

# *eXtremeSQL*

---

User's Guide  
Version 3.1

(C) 2005-2007 McObject LLC  
22525 SE 64th Place, Suite 302  
Issaquah, WA 98027 USA

Tel: +1-425-888-8505  
Fax: +1-425-888-8508  
Email: [info@mcobject.com](mailto:info@mcobject.com)  
Web: <http://www.mcobject.com>

## Contents

Chapter 1: Introduction .....	1
<i>Product Overview</i> .....	1
Benefits of Using <i>eXtremeSQL</i> .....	1
<i>eXtremeSQL</i> Query Optimization.....	2
Chapter 2: SQL and <i>eXtremeSQL</i> .....	3
<i>Introduction to SQL</i> .....	3
<i>SQL Language Categories</i> .....	3
<i>eXtremeSQL Operational Overview</i> .....	4
<i>eXtremeSQL Processing</i> .....	5
<i>eXtremeSQL DML Syntax Summary</i> .....	5
Grammar Conventions.....	5
Syntax.....	6
Identifiers.....	8
<i>eXtremeSQL Reserved Words</i> .....	8
<i>eXtremeSQL DML Statements</i> .....	9
<i>Sample eXtremeSQL Program</i> .....	12
Chapter 3: <i>eXtremeSQL</i> Programming Interface .....	16
<i>C/C++ Header Files</i> .....	16
<i>Type Mapping</i> .....	17
<i>Extracting and Storing Data</i> .....	17
Strings in <i>eXtremeSQL</i> .....	17
StringLiteral.....	18
<i>eXtremeSQL</i> References and Their Usage.....	20
Arrays and Their Usage.....	20
Vectors and Arrays.....	21
Storing Binary Data with Zeros.....	22
Functions.....	22
Other <i>eXtremeSQL</i> Supported Functions.....	24
<i>C Structs</i> .....	25
<i>Initializing/Shutting Down from a C Application</i> .....	26
<i>Initializing/Shutting Down from a C++ Application</i> .....	26
<i>Interactive SQL (XSQL utility)</i> .....	29
Standard and <i>eXtremeDB</i> -Specific Commands.....	29
Example of Report Output.....	29
Chapter 4: <i>eXtremeSQL</i> Query Optimization .....	30
indexable expression.....	32
known value.....	33
Distinct.....	34
Order By.....	34
Subquery.....	34
Show Plan.....	34
Chapter 5: McObject ODBC Driver .....	36
<i>Installing and configuring the eXtremeSQL ODBC Driver</i> .....	38
Connection to a local database.....	39
Connection to a remote database.....	40
<i>McObject ODBC Implementation Details</i> .....	42
McObject ODBC and <i>eXtremeSQL</i> proprietary API Together.....	43
Chapter 6: SQL Meta Data .....	45
Appendices.....	47
Appendix A: sql2mco Utility .....	48
<i>Formal Grammar</i> .....	48



# Chapter 1: Introduction

## *Product Overview*

McObject's *eXtremeSQL*<sup>™</sup> is a high-performance implementation of the SQL database programming language for the *eXtremeDB*<sup>™</sup> in-memory database. With *eXtremeSQL*, McObject targets the real-time enterprise software market by greatly simplifying programming with *eXtremeDB* for corporate developers using SQL. The new interface strengthens *eXtremeDB*'s appeal for application development in fields such as banking and securities trading, where real-time responsiveness is a must and SQL is the dominant database language.

Built on the unsurpassed performance of *eXtremeDB*, and a SQL optimizer tuned for main memory database access, *eXtremeSQL* delivers blazingly fast processing of dynamic SQL queries.

### **Benefits of Using *eXtremeSQL***

- **Co-existence with native *eXtremeDB***  
Use *eXtremeSQL* alongside the *eXtremeDB* API in the same application. Where maximum performance is critical, the *eXtremeDB* API can't be beat. When a higher level of access would be beneficial, such as retrieving data from multiple tables or performing aggregation, then *eXtremeSQL* is advantageous.
- **Broad coverage of the SQL-89 standard**  
*eXtremeSQL* implements much of the ANSI SQL-89 specification.
- **Extensions to exploit *eXtremeDB* features and data types**  
In addition, *eXtremeSQL* implements *eXtremeDB*-specific extensions including support for structures, arrays and vectors, as well as query optimizations based on specific *eXtremeDB* capabilities.
- **Compatibility with all *eXtremeDB* editions**  
*eXtremeSQL* is fully compatible and interoperable with *eXtremeDB Standard Edition*, *High Availability Edition*, and *Transaction Logging Edition*. There is no need to worry about the migration path within the *eXtremeDB* product family.
- **No client/server inter-process communications**  
Like *eXtremeDB*, *eXtremeSQL* is embedded in the application, not deployed as a separate process. This eliminates client/server inter-process communication round-trips from the execution path, resulting in breakthrough performance and zero administration.
- **Interactive SQL utility**  
*eXtremeSQL* comes with an interactive SQL program, XSQL, which can be used to test SQL statements independently from application programs. Full source code for the utility is provided so that it also serves as a useful example of a full *eXtremeSQL* implementation. In addition to executing SQL statements, XSQL also supports *eXtremeDB*-specific commands, such as "report" and "save <file>."

XSQL can also be employed as a batch processing utility by redirecting input from a text file containing *eXtremeSQL* statements.

### ***eXtremeSQL* Query Optimization**

Creating the optimal plan for execution of SQL statements is a very complex and challenging task. SQL optimizers analyze SQL queries sent to the database and select the best search strategies for accessing the database.

There are two classes of SQL optimizers: cost-based and rule-based. Disk-based databases generally use a cost-based optimizer. With cost-based optimizers, query optimization greatly depends on data distribution. Often, optimizers take samples and use statistics provided by the database engine, and collect statistical information themselves, to calculate the cost of candidate execution plans. Therefore, building optimal plans is a very CPU-intensive operation and inherently unpredictable; the amount of time spent in the optimizer varies from query to query and execution plans can change from one invocation to another as the distribution of data changes. The CPU usage is considered a good trade off, given the relatively high cost of disk-access.

For real-time and embedded systems where predictable performance is key, a rule-based optimizer such as is used by *eXtremeSQL* is more appropriate. In addition, the *eXtremeSQL* optimizer makes it possible for applications to specify their own execution plan. For example, the optimizer never reorders tables in the query: the joins are performed in the sequence the tables were specified in the query. Some of the other key rules that are used for query optimization include:

- If possible, an index is used.
- Each table is assigned an ordinal number representing its position in the FROM list.
- The search predicate is divided into the set of conjuncts and the conjuncts are sorted. Therefore the expressions accessing the tables with smaller ordinal numbers are checked first.
- The execution of subqueries is optimized by checking the dependencies of the subquery expression. The results of the subquery are saved and recalculated only if the subquery expression refers to fields from the enclosing scope.

For more information about query optimization in *eXtremeSQL*, see [Chapter 4: \*eXtremeSQL\* Query Optimization](#).

# Chapter 2: SQL and *eXtremeSQL*

This chapter covers the similarities and differences between SQL and *eXtremeSQL*.

## *Introduction to SQL*

SQL (Structured Query Language) was developed by IBM in the mid '70s as a way to get information in and out of database systems. Contrary to what its name suggests, SQL is not only a query language, but also a language for manipulating (adding, changing or deleting) information already in the database.

SQL is a declarative language, whereas conventional programming languages (C/C++, Java, etc.) are procedural languages. Practically speaking, this means that SQL has no control statements (if and while, for example).

SQL has been widely adopted by database vendors and has been institutionalized as a standard by the ANSI and ISO organizations. SQL skills learned with one database system can be largely carried over to other SQL-capable database systems, though virtually every vendor implements proprietary extensions to the language to exploit capabilities of their own database technology.

The ANSI SQL standard has evolved over the years, beginning with the SQL-89 standard, followed by SQL-92 and, most recently, the SQL-99 standard. *eXtremeSQL* provides broad implementation of the SQL-89 standard for SQL DML statements.

## *SQL Language Categories*

SQL consists of a set of statements, which are categorized into:

- Data Definition Language (DDL) statements
- Data Control Language (DCL) statements
- Data Manipulation Language (DML) statements.

DDL statements are used to describe the data stored in the database. Examples are CREATE TABLE and CREATE INDEX statements. The DDL can also define constraints and other attributes of the data. See [Appendix A: sql2mco Utility](#) for more information about the SQL to DDL utility.

DCL statements define privileges to access the data by means of statements such as GRANT and REVOKE.

**Note: *eXtremeSQL* does not implement SQL DCL statements.**

DML statements manipulate or retrieve data, as opposed to define or control access to data. The DML consists of four statements:

- SELECT
- INSERT
- UPDATE
- DELETE

## *eXtremeSQL Operational Overview*

Figure 1.1 depicts the major operational steps and elements of *eXtremeSQL*.

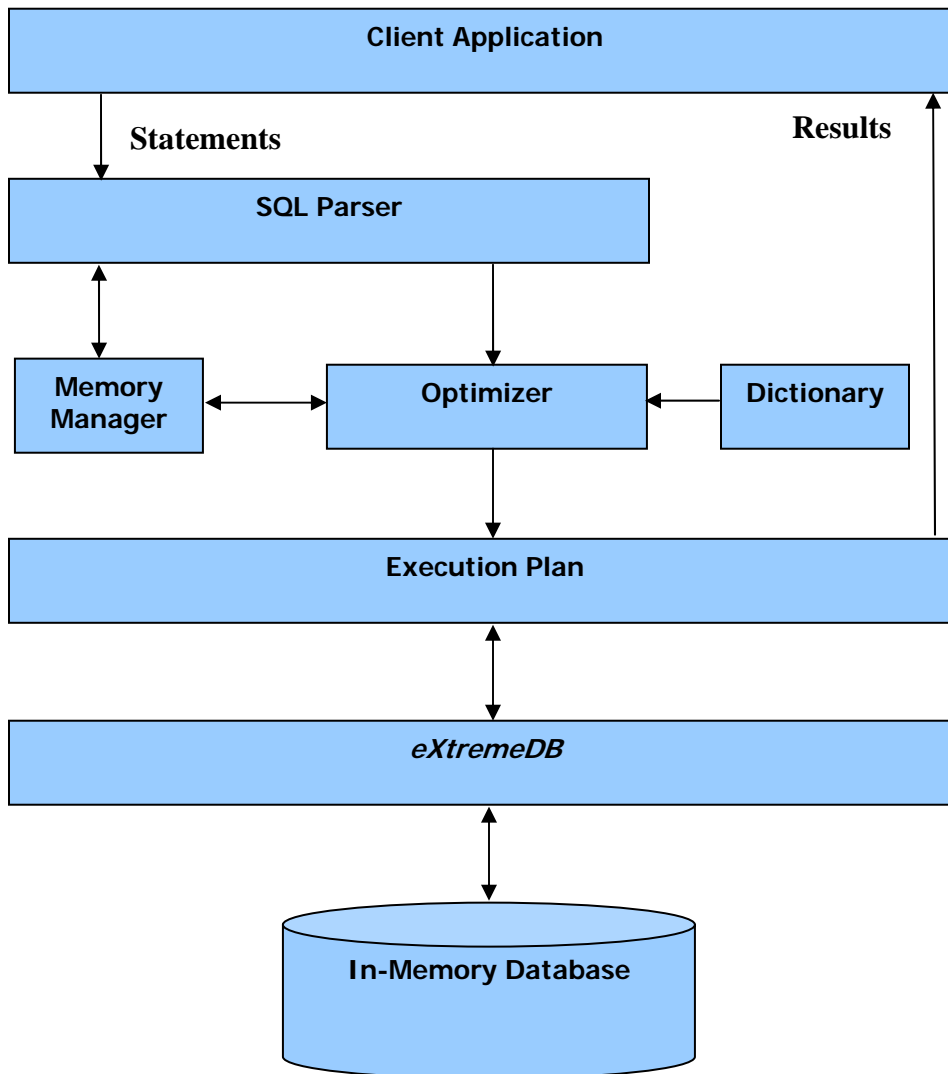


Figure 1.1: Operational Overview – *eXtremeSQL*

SQL statements are embedded within C/C++ programs and submitted to *eXtremeSQL* through the *eXtremeSQL* programming interface described in [Chapter 3: \*eXtremeSQL\* Programming Interface](#).

The SQL statements are parsed, which verifies the syntactical correctness of the statements. If no errors are found, the optimizer is invoked, which attempts to determine the most efficient means of processing the statement by interrogating the database dictionary to discover potential indexes and autoid/reference relationships between classes. Optimization is discussed in [Chapter 4: \*eXtremeSQL\* Query Optimization](#). This step results in an execution plan that identifies the procedural steps *eXtremeSQL* will take in producing the result set. (e.g., locate an object of class X by the index on field A; use the value of field B of the found object as a search value on indexed field D of class Y, and so on.)

After the statement has been parsed, optimized, and an execution plan formed, the application uses the *eXtremeSQL* programming interface to fetch results.

### ***eXtremeSQL* Processing**

Multiple processes can simultaneously access a database in shared memory by using the `McoSqlEngine` class. If two or more threads within a single process need simultaneous access to a database, the sub-class `McoMultithreadedSqlEngine` should be used.

The number of concurrent threads accessing the database through *eXtremeSQL* should be considered carefully. The reason for this stems from the characteristics of the major market segments targeted by McObject database products, e.g. embedded systems. Specifically, embedded systems generally have less memory and relatively slower processors. The nature of any dynamic SQL implementation is that it requires fairly extensive use of dynamic memory allocation, for holding tokens, etc, during parsing, for temporary results during execution, and miscellaneous other uses. Since embedded systems do not have plentiful memory, it is prudent to restrict the number of tasks (threads) simultaneously processing SQL to minimize the memory consumption. Further, the act of allocating and freeing memory consumes CPU cycles: another resource that should not be misused. The implementation of *eXtremeSQL* mitigates the problem by using an allocator that is a stack-based allocator. This means that a mark is placed on a stack of memory and a stack pointer initialized to the location of the mark. As memory is needed during query processing, the pointer is advanced. For example, if 10 bytes are required, the address of the stack pointer is returned and the pointer is advanced 10 bytes. This process continues as the SQL is processed. When memory is no longer required, it is freed by rewinding the pointer to the original mark. This is a very CPU-efficient memory allocation technique, but not one that is thread-safe. If memory allocations of two threads were intermingled in this way, then one thread's rewinding of the stack pointer would deallocate the other thread's memory. Therefore, each thread requires its own dedicated heap.

### ***eXtremeSQL* DML Syntax Summary**

#### **Grammar Conventions**

The following rules, in BNF-like notation, specify the grammar of the *eXtremeSQL* statements: The following grammatical conventions should be noted:

EXAMPLE	MEANING
expression	non-terminals
NOT	terminals
	disjoint alternatives
[NOT]	optional part
{1..9}	repeat zero or more times

## Syntax

The syntax for the DML statements is as follows:

```

statement ::=
query-statement | update-statement | insert-statement | delete-statement
query-statement ::= select-statement {set-op select-statement }
    [ORDER BY columns-ref-list] [LIMIT expression [,expression]] [FOR
UPDATE]
select-statement ::= SELECT [ALL | DISTINCT] [columns-list] FROM table-
list
    [WHERE expression] [group-by]
update-statement ::= UPDATE table SET assignments [WHERE expression]
insert-statement ::= INSERT INTO table [(fields-list)] insert-data
insert-data ::= VALUES expressions-list | select-statement
delete-statement ::= DELETE FROM table [WHERE expression]

columns-list ::= * | column-spec {, column-spec}
column-spec ::= table.* | expression [[AS] identifier]
column ::= [table .] identifier
table-list ::= table-spec {, table-spec}
table-spec ::= table {join-kind table [USING fields-list] } [[AS]
identifier]
table ::= {package .} identifier
join-kind ::= [NATURAL] [LEFT OUTER] JOIN
package ::= identifier
fields-list ::= field {, field }
field ::= identifier { . identifier }
fields-declaration-list ::= field-declaration {, field-declaration}
field-declaration ::= field opensql-type {field-constraint} |
[CONSTRAINT identifier] constraint
constraint: PRIMARY KEY field | FOREIGN KEY field REFERENCES table [
(field) ] [ON DELETE CASCADE]
field-constraint: PRIMARY KEY | USING INDEX | [NOT] NULL | UNIQUE
    | [FOREIGN KEY] REFERENCES table [ (field) ] [ON DELETE CASCADE]
set-op ::= UNION [ALL] | INTERSECT | MINUS
assignments ::= assignment {, assignment}
assignment ::= field = expression
columns-ref-list ::= column-ref {, column-ref}
column-ref ::= (column | column-index) [ASC | DESC]
columns-index ::= integer-constant
group-by ::= GROUP BY columns-ref-list [HAVING expression]
new-fields-decls ::= field-declaration | ( fields-declaration-list ) |
constraint
expression ::= disjunction
disjunction ::= conjunction

```

```

    | conjunction OR disjunction
conjunction ::= comparison
    | comparison AND conjunction
comparison ::= operand = operand
    | operand != operand
    | operand <> operand
    | operand < operand
    | operand <= operand
    | operand > operand
    | operand >= quantor subquery
    | operand != quantor subquery
    | operand <> quantor subquery
    | operand < quantor subquery
    | operand <= quantor subquery
    | operand > quantor subquery
    | operand >= quantor subquery
    | operand [NOT] LIKE operand
    | operand [NOT] LIKE operand escape string
    | operand [NOT] IN operand
    | operand [NOT] IN subquery
    | operand [NOT] IN expressions-list
    | operand [NOT] BETWEEN operand AND operand
    | operand IS [NOT] NULL
    | operand IS table
operand ::= addition
addition ::= multiplication
    | addition + multiplication
    | addition || multiplication
    | addition - multiplication
multiplication ::= power
    | multiplication * power
    | multiplication / power
power ::= term
    | term ^ power
term ::= identifier | parameter | number | string
    | TRUE | FALSE | NULL | SYSDATE | NOW
    | ( expression )
    | NOT comparison
    | - term
    | subquery
    | term [ expression ]
    | identifier . term
    | function term
    | set-function
    | CAST ( expression [AS] opensql-type )
    | EXISTS identifier : term
    | EXISTS subquery
quantor ::= ANY | ALL | SOME
subquery ::= select-statement
function ::= ABS | LENGTH | LOWER | UPPER
    | INTEGER | REAL | STRING | SUBSTR |
    | SIN | COS | TAN | ASIN | ACOS |
    | ATAN | LOG | EXP | CEIL | FLOOR
set-function ::= COUNT ( * )
    | distinct-set-function ( DISTINCT term )
    | all-set-function ( [ALL] term )
distinct-set-function ::= MIN | MAX | AVG | SUM | COUNT

```

```

all-set-function ::= MIN | MAX | AVG | SUM
string ::= ' { { any-character-except-quote } [ '' ] } '
parameter ::= %i | %u | %b | %l | %p | %f | %t | %s | %w | %v | %r
expressions-list ::= ( expression { , expression } )
openssl-type ::= BIT | BOOLEAN | TINYINT | SMALLINT | BIGINT
                | REAL | FLOAT | DOUBLE | DATE | TIME | TIMESTAMP | BLOB |
                | string-type [( integer-constant )]
                | INTEGER [( integer-constant)]
                | UNSIGNED [( integer-constant)]
                | NUMERIC [( integer-constant [, integer-constant])]
                | DECIMAL [( integer-constant [, integer-constant])]
string-type ::= STRING | UNICODE
              | [UNICODE] (CHAR | VARCHAR | BINARY | VARBINARY)

```

### Identifiers

Identifiers are case sensitive, beginning with a..z, A..Z, '\_' or the '\$' character, containing only a-z, A..Z, 0..9 '\_' or '\$' characters, and do not duplicate SQL reserved words.

## *eXtremeSQL* Reserved Words

The following words are reserved for use in *eXtremeSQL* function calls. To ensure compatibility, applications should avoid the use of any of these words.

<i>eXtremeSQL</i> RESERVED WORDS LIST				
abs	acos	add	all	alter
and	any	as	asc	asin
atan	avg	between	binary	bigint
bit	blob	boolean	by	cascade
cast	char	ceil	constraint	cos
count	create	date	decimal	delete
desc	distint	domain	double	drop
escape	exists	exp	false	float
floor	for	foreign	from	group
having	in	index	inner	into
insert	intersect	is	integer	join
key	left	length	like	log
lower	max	min	minus	natural
not	now	null	numeric	on
or	order	outer	primary	real
reference	references	right	select	set
sin	smallint	some	sqrt	string
sum	substr	sysdate	table	tan
time	timestamp	tinyint	to	true
values	varbinary	varchar	unicode	union
unique	unsigned	update	upper	using
where				

*eXtremeSQL* extends the ANSI standard SQL operations by supporting bit manipulation operations. Operators and/or can be applied not only to boolean operands but also to operands of the integer type. The results of applying the and/or operator to integer operands is an integer value with its bits set by the bit-AND/bit-OR operation. Bit operations can be used for efficient implementation of small sets. Raising the integer and floating types, with the power operation ^, is supported by *eXtremeSQL*.

## ***eXtremeSQL DML Statements***

Basic data manipulation SQL statements (SELECT, INSERT, UPDATE, DELETE) let you query, manipulate, and control data.

### **SELECT**

```
SELECT [ALL | DISTINCT] [columns-list] FROM table-list
      [WHERE expression] [group-by] [FOR UPDATE]
```

#### **Description**

The SQL SELECT statement is used to display either all the columns in a table or just specific columns from the table (known as “projection”). See Figure 2.1.

The WHERE clause can be used in the SELECT statement for displaying specific/distinct rows (records) from a table. The WHERE clause can appear only after the FROM clause. Only the rows (records) that satisfy the WHERE condition are retrieved and displayed (known as “selection”). See Figure 2.1.

For example, given a table:

```
CREATE TABLE t (
  A integer,
  B integer,
  C integer,
  D integer,
  E integer
)
```

and given the query:

```
SELECT A, C, E
FROM t
WHERE E >= 10
```

PROJECTION ↓ (columns)					
	A	B	C	D	E
SELECTION → (rows)	1	2	3	4	5
	6	7	8	9	10
	11	12	13	14	15

Figure 2.1: Selection and Projection

The field elements of a few records may repeat. For instance, an Employee table might contain more than one record where the field ‘age’ has the value 35. Such common field based records can be grouped and displayed together using the group-by clause (known as “aggregation”).

### For Update

Note that *eXtremeSQL*’s implementation of the FOR UPDATE clause differs from the standard. *eXtremeSQL* updates are searched updates, not positioned updates. In *eXtremeSQL*, then, by using the FOR UPDATE clause you are instructing *eXtremeSQL* that you intend to execute updates during execution of the query and, to avoid issuing a transaction upgrade (from READ\_ONLY to READ\_WRITE), *eXtremeSQL* should open a READ\_WRITE transaction from the outset.

### INSERT

```
INSERT INTO table [(fields-list)] insert-data
fields-list ::= field { , field }
field ::= identifier { . identifier }
insert-data ::= VALUES expressions-list | select-statement
expressions-list ::= ( expression { , expression } )
```

### Description

The SQL INSERT statement creates a new row in the specified table, with the values indicated by the insert-data, which can be either a list of values or the result of a SELECT statement. If a fields-list is omitted, the application must provide a value in the expressions-list for every field in the table, in the same order in which the fields were defined in the database schema.

For example, given the table definition above, the following statements are equivalent:

```
INSERT INTO t VALUES (1, 2, 3, 4, 5)
INSERT INTO t (A, B, C, D, E) VALUES (1, 2, 3, 4, 5)
```

If a fields-list is provided, then fields that are not specified in the fields-list will take on the default value specified in the database schema, or the system-defined default for the field type (e.g. zero, or an empty string).

The result set of a SQL SELECT statement used in lieu of an expressions-list must return exactly the same number of fields as the table if no fields-list is specified, or the same number of fields specified in the fields-list if one is provided.

For example,

```
INSERT INTO t SELECT V, W, X, Y, Z from t2
or
INSERT INTO t (A, C, E) SELECT V, X, Z from t2
```

### UPDATE

```
UPDATE table SET assignments [WHERE expression]
assignments ::= assignment {, assignment}
assignment ::= field = expression
```

#### Description

This statement modifies the values of one or more fields in one or more rows of the specified table.

*eXtremeSQL* UPDATES are “searched updates.” Currently, *eXtremeSQL* does not support positioned UPDATE (i.e., the WHERE CURRENT OF cursor\_name construction).

The UPDATE statement will fail if the values provided violate uniqueness constraints (hash or unique tree indexes).

The UPDATE statement will fail if the values provided violate the domain dependency specified in the database schema by a field with an enumerated type.

**Note: *eXtremeDB* and, by extension, *eXtremeSQL*, do not enforce referential integrity. Take care NOT to update primary key values for which there are one or more referencing foreign keys.**

### DELETE

```
delete-statement ::= DELETE FROM table [WHERE expression]
```

#### Description

This statement deletes one or more rows of the specified table.

*eXtremeSQL* DELETES are “searched deletes.” Currently, *eXtremeSQL* does not support positioned DELETE (i.e., the WHERE CURRENT OF cursor\_name construction). The searched delete deletes all rows of the table that satisfy the WHERE condition.

**Note: *eXtremeDB* and, by extension, *eXtremeSQL*, do not enforce referential integrity. Take care NOT to delete rows with primary key values for which there are one or more referencing foreign keys.**

## Sample *eXtremeSQL* Program

The following is a simple program written with *eXtremeSQL*. Since *eXtremeSQL* is embedded, the statements are included within the host C program and then submitted to *eXtremeSQL* through the *eXtremeSQL* programming interface.

This basic application illustrates the four DML statements: SELECT, UPDATE, INSERT and DELETE. The database used is a small, two-person database (with two names). It tracks age, weight and name.

**Note: Comments are denoted by “//.”**

```
#include "mcosql.h"

using namespace McoSql;

// define eXtremeDB page size
size_t const PAGE_SIZE = 128;
// define size of eXtremeDB database
size_t const DATABASE_SIZE = 4*1024*1024;
// define map address for eXtremeDB database (shared memory)
void* const MAP_ADDRESS = (void*)0x20000000;

// Define structure corresponding to database record
struct Person {
    char const* name;
    int         age;
    float       weight;
};

// Forward declaration of function describing database schema
GET_DICTIONARY(simpledb);

McoSqlEngine engine; // McoSQL engine

// This function will add new record in database
void addPerson(Person* p)
{
    // Add new record to the table
    // Record will be initialized with data
    // passed through Person struct
    engine.executeStatement("insert into Person %r", p);
}

// This function selects all records from the table
void listPersons()
{
    // Execute query and store returned
    // data source in QueryResult object,
    // which destructor will automatically
    // release this data source after return from
    // the function
    QueryResult result(engine.executeQuery(
        "select * from Person order by age"));
}
```

```

// Get cursor
Cursor* cursor = result->records();

// Loop for all records
while (cursor->hasNext()) {
    Person p;

    // Get next record
    Record* rec = cursor->next();

    // Extract record to the correspondent struct
    result->extract(rec, &p, sizeof(p));

    printf("%s %d %f\n", p.name, p.age, p.weight);
}

// Search person by name pattern
void searchPersonByName(char const* name)
{
    // Execute query with given parameter "name"
    QueryResult result(engine.executeQuery(
        "select * from Person where name like %s", name));

    // Get cursor
    Cursor* cursor = result->records();

    // Loop for all records
    while (cursor->hasNext()) {
        Person p;

        // Get next record
        Record* rec = cursor->next();

        // Extract record to the correspondent struct
        result->extract(rec, &p, sizeof(p));

        printf("%s %d %f\n", p.name, p.age, p.weight);
    }
}

// Calculate average age of persons using aggregate functions
void calculateAverageAge()
{
    // Execute query calculating average age of all persons
    QueryResult result(engine.executeQuery(
        "select avg(age) from Person"));

    // Get cursor
    Cursor* cursor = result->records();

    // Result data source consists of one record with single column
    // Indices of columns start from 0
    Value* avgAge = cursor->next()->get(0);

    printf("Average age: %d\n", (int)avgAge->intValue());
}

```

```

}

// Update record
void updatePersonData(char const* name, int age, float weight)
{
    // Update person with specified name (passed as query parameter)
    // The result of this statement is the number of updated records
    int ret = engine.executeStatement(
        "update Person set age=%i, weight=%f where name=%s",
        age, weight, name);

    if (ret == 0) {
        printf("Person %s not found\n", name);
    }
}

// Delete record from the table
void deletePerson(char const* name)
{
    // Delete person with specified name (passed as query parameter)
    // The result of this statement is the number of removed records
    int ret = engine.executeStatement(
        "delete from Person where name=%s", name);

    if (ret == 0) {
        printf("Person %s not found\n", name);
    }
}

int main()
{
    Person p;

    // Open eXtremeDB database and SQL engine
    engine.open("simplifiedb", // database name
               simplifiedb_get_dictionary(), // database dictionary
               DATABASE_SIZE, // database size
               PAGE_SIZE, // page size
               MAP_ADDRESS); // map address for shared memory mode

    p.name = "John Smith";
    p.age = 35;
    p.weight = 72.1f;
    addPerson(&p);

    p.name = "Peter Brown";
    p.age = 40;
    p.weight = 62.1f;
    addPerson(&p);

    listPersons();

    searchPersonByName("John%");
    searchPersonByName("%Brown%");
}

```

```
calculateAverageAge();

updatePersonData("John Smith", 36, 75.2f);
updatePersonData("Peter Brown", 41, 65.0f);

listPersons();

deletePerson("John Smith");
deletePerson("Peter Brown");

// Close database and SQL engine
engine.close();
return 0;
}
```

# Chapter 3: *eXtremeSQL* Programming Interface

## *C/C++ Header Files*

These are the key C/C++ header files used in the *eXtremeSQL* API:

HEADER FILE	DESCRIPTION
extremesql/inc/sql.h	Main C++ interface to the SQL engine; actually SQL::execute method does most of the work
extremesql/inc/dbapi.h	Interfaces implemented by <i>eXtremeDB</i>
extremesql/inc/apidef.h	Basic classes used by the SQL engine
extremesql/inc/exceptions.h	Exceptions which can be thrown by the SQL engine
extremesql/inc/csql.h*	Pure C version of sql.h

**\*Note: *eXtremeSQL* has two programming interfaces: C++ and C; the file *csql.h* defines the APIs in the pure C API.**

Samples are located in extremesql/samples/sql.

*eXtremeSQL* provides a SQL-89 compatible query language with many extensions:

- Arrays
- Structures
- References
- Support of all C++ built-in types
- User-defined functions

Also there are some incompatibilities with the ANSI standard:

- Assigning null values to columns of some types is not supported.
- Changing the database schema at runtime (add table, add index,...) is not supported.
- Identifiers are case sensitive (SQL keywords are case insensitive).
- The FROM clause of the SELECT statement can only refer to the tables and can not include nested SELECTs.
- Views are not supported.
- Constraints are not enforced by the SQL interface itself, but are enforced by the underlying *eXtremeDB* runtime.
- RIGHT OUTER JOIN and FULL OUTER JOIN are not implemented.
- Triggers are not supported, but underlying *eXtremeDB* event notifications will be invoked if the *eXtremeSQL* action causes such an event.

## *Type Mapping*

The following table maps C++ types to *eXtremeSQL* types with the corresponding C++ type enumeration constant:

C++ TYPE	<i>eXtremeSQL</i> TYPE	TYPE ENUMERATION CONSTANT
bool	boolean bit	tpBool
signed char	tinyint int(1)	tpInt1
signed short	smallint int(2)	tpInt2
signed int	int integer int(4)	tpInt4
signed long long	bigint Int(8)	tpInt8
unsigned char	unsigned(1)	tpUInt1
unsigned short	unsigned(2)	tpUInt2
unsigned int	unsigned unsigned(4)	tpUInt4
unsigned long long	unsigned(8)	tpUInt8
float	float numeric(4)	tpFloat
double	double numeric(8) real decimal	tpDouble
char*	varchar char string	tpString
time_t	date time timestamp	tpDateTime

## *Extracting and Storing Data*

### **Strings in *eXtremeSQL***

All operations that are applicable to arrays are also applicable to strings, which also have their own set of operations. For example, strings can be compared with each other using the standard relational operators.

The construction *like* can be used for matching a string with a pattern containing the special wildcard characters '%' and '\_'. The character '\_' matches any single character, while the character '%' matches any number of characters (including 0). The extended form of the *like* operator with an escape part can be used to handle characters '%' and '\_' in the pattern as normal characters only if they are preceded by the special escape character, which is specified after the escape keyword.

For example,

```
select * from T where name like '#%x%' escape '#'
```

will select all record where the **name** field starts with “%x.”

It is possible to search a substring within the string by using the *in* operator. So the expression 'blue' in color will be true for all the records for which the color field contains the string 'blue'.

For example,

```
select * from car where 'blue' in color;
```

will select car objects with color "dark-blue", "blue", "white-and-blue", "light-blue"...

Strings can be concatenated by using the + or || operators (+ and || may be used interchangeably). The last operator was added only for compatibility with the ANSI SQL standard. *eXtremeSQL* doesn't support implicit conversions to the string type in expressions, so the semantics of the operator + has been redefined for strings. In other words, in many SQL implementations it is possible to write:

```
1+'1'
```

and the result will be 2 (here implicit conversion is performed from string to integer).

The result of

```
1||'1'
```

will be '11'.

*eXtremeSQL* doesn't allow implicit conversion from strings, so the result of 1+'1' will be '11', the same as 1||'1'.

### **StringLiteral**

The `StringLiteral` constructor has a “chars” member variable with a size of 1. This illustrates the standard allocation of a structure with a variable size. When this structure is allocated, it is necessary to add the size of the fixed and the variable portions of the object. In C++, it can be done by using the operator *new* with an extra parameter. However, such an operator is not

predefined, and you would need to define it yourself (as it is done in the `DynamicObject` class):

```
inline void* operator new(size_t fixed, size_t var) {
    return ::new char[fixed + var];
}
```

Now the string literal can be created in the following way:

```
char const* str = "Hello World";
size_t length = strlen(str);
Value* value = ::new (length) StringLiteral(str, length);
```

There are two alternatives:

1. Use the `StringRef` class for which the constructor is given a pointer to the string and it is the responsibility of the programmer to de-allocate the string when it is not needed any longer.
2. Define your own class extending the `String` class:

```
class UserString : public String {
public:
    virtual int size();
    virtual char* body();
    virtual char* cstr();

private:
    const int length;
    char* chars;

public:
    UserString(char const* s, int l);
    ~UserString();
};

int UserString::size()
{
    return length;
}

UserString::UserString(char const* s, int l) : length(l)
{
    chars = new char[l+1];
    memcpy(chars, s, l);
    chars[l] = '\0';
}

UserString::~~UserString()
{
    delete[] chars;
}

char* UserString::body()
{
    return chars;
}

char* UserString::cstr()
{
    return chars;
}
```

***eXtremeSQL* References and Their Usage**

References can be used in *eXtremeSQL* for fast and direct access to the record by AUTOID. Reference fields can also be indexed and used in the ORDER BY clause. References can be de-referenced using the same dot notation as used in accessing structure components.

For example, if we have the following database schema:

```
class Address {
    string city;
    autoid[1000];

    hash<city> city_index[1000];
};

class Company {
    autoid_t<Address> address;
    autoid[10000];

    hash<address> address_index[10000];
};

class Contract {
    autoid_t<Company> company;
    autoid[10000];

    hash<company> company_index[10000];
};
```

Then the following query:

```
select * from Contract where company.address.city = 'Chicago';
```

Will retrieve Contract records where the referenced Company references an Address with the *city* field equal to 'Chicago'. The query execution plan for this request is the following:

1. Perform an index search of the Address table using *city\_index* (*city* = 'Chicago')
2. For all selected Address records, locate Company records referencing these addresses using *address\_index*
3. For all selected Company records, locate Contract records referencing these Company records using *company\_index*

References can be checked for *null* by the *is null* or *is not null* predicates. They can also be compared for equality with each other, as well as with the special *null* keyword. When a null reference is de-referenced, an exception is raised by *eXtremeSQL*.

**Arrays and Their Usage**

*eXtremeSQL* accepts *eXtremeDB* arrays (fixed length) and vectors (dynamic length) as components of records. As with *eXtremeDB*, multidimensional arrays are not supported. *eXtremeSQL* provides a set of special constructions for dealing with arrays and vectors (hereafter collectively referred to as 'array'):

1. It is possible to get the number of elements in the array by using the `length()` function.
2. Array elements can be retrieved by using the `[]` operator. If the index expression is out of the array range, then an exception will be raised.
3. The operator `in` can be used for checking if an array contains values specified by the left operand. This operation can be used only for arrays of atomic types; with **boolean**, **numeric**, **reference** or **string** components.
4. Iteration through array elements is performed by the `exists` operator. Variables specified after the `exists` keyword can be used as an index in the arrays for the expression preceded by the `exists` quantor. This index variable will iterate through all the possible array index values, until the value of expression becomes `true` or the index exceeds the array's range.

For example, given the following database schema:

```
class Company {
    string location;
    autoid;
};

class Contract {
    autoid_t<Company> company;
    unsigned<8> quantity;
    date          delivery;
    autoid;
};

class Detail {
    string name;
    vector< autoid_t<Contract> > contract;
};
```

the query:

```
select * from Detail where exists i:
(contract[i].company.location = 'US');
```

selects all the Detail records of companies located in the US.

While the query:

```
not exists i: (contract[i].company.location = 'US')
```

selects all the Detail records of companies outside the US.

### Vectors and Arrays

*eXtremeSQL*, unlike *eXtremeDB*, does not distinguish between the vector and the array types. Both types are represented by *eXtremeSQL*'s `tpArray` type. `tpList` type is the internal type used for building lists in SQL:

```
SELECT * from T where code in (1, 2, 10);
```

In this case, (1, 2, 10) is represented internally as an instance of `tpList`.

### Storing Binary Data with Zeros

You can store binary data that may contain zeroes in *eXtremeSQL*. To insert binary data with embedded zeroes, and to retrieve binary data with embedded zeroes using *eXtremeSQL*, see this example:

```
class t1
{
    signed<4> n;
    signed<4> o;
    string s;
    signed<1> b[16];
    list;
};
```

In this case, the type of the field will be an `Array` and you can use the `Array::getBody()` method to get the contents of the array.

### Functions

The table on the following page lists the *eXtremeSQL* built-in functions:

## Chapter 3: *eXtremeSQL* Programming Interface

NAME	ARGUMENT TYPE	RETURN TYPE	DESCRIPTION
abs(i)	integer	integer	absolute value of the argument
abs(r)	real	real	absolute value of the argument
sin(r)	real	real	sin (rad)
cos(r)	real	real	cos (rad)
tan(r)	real	real	tan (rad)
asin(r)	real	real	arcsin
acos(r)	real	real	arcsin
atan(r)	real	real	arctan
exp(r)	real	real	exponent
log(r)	real	real	natural logarithm
ceil(r)	real	real	the smallest integer value that is not less than r
floor(r)	real	real	the largest integer value that is not less than r
integer(r)	real	integer	conversion of real to integer
integer(s)	string	integer	conversion of string to integer
length(a)	array	integer	number of elements in array
length(s)	string	integer	length of string
lower(s)	string	string	lowercase string
real(i)	integer	real	conversion of integer to real
real(s)	string	real	conversion of string to real
string(i)	integer	string	conversion of integer to string
string(r)	real	string	conversion of real to string
substr(s,m[,n])	string,integer[,integer]	string	substring of s, beginning at character m, n characters long (if n is omitted, to the end of string)
upper	string	string	uppercase string

## Other *eXtremeSQL* Supported Functions

### Load and Save

The last optional parameter of the `McoSqlEngine::open()` method is a path to the database file, from which the database should be loaded. By default, the value of this parameter is `NULL` and the database is not loaded from disk. If the parameter is not `NULL`, then the `open()` method tries to load the database from the specified file and initializes an empty database if the file doesn't exist. There is also a `McoSqlDatabase::save()` method for saving a database in the specified file.

So to load a database from an image previously saved in a file, you must specify the last parameter of the open method, and to save a database image prior to closing the database, insert a call to the `save()` method before the database is closed.

### Function-Based Group By

You can use *eXtremeSQL* built-in functions and user-defined functions in the *group by* clause of a `SELECT` statement. Here is an example of using a built-in function:

```
class events {
    autoid;
    int datetimestamp; // Number of seconds since epoch
    u_char event_name;
};

"Select event_name, date_format(datetimestamp, "YYYY-MM") from events
group by date_format(datetimestamp, "YYYY-MM");
```

Here is an example of defining and using a user-defined function:

```
// here is the body of the UDF:

static Value* mod(Value* a, Value* b) {
    if (a->isNull() || b->isNull()) {
        return NULL;
    }
    return new IntValue(a->intValue() % b->intValue());
}

// below, f1 is an instance of the SqlFunctionDeclaration class. The
// constructor links this declaration of a UDF named "mod" to the list
// of all UDFs maintained internally by eXtremeSQL:

static SqlFunctionDeclaration f1(
    tpInt,          // tpInt is the return value of the UDF
    "mod",         // the name of the function as we'll use it in
a query
    (void*)mod,    // the function pointer
    2              // the number of arguments to the UDF
);

// below, we use the UDF 'mod' that was setup by the steps above:

select * from T where mod(code,3) = 0;
```

This request selects records where the value of the field named **code** is divisible by 3.

## C Structs

There is a mechanism of extracting and storing data using C structs. First, define a C struct corresponding to the class definition in an *eXtremeDB* data definition file (MCO file) and then fetch a class or table's data from the database using one operation. It looks like this:

```
MyRecord r;
QueryResult result(engine.executeQuery("select * from MyRecord where
i4=%i", i));
Cursor* cursor = result->records();
Record* rec = cursor->next();
result->extract(rec, &r, sizeof(r));
```

It is possible to insert an entire class (table):

```
MyRecord r;
r.i4 = i;
... // assign the rest of MyRecord r fields
engine.executeStatement("insert into MyRecord %r", &r);
```

Currently, the schema compiler does not generate corresponding C structs for class/table definitions and you must do this yourself. The following table specifies the C struct type to be used for each *eXtremeDB* field type:

<i>eXtremeDB</i> FIELD TYPE	C STRUCT TYPE
signed<1>	char
signed<2>	short
signed<4>	int
signed<8>	int64_t
unsigned<1>	unsigned char
unsigned<2>	unsigned short
unsigned<4>	unsigned int
unsigned<8>	int64_t
float	float
double	double
string	char*
nstring	wchar_t*
vector	Array*
array	Array*
struct	Struct*
autoid_t	Reference*
time	time_t
date	time_t

## ***Initializing/Shutting Down from a C Application***

This sample code is taken from the `samples/sql/ctest` directory of the *eXtremeSQL* SDK:

```

{
  database_t engine;
  mco_db_h dbh;

  mco_runtime_start();
  /* Create a database*/
  mco_db_open(DATABASE_NAME, ctestdb_get_dictionary(), start_addr,
             DATABASE_SIZE, PAGE_SIZE));

  /* connect to the database, obtain a database handle */
  mco_db_connect(DATABASE_NAME, &dbh);

  /* Set memory allocator */
  mcosql_initialize_dynamic_memory_manager(&malloc, &free,
                                           ALLOC_QUANTUM, ALLOC_RETAIN);

  /* Initialize eXtremeDB SQL mapper */
  mcoapi_initialize(dbh, ctestdb_get_dictionary() );

  /* Open QSL engine */
  mcosql_open(&engine);

  /* Work with database */
  do_test(engine);

  /* Close SQL engine */
  mcosql_close(engine);

  /* Close eXtremeDB database */
  mco_db_close(DATABASE_NAME);

  /* shutdown database engine */
  mco_runtime_stop();
}

```

Note that `mcosql_open()` should be passed as the **address** of the engine handler, not the value, i.e.,

```

mcosql_open (&engine);

```

instead of:

```

mcosql_open (engine);

```

## ***Initializing/Shutting Down from a C++ Application***

### **Initializing**

```

{
  McoSqlEngine engine;

  engine.open("testperfdb", testperfdb_get_dictionary(),
             DATABASE_SIZE, PAGE_SIZE);
}

```

The prototype of `McoSqlEngine::open` is:

```
open(char const* name,
     mco_dictionary_h dictionary,
     size_t size,
     size_t pageSize = 128,
     void* mapAddress = (void*)0x20000000,
     size_t maxTransSize = 0,
     int flags = ALLOCATE_MEMORY |
                SET_ERROR_HANDLER |
                START_MCO_RUNTIME |
                SET_SQL_ALLOCATOR |
                INITIALIZE_DATABASE,
     char const* databaseFile = NULL);
```

Internally, `McoSqlEngine::open()` does the following:

1. Checks for local or shared memory support.
2. If local memory and `(flags & ALLOCATE_MEMORY)` is true, then calls `malloc()` to allocate `DATABASE_SIZE` bytes.
3. Sets the *eXtreme*DB error handler to `McoDatabase::checkStatus`.
4. Initializes the *eXtreme*DB run-time.
5. Sets the McoSQL memory allocator.
6. Sets the maximal transaction size.
7. Connects to or opens the *eXtreme*DB database.
8. Opens the SQL engine.

### Notes on the Memory Allocator

- For `McoSqlEngine`, the class `DynamicAllocator` is used by default. For `McoMultithreadedSqlEngine`, the `McoSql::MultithreadedAllocator` is the default. The difference between them is that `McoSql::MultithreadedAllocator` maintains separate memory segments for each thread. For the rest of this discussion, references to `DynamicAllocator` also apply to `MultiThreadedAllocator`.
- `DynamicAllocator` uses the standard (unless overwritten) `malloc/free` runtime functions for allocating memory segments. The size of the segment can be specified by a programmer; by default the size is 1MB. The segment is treated as a stack, thus SQL allocates space in it in LIFO order. When the current segment is exhausted (all the space is allocated), a new segment is allocated and the allocator uses this new segment. All segments are linked into a link-list. When segments becomes empty they are released (freed) and the allocator returns to the previous segment.

The `DynamicAllocator` does not use the *eXtreme*DB memory pool. But it is possible to derive a class from `DynamicAllocator` that would use the `mco_malloc()` and `mco_free()` functions to allocate segments.

The amount of memory that the allocator grabs depends on each SQL statement. The requirement can vary from several kilobytes to several megabytes (that much memory can be necessary to process queries that select large number of records and, for example, sort them).

The allocator pointed to by the `MemoryManager::allocator` is used for every memory allocation required by *eXtremeSQL*. The *eXtremeSQL* library also implements a so called `StaticAllocator`. In contrast to the `DynamicAllocator` that grabs new segments when necessary and maintains a list of them, the `StaticAllocator` is given a memory pool that is never extended.

The application can install this allocator by calling the `MemoryManager::setAllocator()`:

```
MemoryManager::setAllocator(new StaticAllocator(mem, memSize));
```

and call `McoSqlEngine.open()` without the `SET_SQL_ALLOCATOR` flag.

`MemoryManager::setAllocator()` is a static method. You do not need to have an instance of this class. You can just call it as in the example above.

### Shutting Down

```
{  
engine.close();  
}
```

Internally, `McoSqlEngine::close()` does the following:

1. Releases the memory allocator.
2. Closes the database (`mco_db_disconnect()` & `mco_db_close()`).
3. Releases the memory, if it was allocated during initialization.
4. Terminates the *eXtremeDB* run-time.

### *Preparing SQL statements for repetitive executions*

```
McoSqlSession session(engine);  
int bid, tid, aid, delta;  
PreparedStatement stmt[2];  
session.prepare(stmt[0], "UPDATE accounts SET Abalance=Abalance+*i  
WHERE Aid=*i", &delta, &aid);  
session.prepare(stmt[1], "SELECT Abalance FROM accounts WHERE  
Aid=*i", &aid);  
session.executePreparedStatement(stmt[0]); // insert, update/delete  
DataSource* result = session.executePreparedQuery(stmt[1]);  
Cursor* cursor = result->records();  
int aBalance = 0;  
  
while (cursor->hasNext()) {  
    aBalance = (int)cursor->next()->get(0)->intValue();  
}  
result->release();
```

## ***Interactive SQL (XSQL utility)***

The XSQL utility is an interactive tool for processing SQL statements.

### **Standard and *eXtremeDB*-Specific Commands**

In addition to the standard commands: help, exit, trace (on|off), format (HTML, XML, TEXT), report and save <file>, XSQL can produce query output in XML and HTML formats.

COMMAND	DESCRIPTION
help	Displays this list of commands
exit	Ends the XSQL session
trace <on off>	Causes XSQL to display the query execution plan
format <HTML XML TEXT>	Changes the output format of XSQL
Report	Generates a report of database statistics (see example below)
save <file>	Equivalent to the mc_db_save() function, it save a copy of the in-memory database to the specified file
<statement>	Any valid <i>eXtremeSQL</i> statement (i.e., SELECT, UPDATE, INSERT, DELETE)

### **Example of Report Output**

```
Database size 15976Kb
Hash Index (Album.by_seq): 300 objects, 6Kb
Tree Index (Album.by_name): 300 objects, 6Kb, height=3
Tree Index (Album.by_artist): 300 objects, 6Kb, height=3
Hash Index (Artist.by_seq): 100 objects, 2Kb
Tree Index (Artist.by_name): 100 objects, 2Kb, height=2
Hash Index (Genre.by_seq): 0 objects, 0Kb
Tree Index (Genre.by_name): 0 objects, 0Kb
Tree Index (Playlist.by_name): 0 objects, 0Kb
Hash Index (Playlist.by_nbr): 0 objects, 0Kb
Hash Index (Track.by_seq): 3000 objects, 69Kb
Tree Index (Track.by_name): 3000 objects, 62Kb, height=4
Tree Index (Track.by_album): 3000 objects, 62Kb, height=4
List (counter.list_index_): 1 objects, 0Kb
Tree Index (track_playlist.by_track): 0 objects, 0Kb
Tree Index (track_playlist.by_playlist): 0 objects, 0Kb
Database Instance (): 0 objects, 1020Kb
```

# Chapter 4: *eXtremeSQL* Query Optimization

As mentioned in Chapter 1, finding the optimal plan for execution of SQL statements is a very complex and challenging task. SQL optimizers analyze SQL queries sent to the database and select the best search strategies for accessing the database.

There are two types of optimizers: **cost-based and rule-based** (*eXtremeSQL* uses a rule-based optimizer). Cost-based optimization greatly depends on the distribution of data in tables involved in queries, so RDBMS that employ cost-based optimizers keep statistics about the number of records in each table, selectivity of keys, etc. These statistics can be evaluated during query optimization to weigh the relative cost of possible execution plans, but doing so requires a lot of CPU time. CPU time is also required during other aspects of RDBMS operation for gathering the statistics. These uses of CPU cycles run counter to the goal of real-time systems and consume a resource that is not in great supply in many embedded systems. Further, it is impossible to predict which execution plan will be used by the RDBMS for a query from one execution to another as the distribution of data changes dynamically, changing the calculated execution steps' weight. In summary, cost-based optimizers consume CPU cycles and are inherently unpredictable in two ways: it cannot be predicted how long the optimizer will require generating the final execution plan, and the execution plan itself cannot be predicted – hence, the performance of the query execution cannot be predicted.

Rule-based optimizers, like the kind *eXtremeSQL* uses, do not consider the cost of execution plan steps. Rather, they iterate over a finite and well known set of rules that determine how the query will be executed. Because the set of rules do not change dynamically like data distribution statistics do, a rule-based optimizer spends a predictable amount of time formulating the execution plan, and the execution plan does not change from one execution of the query to the next. In summary, the problems posed by cost-based optimizers in embedded/real-time systems are overcome: CPU cycles are not used gathering statistics, the optimization phase of query execution becomes predictable, and the execution plan isn't subject to change, further lending predictability.

*eXtremeSQL* uses a **rule-based optimizer** because it is not practical for embedded systems to spend time and memory for collecting statistic and finding the optimal execution plan based on cost. Also, embedded systems rarely use very complex database queries such as are found in enterprise uses, like data mining. The main goals of *eXtremeSQL* are to provide: 1) predictable and fast execution of relatively simple queries, and 2) the developer with the possibility to easily tune the query and indirectly specify the optimal execution plan for it by arranging the tables and filters according to simple rules.

*eXtremeSQL* query optimization is based on set of rules. Optimization is mostly based on using *indices*, and avoiding sequential table scans whenever possible. Below is the set of rules used by optimizer that you can use to guide your phrasing of SQL statements:

1. Tables specified in the FROM clause of the SELECT statement are taken in the order in which they are specified in the query, from left to right. Each table is assigned a number representing its position in the FROM list (starting from 1). So, the order of table joins is unambiguously specified by the order of the tables in the FROM clause, which allows the programmer to easily choose the optimal order of joins.

To choose the optimal order of joins, consider the following rules:

- If there is no filter condition for records of two tables to be joined, or the conditions cannot be evaluated using indexes (discussed later), then list the smaller table first.
  - In the presence of filters that can be evaluated using indexes, the goal is to list the table with the filter having the greatest selectivity, i.e. which returns the fewest number of rows regardless of the size of the underlying table. In other words, the table can be very large but if the condition has the form “ $x = 1$ ” where  $x$  is a primary key, then we know that only one record from the table will be selected.
2. The search predicate is split into the set of conjuncts (expressions combined by the AND operator) and then the conjuncts are sorted according to their weight. The weight of a conjunct is derived from the number assigned to the field’s table and a “penalty” if an index can not be used for retrieving records matching this condition. In other words, conjuncts are sorted in such a way that the expressions accessing the tables listed first in the FROM clause are checked first. Literals are considered to belong to the table numbered 0. The formula for calculating a conjunct’s weight is the following: **(table\_no \* 2 + index\_is\_not\_applicable)**. First applying the conditions that can be evaluated using an index allows *eXtremeSQL* to avoid sequential searches of tables whenever possible.

Now it is necessary to clarify when a sequential search can be replaced with an index search. Below is a list of rules specifying when an index is applicable:

1. Binary logical OR expressions can be evaluated using an index if both of its operands can be evaluated using index. In this case, interim results produced by both index searches are merged.
2. Binary logical AND expressions can be evaluated using an index if one of its operands can be evaluated using an index. If both operands also can be evaluated using an index, then *eXtremeSQL* performs two index searches and intersects their results. Otherwise the first operand is evaluated using an index with the second operand used as a filter (all records selected using the index will be filtered using the condition represented by the second operand).

For example, suppose we have the conditions:

```
WHERE salary > 5000
AND state = "WA"
```

If salary has an index and state does not, then the conjunct 'salary > 5000' will be evaluated using the index and each row will be filtered by the right-side conjunct state = "WA."

However, if state has an index and salary does not, the formula expressed above will reorder the conjuncts so that state is evaluated with its index first and each row will be filtered by 'salary > 5000.'

Finally, if both salary and state have an index, then two index searches are conducted and the intersection of the results is returned.

3. A comparison expression (< > <= >= =) can be evaluated using an index if one of its operands is an **indexable expression** (definition of **indexable expression** is specified below) and another is a **known value** (definition of known value is specified below). In the case of an equality comparison both hash and B-Tree indices can be used, but in the case of a relational comparison (< > <= >=) only a B-Tree index can be used.
4. A BETWEEN expression can be evaluated using an index if its first operand is an **indexable expression** using a B-Tree index (a hash index can not be used in this case) and the second and third operands (boundaries) are **known values**.
5. A LIKE expression can be evaluated using an index if its first operand is an **indexable expression** using B-Tree index (a hash index can not be used in this case) and the second operand (pattern) and optional third operand (escape character) are **known values**. If the pattern value contains a wildcard character (% or \_) in the first position it is not an **indexable expression**. The substring formed by the pattern before the first wildcard is the search prefix. *eXtremeSQL* will perform an index search to locate records with this prefix, and for each such record evaluate the LIKE expression to match the rest of the pattern.
6. An IN expression can be evaluated using an index if its first operand (selector) is an **indexable expression** and the second operand is a set of **known values**. *eXtremeSQL* will apply the index for each element of the set and join the result sets.
7. IS NULL, IS TRUE and IS FALSE expressions can be evaluated using an index if the operand is an **indexable expression**. *eXtremeSQL* can evaluate IS NULL only on reference fields, fixed length char fields and variable length string fields.
8. If an expression is a sequence of several conjuncts **C1, C2,...Cn**, where each conjunct **Ci** has the form (**Fi = known\_value**) where **Fi** is a field of the table being inspected and **CM*Pi*** is a comparison operation, and there is a compound index <**F1,F2,...,Fm**> (where  $m \geq n$ ) defined in the database schema, then this expression can be evaluated using a single search in the compound index if this compound index is a B-Tree index, or if it is a hash index and  $m == n$ .

### **indexable expression**

An expression is an **indexable expression** if:

- it is a column of the table being inspected (while performing a table join) and there is an index defined in the database schema for this field or for a set of fields (compound key) where this field is the first.

- the expression is a reference access expression (`pointer_field.referenced_expression`) where `pointer_field` is a reference field of the table being inspected and `referenced_expression` is an indexable expression and there is an index defined in the database schema for `pointer_field`. In this case, *eXtremeSQL* will first perform an index search to select records matching `referenced_expression` and then perform an index search in this table using the index for `pointer_field`, locating records in which `pointer_field` refers to one of the records in this selection.

For example, consider we have two tables, `Supplier` and `Order`, and the query:

```
Select * from Order where supplier.location.city='New York';
```

This query can be executed using an index search if:

- there is an index for the field “`location.city`” in the `Supplier` table.
- there is an index for the field “`supplier`” in the table `Order` (where “`supplier`” is a reference field having type `autoid_t` and it references the `autoid` of a `Supplier` record).
- `AUTOID` is maintained for the table `Supplier`.

If these conditions are true, then *eXtremeSQL* will first perform an index search in the table `Supplier`, selecting suppliers located in New York, and for each selected record search its `AUTOID` in the index for field “`supplier`” in the table `Order`.

### known value

An expression is a **known value** if it is a literal or a query parameter, and it is a field of a table with a smaller table number than the table being inspected.

For example, consider two tables:

```
Create table Person (pid integer primary key, name string using index);  
Create table Hobby (pid person key, description string);
```

To select all hobbies of a person, there are two possible formulations of the SQL:

GOOD example:

```
Select * from Person p, Hobby h where p.name='John Smith' and p.pid =  
h.pid;
```

BAD example:

```
Select * from Hobby h, Person p where p.name='John Smith' and p.pid =  
h.pid;
```

From the information given earlier we know that the conjuncts will be ordered as follows:

GOOD EXAMPLE	BAD EXAMPLE
p.name = "John Smith"	p.pid = h.pid
p.pid = h.pid	p.name = "John Smith"

In the GOOD example, an index search for p.name = "John Smith" will be evaluated first, then an index join to select its hobby/hobbies.

In the BAD example, a sequential scan of Hobby will be processed and for each record find the related Person using indexed join, and then check that name of this person is John Smith.

**Distinct**

When the DISTINCT qualifier is in the query, then after applying all possible optimizations, all selected tuples are sorted by all columns and duplicates are removed. If the ORDER BY or GROUP BY clause is present in the statement together with the DISTINCT qualifier, then the fields in the ORDER BY or GROUP BY are compared first during sorting, so the sort operation is performed only once.

*eXtremeSQL* uses the Quicksort algorithm for sorting records. *eXtremeSQL* first extracts the sort keys into a separate array (or part of the key in case of strings), then sorts this array, and finally refines the order by performing a comparison of all columns mentioned in the ORDER BY list.

**Order By**

When a B-Tree index is used to select records, selection is automatically sorted by the key corresponding to this index. If there is an explicit ORDER BY clause in this statement, sorting records by this key, then *eXtremeSQL* does not perform extra sorting, since the records selected are already in the requested order. The direction (ascending/descending) of the ORDER BY clause should be the same as the direction specified for this field in the index.

**Subquery**

*eXtremeSQL* optimizes the execution of subqueries by checking the dependencies of the subquery expression. The result returned by the subquery execution is saved and only recalculated if the subquery expression refers to the fields from the enclosing scope.

**Show Plan**

To view the query execution plan, invoke the trace(true) method of the SqlEngine class. Then, during execution of each statement you will see a dump of the query and execution of indexed and sequential searches. Please note that *eXtremeSQL* doesn't store query execution plans and the decision whether to apply an index or not is taken when the query is executed. In other words, *eXtremeSQL* has no precompiled queries. Making decisions about applying indices during query execution time allows taking in account the actual values of searched operands.

For example, use an index search in query:

```
db.executeQuery("select * from T where name like %s", "John%");
```

and use a sequential search in query:

```
db.executeQuery("select * from T where name like %s", "%John%");
```

A precompiled execution plan would have insufficient information to make this judgment.

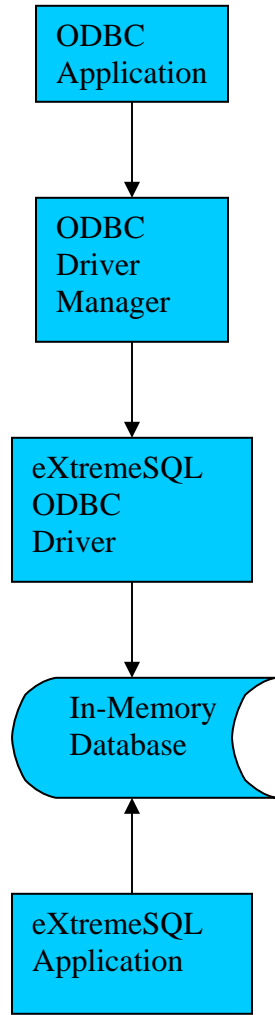
# Chapter 5: McObject ODBC Driver

This chapter describes the eXtremeSQL ODBC Driver implementation and how to use the ODBC driver to connect to an existing eXtremeSQL database.

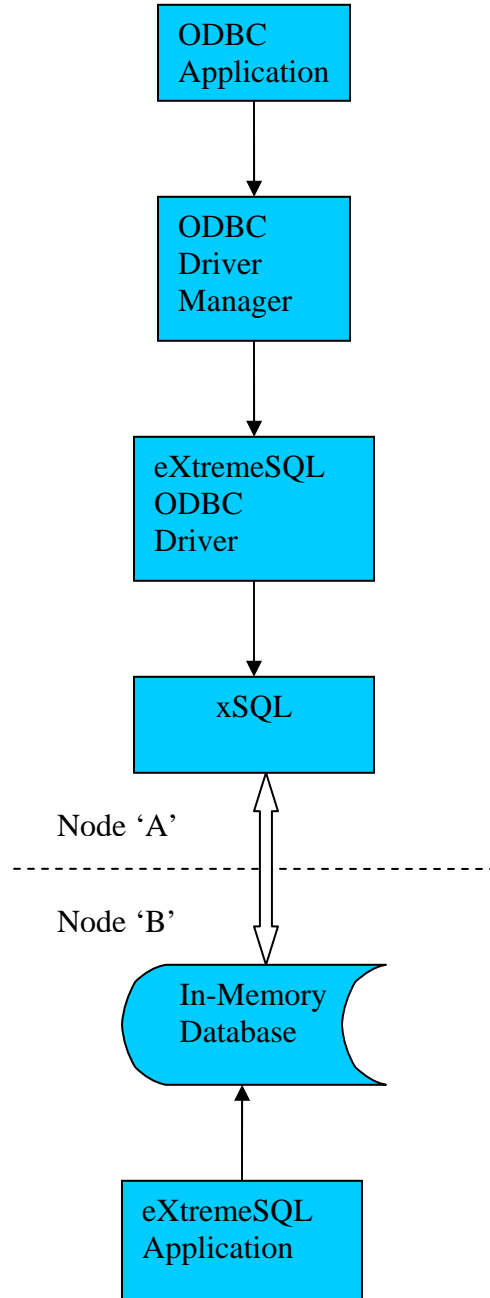
The objective of the eXtremeSQL ODBC Driver is to allow ODBC client applications to gain access to a standard eXtremeSQL application. It is not intended, and it is not possible, to create a stand-alone ODBC application. eXtremeSQL are in-memory, embedded databases; the database dictionary that describes the structure of the database is encoded in the application code by virtue of linking the schema compiler –generated C file. Further, eXtremeSQL databases don't exist until they are created by the application, unlike on-disk databases that persist across executions of the application and which store the database dictionary with the on-disk database files.

The eXtremeSQL ODBC Driver can be used to connect to either a local database (an in-memory database created on the same machine by a standard eXtremeDB or eXtremeSQL application), or a remote database through the eXtremeSQL xSQL program. From this point, “eXtremeSQL” will be used to refer generically to eXtremeDB or eXtremeSQL.

ODBC access to a local in-memory database



ODBC access to a remote in-memory database



## Installing and configuring the eXtremeSQL ODBC Driver

The target system needs to have the ODBC Driver Manager installed, which is part of the MDAC (Microsoft Data Access Components) available here:

<http://msdn2.microsoft.com/en-us/data/aa937730.aspx> but which should already be installed on Windows systems.

Open a DOS CMD.EXE window and change to the odbc/bin directory of the eXtremeSQL ODBC installation. Register the DLL:

```
Regsvr32 .\mcoodbcsetup.dll
```

```

C:\WINDOWS\system32\cmd.exe
C:\McObject>cd demos
C:\McObject\demos>cd odbc
C:\McObject\demos\odbc>dir
Volume in drive C has no label.
Volume Serial Number is FC13-861D

Directory of C:\McObject\demos\odbc

03/09/2007  01:43 PM    <DIR>          .
03/09/2007  01:43 PM    <DIR>          ..
03/09/2007  01:42 PM             323,584 mcoodbc.dll
03/09/2007  01:42 PM             49,152 mcoodbcsetup.dll
03/09/2007  01:42 PM             28,672 mcsetupdlg.dll
03/09/2007  12:25 PM    <DIR>          simple
03/09/2007  12:25 PM    <DIR>          simpleodbc
03/09/2007  12:25 PM    <DIR>          tpc
03/09/2007  12:25 PM    <DIR>          tpcodbc
               3 File(s)          401,408 bytes
               6 Dir(s)    13,006,557,184 bytes free

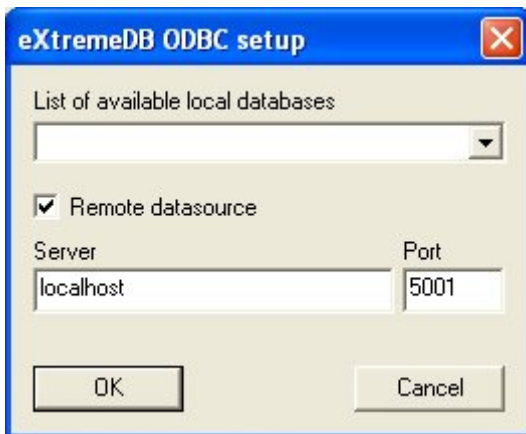
C:\McObject\demos\odbc>regsvr32 .\mcoodbcsetup.dll
C:\McObject\demos\odbc>

```

Next, open up the Windows Control Panel, double-click on Administrative Tools, then open Data Source (ODBC). In the User DSN tab, click the “Add...” button and you should now see the McObject ODBC driver displayed.

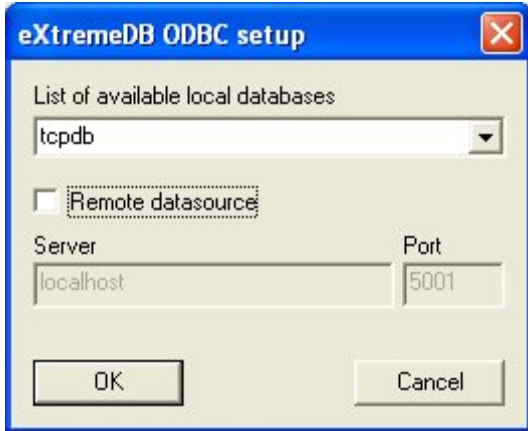


Click Finish. This will bring up the McObject ODBC Setup dialog. Here you can set up ODBC access for a local or a remote database (see below).



### Connection to a local database

In this case, uncheck the check box in the setup dialog and type the name of the database as it is passed to the `engine.open()` or `mco_db_open()` function calls. In the example below it is “tpcdb”.



Naturally, the eXtremeSQL database must be created in shared memory (not local memory) so that it can be accessed by more than one application (at a minimum, the eXtremeSQL application and the eXtremeSQL ODBC driver).

To connect to the database, specify the eXtremeSQL database name as the first parameter of `SQLConnect`:

```
SQLRETURN SQL_API SQLConnect( SQLHDBC hDbc,
                               SQLCHAR *serverName,
                               SQLSMALLINT nameLength1,
                               SQLCHAR *userName,
                               SQLSMALLINT nameLength2,
                               SQLCHAR *authentication,
                               SQLSMALLINT nameLength3 )
```

'nameLength1' should be either the length of the 'serverName' argument, or `SQL_NTS`. The parameters after nameLength1 are ignored since eXtremeSQL is an embedded database and doesn't have a concept of "users".

The 'serverName' is the same name that was given to the eXtremeSQL API function `engine.open()` or the eXtremeDB API function `mco_db_open`. So, for the TPC example program, the database is opened as

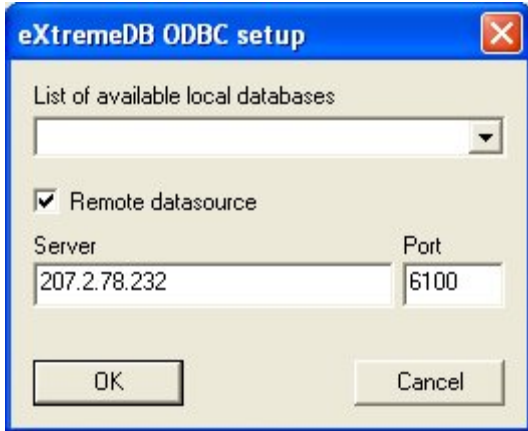
```
engine.open("tpcdb",
            tpcdb_get_dictionary(),
            DATABASE_SIZE,
            PAGE_SIZE,
            MAP_ADDRESS)
```

and therefore the `SQLConnect` would be

```
SQLConnect( hDbc, "tpcdb", SQL_NTS, 0, 0, 0, 0)
```

### Connection to a remote database

In this case, leave the check box checked in the ODBC setup dialog and type the IP address of the system that is hosting the database, and the port number on which the xSQL process is listening, as described below.



Connection to the remote database is done through xSQL, which plays the role of the database server. Refer to the block diagram above.

First, we need to start the application on the host computer that will create the database. Let's assume the IP address of the host computer is 207.2.78.232. We'll use the SDK example program 'tpcodbc':

```
tpc -clients 1 -tpc 1000000
```

The tpc program creates the database as follows:

```
engine->open("tpcdb",
             tpcdb_get_dictionary(),
             DATABASE_SIZE,
             PAGE_SIZE,
             MAP_ADDRESS);
```

Note the name of the database, "tpcdb".

Next, on the same computer, we need to start the 'xSQL' program that will provide the remote access to the in-memory database:

```
xsql tpcdb:6100
```

This command starts the xsql program, instructs it to open the "tpcdb" database and to listen on port 6100 for remote processes wanting to connect to the database.

Finally, on another computer on the network, we start the 'tpcodbc' client program:

```
tpcodbc -tpc 1000000 -db 207.2.78.232:6100
```

This command line instructs the program to run 1,000,000 iterations of the tpc test, opening the ODBC data source name "207.2.78.232:6100" which is, of course, the xSQL process running on the host computer and listening on port 6100. The C source code for the tpcodbc establishes the ODBC connection as follows:

```

// extract the data source name from the command line
database = argv[i];
// allocate environment handle
SQLAllocEnv(&hEnv);
// allocate connection handle
SQLAllocConnect(hEnv, &hDbc);
// connect to data source (McObject ODBC driver
    SQLConnect(hDbc, database, SQL_NTS, NULL, 0, NULL, 0);

```

## McObject ODBC Implementation Details

McObject ODBC implements most of the ODBC 3.0 standard except ‘descriptors’, which are supplementary. Currently, the McObject ODBC driver returns “02.00” as the driver version.

The list below enumerates the list of supported ODBC function calls

SQL_API_SQLALLOCONNECT	SQL_API_SQLPREPARE
SQL_API_SQLALLOCENV	SQL_API_SQLEXECDIRECT
SQL_API_SQLALLOCSTMT	SQL_API_SQLEXECUTE
SQL_API_SQLFREECONNECT	SQL_API_SQLROWCOUNT
SQL_API_SQLFREEENV	SQL_API_SQLFETCH
SQL_API_SQLFREESTMT	SQL_API_SQLSETCONNECTATTR
SQL_API_SQLCOLATTRIBUTES	SQL_API_SQLFREEHANDLE
SQL_API_SQLERROR	SQL_API_SQLFREESTMT
SQL_API_SQLSETPARAM	SQL_API_SQLGETCONNECTATTR
SQL_API_SQLTRANSACT	SQL_API_SQLSETENVATTR
SQL_API_SQLSETCONNECTOPTION	SQL_API_SQLGETCURSORNAME
SQL_API_SQLGETCONNECTOPTION	SQL_API_SQLSETSTMTATTR
SQL_API_SQLENDTRAN	SQL_API_SQLGETDATA
SQL_API_SQLALLOCHANDLE	SQL_API_SQLEXTENDEDFETCH
SQL_API_SQLBINDCOL	SQL_API_SQLCOLUMNS
SQL_API_SQLCANCEL	SQL_API_SQLSPECIALCOLUMNS
SQL_API_SQLGETDIAGFIELD	SQL_API_SQLTABLES
SQL_API_SQLCLOSECURSOR	SQL_API_SQLBINDPARAMETER
SQL_API_SQLGETDIAGREC	SQL_API_SQLNATIVESQL
SQL_API_SQLCOLATTRIBUTE	SQL_API_SQLNUMPARAMS
SQL_API_SQLGETENVATTR	SQL_API_SQLPRIMARYKEYS
SQL_API_SQLCONNECT	SQL_API_SQLDESCRIBEPARAM
SQL_API_SQLGETFUNCTIONS	SQL_API_SQLDRIVERCONNECT
SQL_API_SQLGETINFO	SQL_API_SQLFOREIGNKEYS
SQL_API_SQLGETSTMTATTR	SQL_API_SQLMORERESULTS
SQL_API_SQLDESCRIBECOL	
SQL_API_SQLGETTYPEINFO	
SQL_API_SQLDISCONNECT	
SQL_API_SQLNUMRESULTCOLS	

The following list enumerates the SQL functions that are not implemented in the current McObject ODBC driver:

```
SQL_API_SQLGETDESCFIELD
SQL_API_SQLGETDESCREC
SQL_API_SQLCOPYDESC
SQL_API_SQLDATASOURCES
SQL_API_SQLDRIVERS
SQL_API_SQLPARAMDATA
SQL_API_SQLPUTDATA
SQL_API_SQLFETCHSCROLL
SQL_API_SQLSETCURSORNAME
SQL_API_SQLSETDESCFIELD
SQL_API_SQLSETDESCREC
SQL_API_SQLSETSCROLLOPTIONS
SQL_API_SQLSTATISTICS
SQL_API_SQLBROWSECONNECT
SQL_API_SQLBULKOPERATIONS
SQL_API_SQLCOLUMNPRIVILEGES
SQL_API_SQLPROCEDURECOLUMNS
SQL_API_SQLPROCEDURES
SQL_API_SQLSETPOS
SQL_API_SQLTABLEPRIVILEGES
```

Each thread of a McObject ODBC process must use its own ODBC connection. That is to say, that the SQLHDBC instance (variable) cannot be shared by threads.

### **McObject ODBC and eXtremeSQL proprietary API Together**

Note that it is not necessary for a local application to use the ODBC Driver Manager. The McObject ODBC API library can be linked directly in the project.

To use the ODBC API, follow these steps:

Open the eXtremeSQL database engine (C++ API) in the standard way.

```
McoMultithreadedSqlEngine engine;
engine.open("simplifiedb",
           simplifiedb_get_dictionary(),
           DATABASE_SIZE,
           PAGE_SIZE,
           MAP_ADDRESS);
```

Then invoke the SQLConnect function in this way:

```
SQLConnect(hDbc,  
           (SQLCHAR*)&engine,  
           0, NULL, 0, NULL, 0);
```

Passing 0 (zero) as the value of the third parameter (NameLength1) informs the McObject ODBC API that it should treat the second parameter (ServerName) not as a data source name, but as a pointer to the engine class instance (object). Now it is possible to access the database both from the ODBC and the native eXtremeSQL APIs.

The 'simpleodbc' example program illustrates the implementation.

# Chapter 6: SQL Meta Data

A database schema defined using either the eXtremeDB proprietary data definition language (DDL) or the eXtremeSQL 'create table' syntax is processed by a schema compiler (mcoomp or sql2mco, respectively) that, among other things, creates a binary form of the schema in the generated *dbname.c* file. The binary form of the schema is called a *database dictionary*.

Beginning in version 3.1 of eXtremeSQL, the database dictionary can be queried through a system table called *Metatable*. The table is defined as follows:

Column Name	Column Type	Description
TableName	String	The name of the table (class)
FieldNo	int2	The field number in the class (1-based)
FieldName	string	Field (column) name
FieldTypeName	string	Field type name
FieldType	int4	Field type code (see enum from apidef.h below)
FieldSize	int4	Size of the field (0 for variable length field types)

```
enum Type {
    tpNull,
    tpBool,
    tpInt1,
    tpUInt1,
    tpInt2,
    tpUInt2,
    tpInt4,
    tpUInt4,
    tpInt8,
    tpUInt8,
    tpReal4,
    tpReal8,
    tpDateTime, // time_t
    tpUnicode,
    tpString,
    tpRaw,
    tpReference,
    tpArray,
    tpStruct,
    tpBlob,
    tpDataSource,
    tpList,
    tpInt = tpInt8,
    tpReal = tpReal8,
    tpLast
};
```

Example queries that can be used to extract database dictionary information from Metatable follow.

1. List all tables containing the field "foo":

```
SELECT DISTINCT TableName
FROM Metatable
WHERE FieldName='foo';
```

2. Get a list of all field (column) names for a class (table):

```
SELECT FieldName
FROM Metatable
WHERE TableName='abc'
ORDER BY FieldNo;
```

3. Get a list of all classes (tables) in the database

```
SELECT DISTINCT TableName
FROM Metatable;
```

4. Get a list of all fields of a specified table excluding auto-generated fields:

```
SELECT *
FROM Metatable
WHERE TableName='abc'
AND NOT AutoGenerated
ORDER BY FieldNo;
```

5. Fetch all tables and fields that reference a specified table:

```
SELECT TableName, FieldName
FROM Metatable
WHERE ReferencedTable='xyz';
```

# Appendices

## *eXtremeSQL* Namespace and Classes

There is one namespace for *eXtremeSQL*: `McoSql`. There are about 70 classes associated with the namespace `McoSql`. All classes, structs, unions and interfaces used in *eXtremeSQL* are fully described in the reference manual.

# Appendix A: sql2mco Utility

The sql2mco utility is a simple command-line driven interactive SQL utility that converts SQL-89 DDL statements (e.g. CREATE TABLE) to the equivalent *eXtremeDB* DDL.

The sql2mco utility is invoked from the command line, in the form:

```
sql2mco <input-sql-file> <output-mco-file>
```

## ***Formal Grammar***

The following page lists the formal grammar for the sql2mco utility:

```

<create table statement> ::=
    CREATE TABLE <table name>
    ( <table element> [ { , <table element> } ... ] )

<table element> ::=
    <column definition> | <table constraint>

<column definition> ::=
    <column name> <data type> [ DEFAULT <default value> ] [ <column
constraint> [ { , <column constraint > } ... ]

<column constraint> ::=
    [ CONSTRAINT <constraint name> ] [ <constraint implementation> ]
    { <unique column constraint> | <primary key column constraint> |
    <foreign key column constraint> | <not null constraint> }

<unique column constraint> ::= UNIQUE

<primary key column constraint> ::= PRIMARY KEY

<foreign key column constraint> ::= REFERENCES <table name> [ ( <column
name> ) ]

<not null constraint> ::= NOT NULL

<table constraint> ::=
    [ CONSTRAINT <constraint name> ] [ <constraint implementation> ]
    { <unique table constraint> | <primary key table constraint> |
    <foreign key table constraint> }

<unique table constraint> ::= UNIQUE ( <column name> [ { , <column
name> } ... ] )

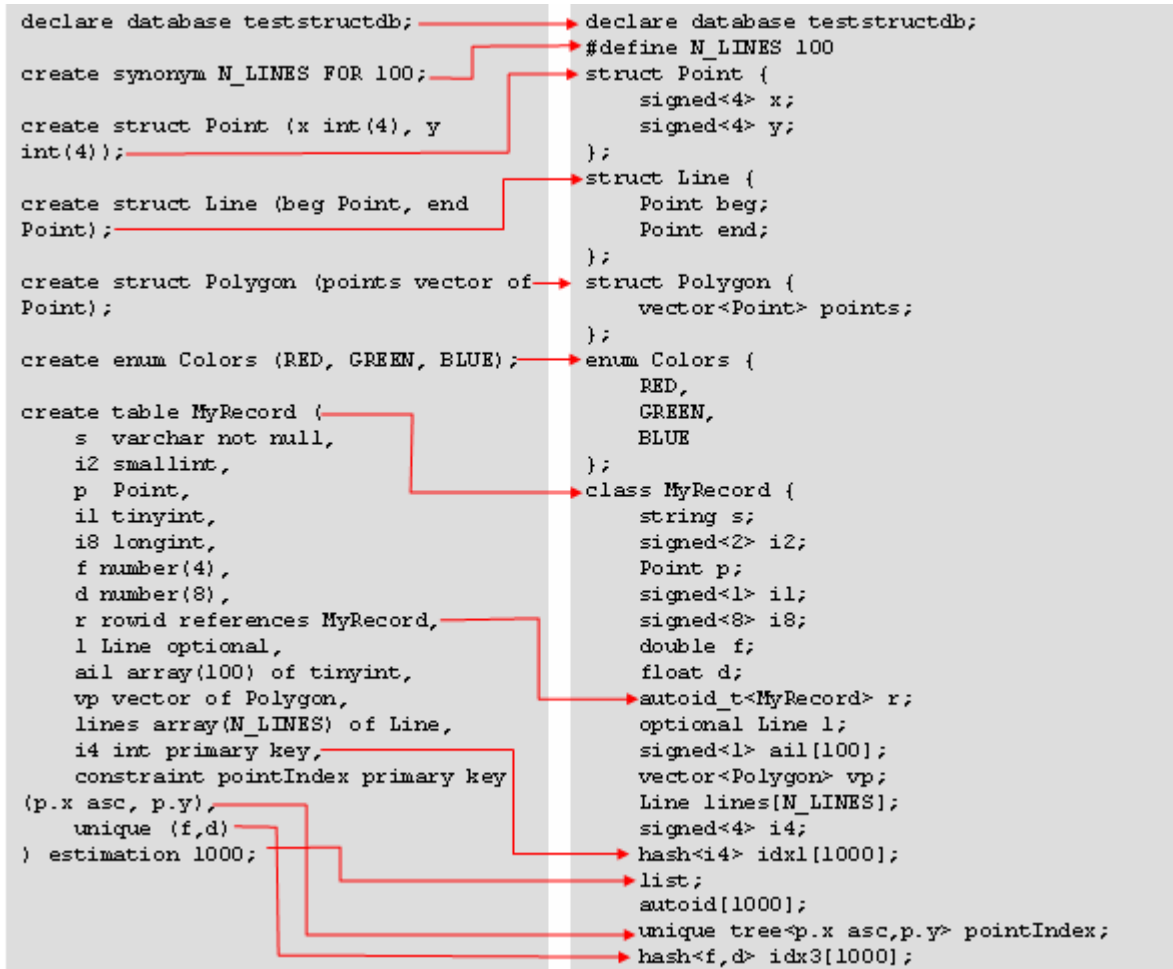
<primary key table constraint> ::= PRIMARY KEY ( <column name> [ { ,
<column name> } ... ] )

<foreign key table constraint> ::= FOREIGN KEY ( <column name> [ { ,
<column name> } ... ] )
                                REFERENCES <table name> [ ( <column
name> [ { , <column name> } ... ] ) ]

<constraint implementation> ::=
    { HASH ( <capacity> ) | TREE } INDEX

```

Here is a table that shows an example of an SQL DDL and the equivalent MCO DDL that it generates.



When using the *eXtremeSQL* programming interface for *eXtremeDB*, you have the choice of defining the database schema (i.e., the data definition language) in either the native *eXtremeDB* grammar or in SQL grammar.

We created certain extensions to standard SQL DDL grammar to support *eXtremeDB*-specific features, such as the choice between hash index and tree index, vectors, arrays, optional fields, and autoid and reference field types. Please refer to the grammar and the examples for more specific information.

Whether you use the SQL DDL to generate the *eXtremeDB* native DDL, or you start from *eXtremeDB* native DDL, be sure you compile the *eXtremeDB* native DDL with the copy of MCOCOMP that was installed with your *eXtremeSQL* SDK and that you pass the “-sql” flag to the compiler, e.g.,

```
mcocomp -sql simple.mco
```

This version of MCOCOMP inserts certain additional information that *eXtremeSQL* requires into the *eXtremeDB* run-time database dictionary.

# Appendix B: XSQL - Interactive SQL Utility

XSQL is a simple command-line driven interactive SQL utility you can use to test SQL statements without having to write an *eXtremeSQL* program first, and to experiment with different ways of expressing the query to influence the rule-based optimizer and then view the effect on the chosen execution plan and repeat as necessary until the desired execution plan is generated.

The format of the XSQL command line is:

```
XSQL DATABASE_NAME {SQL_FILE}
```

The database must already exist in shared memory in which case XSQL can open and connect to it. So, one way of using XSQL is to write a program to populate a database, and then cause the program to wait on keyboard input. Meanwhile, invoke XSQL and experiment with different query formulations.

Alternatively, you can use XSQL to populate the database by creating a SQL\_FILE, a text file containing SQL statements, and include as many SQL INSERT statements as are needed. Note that you can use XSQL to save a database that can be loaded in a later session, which may be faster than re-executing many SQL statements to create an initial state for testing.

<b>&lt;sql-statement&gt;';'</b>	<b>any valid SQL statement</b>
format (TEXT HTML XML)	Select query output format
save <file-path>	Save database image to specified file
input (<file> console)	Redirect input from specified file
exit	Terminate interactive SQL session
help	Print supported commands
trace (on off)	Toggle query execution trace
output (<file> console)	Redirect output to specified file
report	Show database statistics