction is placed on the times in that time1 must be less than time2.

msqlDiffDates()

```
int msqlDiffDates( date1, date2 )
        char     * date1,
                 * date2;
```

The msqlDiffDates( ) function can be used to determine the number of days between two dates. Date1 must be less than date2 and the two dates must be valid mSQL date formatted strings. In conjunction with the msqlDiffTimes( ) function it is possible to determine a complete time difference between two pairs of times and dates.

msqlDatetimeToUnixTime()

```
time_t msqlDatetimeToUnixTimel( datetime)
        char     * date1;
```

This function converts a string formatted as an mSQL datetime value into a UNIX time value. The mSQL timedate format is "DD-Mon-YYYY HH:MM:SS" (for example "12-Jun-1997 12:30:59") while the returned value will be the number of seconds since the UNIX epoch. The mSQL date routines will assume the 20th century if only 2 digits of the year value are presented. Although the valid range of mSQL dates is 31st Dec 4096bc to the 31st Dec 4096, the UNIX format cannot represent dates prior to the 1st Jan 1970.

msqlUnixTimeToDatetime()

```
char * msqlUnixTimeToDatetime( clock )
        time_t   clock;
```

msqlUnixTimeToDatetime( ) converts a standard UNIX time value to an mSQL datetime string. The time value is specified as seconds since the UNIX epoch (1st Jan 1970) while the mSQL datetime string will contain the date and time formatted as "DD-Mon-YYYY HH:MM:SS" (e.g. "12-Jun-1997 12:30:59"). A convenient use of this function is to determine the mSQL datetime value for the current moment using the following C code example.

```
clock = time( );
datetime = msqlUnixTimeToDatetime( clock );
```

msqlMillitimeToTimeval( )

```
int msqlMillitimeToUnixTimeval( millitime , tv )
        char     * millitime;
        struct timeval  * tv;
```

This function converts a string formatted as an mSQL millitime value into a timeval structure. The "struct timeval" is a commonly used format for the representation of time values, including fractions of a second, on UNIX platforms. This function takes a string formatted as "HH:MM:SS.SSSS" and a pointer to a timeval structure. The tv_sec and tv_usec fields of the timeval structure are filled. Errors are reported by a return value of -1.

msqlTimevalToMillitime()

```
char * msqlTimevalToMillitime( tv )
        struct timeval  *tv;
```

msqlTimevalToMillitime( ) converts a UNIX timeval structure to an mSQL millitime string. The timeval structure must contain the desired time represented in seconds (in the tv_sec field) and microsecond (in the tv_usec field). The values are converted to an string formatted as "HH:MM:SS.SSSS" including milliseconds (not microseconds).

msqlMillidatetimeToTimeval( )

```
int msqlMillidatetimeToUnixTimeval( millidatetime, tv )
        char           * millidatetime;
        struct timeval  * tv;
```

This function converts a string formatted as an mSQL millidatetime value into a timeval structure. The "struct timeval" is a commonly used format for the representation of time values, including fractions of a second, on UNIX platforms. This function takes a string formatted as "DD-Mon-YYY HH:MM:SS.SSSS" and a pointer to a timeval structure. The tv_sec and tv_usec fields of the timeval structure are filled. Errors are reported by a return value of -1. The range of dates supported by this function is limited by the underlying operating system. The mSQL millidatetime datatype supports a far greater range of dates than can be represented using a UNIX timeval structure.

msqlTimevalToMillidatetime()

```
char * msqlTimevalToMillidatetime( tv )
        struct timeval  *tv;
```

msqlTimevalToMillidatetime( ) converts a UNIX timeval structure to an mSQL millidatetime string. The timeval structure must contain the desired time represented in seconds since the UNIX epoch (in the tv_sec field) and microsecond (in the tv_usec field). The values are converted to an string formatted as "DD-Mon-YYYY HH:MM:SS.SSSS" including milliseconds (not microseconds). Note that the UNIX operating system imposed restrictions on the range of dates that can be represented using the timeval structure. This function is restricted by those limitations even though the mSQL data type is not.

Miscellaneous Functions

msqlLoadConfigFile()

```
int msqlLoadConfigFile( file )
        char           * file;
```

The msqlLoadConfigFile( ) function can be used to load a non-default configuration file into your client application. The configuration file can include information such as the TCP/IP and UNIX ports on which the desired mSQL server will be running. The file to be loaded is determined by the value of the file parameter. If the value of the parameter is *new*, the msqlLoadConfigFile( ) function would search for the file in the following places (and in the order specified).

```
Inst_Dir/new
Inst_Dir/new.conf
new
```

That is, if a file called "new" exists in the installation directory, it is loaded. Otherwise, an attempt will be made to load a file called new.conf from the installation directory. If that fails, the filename specified is assumed to be a complete, absolute pathname and an attempt to open the file is made. On failure, the function will return a value of –1, otherwise a value of 0 is returned.

System Variables

Mini SQL 3.x includes internal support for system variables (often known as pseudo fields or pseudo columns). These variables can be accessed in the same way that normal table fields are accessed although the information is provided by the database engine itself rather than being loaded from a database table. System variables are used to provide access to server maintained information or meta data relating to the databases.

System variables may be identified by a leading underscore in the variable name. Such an identifier is not normally valid in mSQL for table or field names. Examples of the supported system variables and uses for those variables are provided below.

_rowid

The _rowid system variable provides a unique row identifier for any row in a table. The value contained in this variable is the internal record number used by the mSQL engine to access the table row. It may be included in any query to uniquely identify a row in a table. An example of the use of the _rowid system variable is shown below.

```
select _rowid, first_name, last_name from emp_details
        where last_name = 'Smith'

update emp_details set title = 'IT Manager'
        where _rowid = 57
```

The query optimiser is capable of utilising _rowid values to increase the performance of database accesses. In the second example query above, only one row (the row with the internal record ID of 57) would be accessed. This is in contrast to a sequential search through the database looking for that value which may result in only one row being modified but every row being accessed. Using the _rowid value to constrain a search is the fastest access method available in mSQL. As with all internal access decisions, the decision to base the table access on the _rowid value is automatic and requires no action by the programmer or user other than including the _rowid variable in the *where* clause of the query.

The rowid of a table row is intended to be used for "in place" updates. An example of such an update is the query above. The rowid can be used when there is no other way to identify a particular row (e.g. there are two people called John Smith and staff identifiers are not being used). It is not to be used as a substitute for an application maintained key or index. Applications should use sequences if they wish to use server maintained unique values.

_timestamp

The _timestamp system variable contains the time at which a row was last modified. The value, although specified in the standard UNIX time format (i.e. seconds since the epoch), is not intended for interpretation by application software. The value is intended to be used as a point of reference via which an application may determine if a particular row was modified before or after another table row. The application should not try to determine an actual time from this value as the internal representation used may change in a future release of mSQL.

The primary use for the _timestamp system variable will be internal to the mSQL engine. Using this information, the engine may determine if a row has been modified after a specified point in time (the start of a transaction, for example). It may also use this value to synchronise a remote database for database replication. Although neither of these functions is currently available, the presence of a row timestamp is the first step in the implementation.

Example queries showing possible uses of the _timestamp system variable are show below.

```
select first_name, _timestamp from emp_details
        where first_name like '%fred%' order by _timestamp

select * from emp_details where _timestamp 88880123
```

_seq

The _seq system variable is used to access the current sequence value of the table from which it is being selected. The current sequence value is returned and the sequence is updated to the next value in the sequence (see the CREATE definition in the Language Specification section for more information on sequences). Once the sequence value has been read from the server using a select statement, the value can be inserted into "normal" fields of a table as a unique index value such as a serial number or staff identifier.

An example query using _seq system variable is shown below.

```
select _seq from staff
```

_sysdate

The server can provide a central standard for the current date. If selected from any table, the _sysdate system variable will return the current date on the server machine using the mSQL date format of 'DD-Mon-YYYY'.

An example query using _sysdate system variable is shown below.

```
select _sysdate from staff
```

_systime

The server can provide a central standard for the current time. If selected from any table, the _systime system variable will return the current time on the server machine using the mSQL time format of 'HH:MM:SS'.

An example query using _systime system variable is shown below.

```
select _systime from staff
```

_user

By selecting the _user system variable from any table, the server will return the username of the user who submitted the query.

An example query using _user system variable is shown below.

```
select _user from staff
```

Standard Programs and Utilities

The mSQL distribution contains several programs and utilities to allow you to use and manage your databases. The tools provided allow you to communicate with the database server, import data, export data, submit queries and view your database structures. The sections below provide detailed descriptions on the various tools provided in the distribution.

The monitor – msql

Usage

msql [-h host] [-f confFile] database

Options

-h	Specify a remote hostname or IP address on which the mSQL server is running. The default is to connect to a server on the localhost using a UNIX domain socket rather than TCP/IP (which gives better performance).
-f	Specify a non-default configuration file to be loaded. The default action is to load the standard configuration file located in INST_DIR/msql.conf (usually /usr/local/msql3/msql.conf). Please see the msqlLoadConfigFile entry in the API section of this manual to understand the method used to select the config file from the specified file name.

Description

The mSQL monitor is an interactive interface to the mSQL server. It allows you to submit SQL commands directly to the server. Any valid mSQL syntax can be entered at the prompt provided by the mSQL monitor. For example, by typing a “create table” clause at the mSQL monitor prompt you can instruct the database server to create the specified table. The mSQL monitor is intended to be used as a mechanism for creating your database tables and for submitting ad-hoc SQL queries to the server. It is not intended to be used for client application development other than for testing queries before they are coded into your applications.

Control of the monitor itself is provided by four internal commands. Each command is comprised of a backslash followed by a single character. The available commands are

\q	Quit (also achieved by entering Control-D)
\g	Go (Send the query to the server)
\e	Edit (Edit the previous query)
\p	Print (Print the query buffer)

Schema viewer – relshow

Usage

relshow [-h host] [-f confFile] [database [rel [idx]]]

Options

-h	Specify a remote hostname or IP address on which the mSQL server is running. The default is to connect to a server on the localhost using a UNIX domain socket rather than TCP/IP (which gives better performance).
-f	Specify a non-default configuration file to be loaded. The default action is to load the standard configuration file located in INST_DIR/msql.conf (usually /usr/local/msql3/msql.conf). Please see the msqlLoadConfigFile entry in the API section of this manual to understand the method used to select the config file from the specified file name.

Description

Relshow is used to display the structure of the contents of mSQL databases. If no arguments are given, relshow will list the names of the databases currently defined. If a database name is given it will list the tables defined in that database. If a table name is also given then it will display the structure of the table (i.e. field names, types, lengths etc).

If an index name is provided along with the database and table names, relshow will display the structure of the specified index including the type of index and the fields that comprise the index.

Admin program - msqladmin

Usage

msqladmin [-h host] [-f confFile] [-q] Command

Options

-h	Specify a remote hostname or IP address on which the mSQL server is running. The default is to connect to a server on the localhost using a UNIX domain socket rather than TCP/IP (which gives better performance).
-f	Specify a non-default configuration file to be loaded. The default action is to load the standard configuration file located in INST_DIR/msql.conf (usually /usr/local/msql3/msql.conf). Please see the msqlLoadConfigFile entry in the API section of this manual to understand the method used to select the config file from the specified file name.
-q	Put msqladmin into quiet mode. If this flag is specified, msqladmin will not prompt the user to verify dangerous actions (such as dropping a database).

Description

msqladmin is used to perform administrative operations on an mSQL database server. Such tasks include the creation of databases, performing server shutdowns, etc. The available commands for msqladmin are

create db_name	Creates a new database called db_name.
drop db_name	Removes the database called db_name from the server. This will also delete all data contained in the database!
shutdown	Terminates the mSQL server.
reload	Forces the server to reload ACL information.
version	Displays version and configuration information about the currently running server.
stats	Displays server statistics.
copy fromDB toDB	Copies the contents of the database specified as the fromDB into a newly created database called toDB. If the toDB already exists an error will be returned. This command provides a simple mechanism for creating a backup copy of a data for use as a test or development environment.
move fromDB toDB	Renames an existing database called fromDB to toDB. The data is not modified in any way.

Note : most administrative functions can only be executed by the user specified in the run-time configuration as the admin user. They can also only be executed from the host on which the server process is running (e.g. you cannot shutdown a remote server process).

Data dumper - msqldump

Usage

msqldump [-h host] [-f confFile] [-c] [-v] [-t] [-w WhereClause] database [table]

Options

-h	Specify a remote hostname or IP address on which the mSQL server is running. The default is to connect to a server on the localhost using a UNIX domain socket rather than TCP/IP (which gives better performance).
-f	Specify a non-default configuration file to be loaded. The default action is to load the standard configuration file located in INST_DIR/msql.conf (usually /usr/local/msql3/msql.conf). Please see the msqlLoadConfigFile entry in the API section of this manual to understand the method used to select the config file from the specified file name.
-c	Include column names in INSERT commands generated by the dump.
-t	Dump only the table definition, not the table data
-v	Run in verbose mode. This will display details such as connection results, etc.
-w	The argument following the –w flag is used as a query condition to limit the data that is dumped. An example may be -w "last_name = 'Hughes' "

Description

msqldump produces an ASCII text file containing valid SQL commands that will recreate the table or database dumped when piped through the mSQL monitor program. The output will include all CREATE TABLE commands required to recreate the table structures, CREATE INDEX commands to recreate the indices, and INSERT commands to populate the tables with the data currently contained in the tables. If sequences are defined on any of the tables being dumped, a CREATE SEQUENCE command will be generated to ensure the sequence is reset to its current value.

Data exporter - msqlexport

Usage

msqlexport [-h host] [-f conf] [-v] [-s Char] [-q Char] [-e Char] database table

Options

-h	Specify a remove hostname or IP address on which the mSQL server is running. The default is to connect to a server on the localhost using a UNIX domain socket rather than TCP/IP (which gives better performance).
-f	Specify a non-default configuration file to be loaded. The default action is to load the standard configuration file located in INST_DIR/msql.conf (usually /usr/local/msql3/msql.conf). Please see the msqlLoadConfigFile entry in the API section of this manual to understand the method used to select the config file from the specified file name.
-v	Verbose mode.
-s	Use the character Char as the separation character. The default is a comma.
-q	Quote each value with the specified character.
-e	Use the specified Char as the escape character. The default is \

Description

msqlexport produces an ASCII export of the data from the specified table. The output produced can be used as input to other programs such as spreadsheets. It has been designed to be as flexible as possible. The user may specify the character to use to separate the fields, the character to use to escape the separator character if it appears in the data, and whether the data should be quoted and if so what character to use as the quote character. The output is sent to stdout with one data row per line.

An example use of msqlexport would be to create a Comma Separated Values (CSV) file to be imported into a popular spreadsheet application such as Microsoft Excel. The CSV format uses a comma to separate data fields and quotation marks to quote the individual values. If a value contains a quotation mark, it is escaped by prefixing it with another quotation mark. To generate a CSV representation of a table called staff in the company database, the msqlexport command below would be used.

```
msqlexport –s ,  q “  –e “  company staff
```

Data importer - msqlimport

Usage

msqlimport [-h host] [-f conf] [-v] [-s Char] [-q Char] [-e Char] database table

Options

-h	Specify a remote hostname or IP address on which the mSQL server is running. The default is to connect to a server on the localhost using a UNIX domain socket rather than TCP/IP (which gives better performance).
-f	Specify a non-default configuration file to be loaded. The default action is to load the standard configuration file located in INST_DIR/msql.conf (usually /usr/local/msql3/msql.conf). Please see the msqlLoadConfigFile entry in the API section of this manual to understand the method used to select the config file from the specified file name.
-v	Verbose mode.
-s	Use the character Char as the separation character. The default is a comma.
-q	Remove quotes around field values if they exist (the specified character is the quote character).
-e	Use the specified Char as the escape character. The default is \

Description

msqlimport loads a flat ASCII data file into an mSQL database table. The file can be formatted using any character as the column separator. When passed through msqlimport, each line of the text file will be loaded as a row in the database table. The separation character, as specified by the -s flag, will be used to split the line of text into columns. If the data uses a specific character to escape any occurrence of the separation character in the data, the escape character can be specified with the -e flag and will be removed from the data before it is inserted. Some data formats (such as the CSV format) will enclose an entire value in quotation marks. The –q option can be used to indicate such a format and to specify the character being used for quoting.

To import a file formatted in the Comma Separated Values format (CSV) into a table called staff in the company database, the msqlimport command below would be used.

```
msqlimport –s , -q “ –e “ company staff
```

Appendix A - mSQL Error Messages

Listed below is a complete set of the error messages generated by the mSQL database engine and the client API library. Accompanying each error message is an indication of the cause of the error and actions that you can take to resolve the problem. The errors relate to user-generated problems, such as incorrect SQL query syntax, and also system related problems, such as running out of disk space. A user will not normally see many of these errors.

API Library Error Messages

Bad packet received from server

The server process received a data request that was incorrectly formatted. Such requests are ignored by the server and are probably caused by using an incorrect implementation of the client API library.

Can't find your username. Who are you?

The mSQL client library has attempted to translate the User ID (Unix UID) of the process running the client application into a username. The attempt failed. This will be the result if the UID of the client process is not listed in the system's password file.

Can't create UNIX socket

The API library has attempted to communicate with an mSQL server running on the local machine. In doing so, it tried to create a UNIX domain socket over which the client / server communications would pass. The attempt to create this socket failed. If this error message is generated when starting the server process, it implies that the user running the server does not have permission to create the UNIX socket. Check the msql.conf file to determine whether the mSQL_User field and the UNIX_Port fields are set correctly. If the error is generated by a client application it can indicate that the client process has too many files open at the same time.

Can't connect to local mSQL server

An attempt was made to form a connection with an mSQL server running on the local machine. The attempt failed. The mSQL server process is probably not running on this machine.

Can't connect to mSQL server on

An attempt to connect to an mSQL server process running on a remote machine failed. This is usually due to not having an mSQL server running on the remote machine

Can't create IP socket

The API library attempted to create a TCP/IP socket for use in communicating with an mSQL process on a remote machine. The attempt failed. This is commonly caused by having too many open files in the client application.

mSQL server has gone away

While the API library was communicating with an mSQL server process, the server process closed the client / server connection. This can be caused by the server process being terminated, the machine on which the server is running being rebooted, or a network failure if the server is on a remote machine.

Protocol mismatch. Server Version = x Client Version = y

The version of the mSQL client / server protocol used by the client API library and the server process do not match. Upgrading the server software but not relinking the client applications that are communicating with the server can cause this problem.

Unknown mSQL error

An error occurred while communicating with the server process but the server was not able to determine the cause of the error.

Unknown mSQL Server Host

The hostname passed to the msqlConnect( ) function could not be resolved into an IP address. Ensure that the listed machine is in the nameserver or hosts file of the machine running the client applications

Server Error Messages

Access to database denied

An attempt to access a database was rejected based on the contents of the access control list for that database.

Bad handshake

The initial handshake during connection establishment between a client application and the mSQL server process was incorrectly formatted.

Bad order field. Field "x" was not selected

The field x was used in an "order by" statement but it was not included in the list of fields selected from the table. You can only order by a field you have selected.

Bad type for comparison of 'x'

The value used within the where condition associated with the field x was of an incompatible type.

Can't perform LIKE on int value

An attempt was made to perform a regular expression match on an integer value. Regex searching can only be performed on character fields.

Can't perform LIKE on real value"

An attempt was made to perform a regular expression match on an real value. Regex searching can only be performed on character fields.

Can't get hostname for your address

The server process tried to resolve a hostname from the IP address of the machine running the client application. This will occur if the client machine is not listed in the nameserver. You may overcome this problem by setting the Host_Lookup field of msql.conf to false. Note that disabling this option will effect access control for the databases as the hostname of the client is used during ACL tests.

Can't open directory "x" (y)

An attempt to access the directory listed as x was not successful. The system error message is returned as y.

Can't use TEXT fields in LIKE comparison

A select query attempted to perform a regular expression match against a TEXT field. TEXT fields cannot be used in regex matches.

Couldn't create temporary table (y)

An attempt to create the files associated with a temporary table failed during the execution of a query. This is usually caused by having the permission of the msqldb/.tmp directory in the installation directory set incorrectly. The system error message is returned as y.

Couldn't open data file for x (y)

An attempt to open the data file for a table called x in the currently selected database failed. The system error message is returned as y.

Data write failed (x)

A write operation failed while attempting to store data into a table or temporary table. The system error message describing the error is returned as x. The usual reason for this problem is lack of disk space during an insert operation or a multi-table join that produced an overly large result set.

Error creating table file for "x" (y)

Creation of the data file for table x failed. A description of the error is returned in y

Error reading table "x" definition (y)

An error was encountered while the server attempted to read the table definition for the table called x in the currently selected database. The system error message is returned as y.

Field "x" cannot be null

An attempt was made during either an insert or an update operation to set the value of a NOT NULL field to NULL.

Index field "x" cannot be NULL

An attempt to insert a NULL value into an index was rejected by the server. Indices cannot be set to NULL.

Index condition for "x" cannot be NULL

Because indices cannot contain NULL values, the attempt to use a NULL condition for an index lookup was rejected.

Invalid date format "x"

The value “x” is not a valid date format and was rejected by the server. This may occur when inserting into a date field, updating the value of a date field, or when using a date value in a where condition

Invalid time format "x"

The value “x” is not a valid time format and was rejected by the server. This may occur when inserting into a time field, updating the value of a time field, or when using a time value in a where condition

Literal value for 'x' is wrong type

During either an insert or an update operation, an attempt was made to set the value of a field to a type that was not compatible with the defined type of the field (e.g. setting an integer field to a character value).

No Database Selected

A query was sent to the server before a database had been selected. The client application must call the msqlSelectDB function before submitting any queries.

No value specified for field 'x'

An insert operation sent to the database engine did not include a value for field x. Either the field x was specified in the field list and not enough values were provided or the number of values provided did not match the total number of fields in the table.

Non unique value for unique index

A value that was assigned to a unique index via either an insert or an update clause attempted to insert a duplicate value.

Out of memory for temporary table

An attempt by the server to malloc a memory segment was refused during the creation of a temporary table (during execution of a query).

Permission denied

An attempt to access a database in a particular manner was refused due to the ACL configuration of the selected database. Usually, write access (i.e. inserts or updates) are rejected because the database is listed as read-only for that client.

Reference to un-selected table "x"

A select query referenced a variable from table x when x was not listed as a table in the select clause.

Table "x" exists"

An attempt to create a table called x failed because the table already exists.

Too many connections

An attempt to connect to the mSQL server process was refused because the maximum number of simultaneous client connections has been reached.

Too many fields in query

The query attempted to reference more fields in a single query than the server permits. By default, up to 75 fields can be referenced in the same query.

Too many fields in condition

The number of conditions included in the where clause exceeded the allowed maximum of 75.

Unknown command

A unknown client / server protocol command was received by the server and was rejected.

Unknown database "x"

An attempt was made to access a database called x although no such database is defined on the server

Unknown table "x"

An attempt was made to access a table called x although no such table is defined in the currently selected database.

Unknown field "x.y"

An attempt was made to reference a field called x in a table called y. Table y of the currently selected database does not contain a field called x.

Unknown system variable "x"

A query referenced a system variable called x when no such system variable exists.

Unqualified field in comparison

An unqualified field name was used in the where clause of a join. You must fully qualify all field names if you reference more than one table.

Unqualified field "x" in join

The field x was not fully qualified in the field list of a join. All fields referenced in a join must be fully qualified.

Value for "x" is too large

The value provided for field x in either an insert or an update clause was larger than the defined length of the field