

## **SQLBase Performance**

By Raj Parmar  
Sales Engineer

**September 2004**

**GUPTA™**

## Table of Contents

<b>Introduction .....</b>	<b>4</b>
<b>Physical organization and structure .....</b>	<b>4</b>
Operating System, disk and network settings.....	4
What makes up the SQLBase database and recovery files? .....	5
Database size.....	5
Cache pages and check pointing.....	5
Cache, Sortcache, and Batchpriority .....	6
Cache .....	6
Sortcache.....	6
Batchpriority (Windows only) .....	7
PCTFREE.....	7
Extent pages .....	7
Row sizes.....	7
Achieving row-level locking.....	8
Log file size and preallocation.....	8
Optimizer and statistics .....	9
Forcing Statistics .....	9
REORGANIZE procedure.....	10
What about Bulk Operations?.....	11
Locking the Database.....	11
Timeout mechanism.....	11
Precompiled queries in the database .....	12
Types of Procedures.....	13
External Functions .....	13
Triggers .....	13
Trace statement.....	14
<b>Tools and Tuneable settings .....</b>	<b>14</b>
Monitoring Tools .....	14
SQLConsole.....	14
SQL.INI settings .....	15
groupcommit.....	15
groupcommitdelay .....	15
Sortcache.....	16
optimizefirstfetch.....	16
logfileprealloc.....	16
inmessage .....	16
outmessage .....	16
Fetchthrough.....	17

## Application specific considerations.....17

Query performance.....	18
Indexes.....	19
Why use indexes? .....	19
Clustered Hashed Indexes.....	20
Composite indexes.....	20
Choosing an Index.....	20
Transaction semantics.....	21
AUTOCOMMIT .....	21
Cursor Context Preservation (CCP).....	22
Locking .....	22
SQLBase Isolation Levels .....	22
Read Repeatability (RR) .....	23
Cursor Stability (CS).....	23
Read-Only (RO).....	23
Release Locks (RL) .....	24
Locking strategies .....	24
Result Set Mode.....	26

## Introduction

The paper serves to give the reader some information on considerations that affect SQLBase performance. While it is more than a checklist of what one should look for when assessing performance issues it does not cover ancillary related areas such as backup and recovery.

The paper is organized into main sections: Physical organization and structure, Tuneable settings and Application specific considerations.

## Physical organization and structure

We will concentrate on how the data is stored and organized, what files are used to ensure database consistency and what techniques are used to improve overall database performance in a multi-user environment. We will also look at improving performance through the use of compiled queries held on the database plus efficient use of message buffers.

### *Operating System, disk and network settings*

SQLBase will run on Windows and Linux. Let's start the discussion on what attributes of the Operating System can help us achieve better performance.

First, if the OS has multiple controllers – use ones that support multi-threaded I/O.

Check how much RAM is on the server? 512mb is recommended.

With regard to RAID controllers we have found that get the best performance by using disk striping **without** partitioning the database and letting the OS decide where to place the database and logs.

For SQLBase server on Windows, ensure that the physical server is set up as an application server and not as a file server. For NT, go into Control Panel, select network, then highlight 'server' from the selectbox. Make sure the default usage for the server is set to 'Application' rather than 'file'. For Windows 2000 and XP, go into Control Panel, double-click on Network and Dial-up Connections, right-click on the LAN (or other such network connection) and select Properties, highlight the File & Print Sharing and select Properties, then select the 'Maximize data throughput for Network Applications'.

For SQLBase server on Linux, ensure that "X" is not running.

Next, there is no substitute for good database design. No matter how much tuning of the server and OS configuration you do, the most gain will stem from the way you have designed the physical structure of the database with respect to how your application will be using the data. Application and locking strategies are covered later in this paper.

**Tunable OS  
and Controller  
settings can  
improve  
performance  
significantly**

### ***What makes up the SQLBase database and recovery files?***

The database (.dbs) and the log file(s) (.log) constitute the physical database. In addition, temporary and history files (.tmp) are used for aggregate type functions and read-only isolation levels respectively.

Increased performance can be gained by placing database, log and temp files on different drives. It is better to place these on different physical drives on the same machine rather than on a remote network file server. Organizing the specific files this way also increases the chance for a successful restore in case of a system failure.

The above strategy is not effective with a partitioned database as it uses the 'MAIN' database to hold information about specific database, log and temporary areas. Also, as stated above, if you are using RAID5, having specific files on separate drives has no effect.

### ***Database size***

The extent size for the database indicates how much the database will be extended when it runs out of room. This value is typically around 100K. With a large database with lots of growth, you might want it to expand by 10-20 MB, which can reduce disk fragmentation of the file.

### ***Cache pages and check pointing***

A page is 1024 bytes (1 Kbyte) in size. When a user reads or writes a data row or index, SQLBase checks to see if the page in which it is located is already in cache. If the page is not in cache, SQLBase reads it from disk and copies it to the cache. If the page is already in cache, SQLBase uses the copy in cache. This reduces disk input and output. It is much faster to read or write to computer memory than to read or write to a disk file.

When an implicit or explicit commit occurs, SQLBase writes a commit record to the transaction log on disk. However, the database page in cache is written back (flushed) to the database based on an LRU (least-recently used) algorithm. The information in the transaction log is enough to completely recreate the update to the database if there is a crash, so there is no need to flush the page from cache immediately after a commit.

A fuzzy *check-pointing* scheme minimizes the time to perform crash recovery. *Fuzzy check pointing* means that modified database cache pages are marked at one checkpoint and then written at the next if they have not already been written by normal cache management. A transaction log record is written which describes the active transactions at each checkpoint. Logs containing the last two checkpoint log records are pinned to the disk (not deletable) as these checkpoints are used to quickly establish a redo start point for the rollforward phase of crash recovery.

**Cache page and checkpoint settings can be tuned to maximize throughput**

A checkpoint time interval parameter governs how often a recovery checkpoint operation is done. You can set the checkpoint time interval with SQLTalk's 'SET CHECKPOINT' command or the SQL/API's *sql/set* function. The default is every minute.

Depending on the applications running on the server, a checkpoint operation may affect performance. In this case, you can increase the checkpoint interval until the desired performance is gained.

The advantages of using cache and check pointing in this way are that pages are buffered in memory with the potential of multiple processes accessing the same data and sequential reads and heavily accessed tables may be in-memory, increasing performance. This is very effective for non-fragmented tables during sequential reads.

### ***Cache, Sortcache, and Batchpriority***

#### **Cache**

Cache size increases should be done in increments with measurable goals in order to achieve maximum efficiency. It is possible that increasing cache size can lead to diminishing returns on some minimally configured machines if the computer becomes starved for memory. If this setting is too low there will be more 'real' reads from the disk.

You need to experiment and find what works best for your server application mix!

Remember that 'the optimum' may change as new applications and processes are installed later utilizing more memory. You should keep an eye on potential problems before they become real problems.

Use SQLConsole and SQLPerformance tools to monitor physical I/O and cache-hit ratios for cache efficiency.

Suggested Cache settings are one fourth of the physical memory.

#### **Sortcache**

The sortcache is used when creating indexes and when any sorting of the retrieved data is done, i.e. when doing 'select' statements using 'order by', 'group by' or 'distinct'. It is used on a per user basis when it's needed and then the memory is released.

Since this is a per-user setting care should be taken not to set this value too high as it can create memory-starved situations.

SQLBase will dynamically adjust this setting up to 10MB when necessary and as such it is recommended that the default of 2000 be used.

**Cache and Sortcache settings are dependant on types transactions and may be set to minimize number of physical I/O's**

## **Batchpriority (Windows only)**

SQLBase on Windows operates at a higher than normal thread priority. For Server configurations, this ensures the best performance. However on machines that are used for more than just a database server, this can cause CPU starving for other applications running on the machine. It is recommended that batchpriority be enabled (batchpriority=1)

## ***PCTFREE***

This is the percent of each row, which is reserved for increase in row data. Used when rows are updated with longer data, NULL columns updated with non-NULL data and when new columns are added to the table. The free space is calculated for each row based on the size of the actual data, i.e. Character columns are variable length and if only 10 bytes are entered in a 30-character column, the actual size used for the calculation is 10. The free space is placed directly after the inserted row.

When increasing the row's length exceeds a row's PCTFREE area, Extent pages are created and the data is moved to the Extent page. Extent pages force additional I/O operations due to the requirement of reading/writing to two pages instead of one and can significantly affect performance.

A smaller PCTFREE value yields more rows per page and higher cache hit ratios, better performance during sequential reads and smaller database size. It does have the disadvantage of increased contention.

A large PCTFREE value yields fewer rows per page and larger database size with decreased contention. It generally also decreases performance.

If you know the data field is to increase once the row has been built it may be a good idea to increase PCTFREE.

## ***Extent pages***

When Extent pages exist, multiple reads must be done to return a complete row. This can degrade performance badly, so size PCTFREE to avoid extent pages.

## **Study table and row growth**

### ***Row sizes***

As a practical matter, you should make every effort to keep table widths small enough to fit on a single page for better performance. If the average row width is close to the usable page size, it's a good idea to split it into two tables.

If the size of a row is greater than 469 bytes, SQLBase will only be able to store one row per page. If the row size is 469 bytes or less, SQLBase will be able to store two or more rows on a page

SQLBase uses exactly as much space as needed to store a row

For example, if you define a column as char(35), but only insert a value of 3 characters, SQLBase will only allocate storage for the 3 characters.

Consider a table that has some relatively large CHAR or VARCHAR columns, in the order of 150 to 200 characters.

The table is initially populated with rows where the character columns have null values or small strings. In the future, the character columns will likely have larger values, closer to their column size. By specifying an appropriate PCTFREE parameter, you can reserve space in the table page for future row growth within the table page. This minimizes the need to allocate extent pages when a row increases in size.

### ***Achieving row-level locking***

Some database designers have created tables with the PCTFREE parameter set to 99%, to force one row per page. This can be used to achieve row level locking and reduce the possibility of deadlocks however, performance is adversely affected as SQLBase is reduced to one row per disk I/O. Potentially, a large amount of storage space is wasted. This strategy may be viable for a table with a trivial number of rows, but in general, it is to be discouraged.

### ***Log file size and preallocation***

The size of the logfiles is not dependent on the size of the database, but on the amount of log activity that you have. The maximum log file size can be set with the SQLTalk SET LOGFILESIZE command or the [sqlset](#) API function. When the log file grows past this size, SQLBase closes it and creates a new log file. The default size of the log file is 1 megabyte. The smallest size is 100,000 bytes. Once set, it will stay this size until an unload/load is done.

If you're creating 50-60 log files a day and your log files are 1 MB, you might look to setting the logfilesize to 10 MB. You must also think about how often a backup is done. When you do a backup, the log file is 'rolled over' to a new log, regardless of how much of the current log is used.

A large log file will improve database performance slightly because log files will need to be created less often. However, if the log file is too large, it wastes disk space.

By default, the current log file grows in increments of 10 percent of its current size. This uses space conservatively but can lead to a fragmented log file. The size of the log file can be pre-allocated for added performance. This will build the log file at the full size specified

**Log files are essential for Transaction Integrity and Backup. Settings can be tuned for individual servers**

on creation and not grow incrementally. To use, 'set logfileprealloc=1' in the sql.ini file.

Although you may think you have plenty of disk space, it may not be adequate to fulfil SQLBase recovery requirements. The physical disk space required for recovery is twice the size of the data required for processing. SQLBase also requires space allowance for a second recovery attempt, should the first fail.

Never delete or rename transaction log files. Depending on whether LOGBACKUP is on or off, SQLBase automatically deletes log files either when they are backed up, or when they are no longer needed for transaction rollback or crash recovery. A database (.dbs) file is useless without its associated log files.

### ***Optimizer and statistics***

For the SQLBase optimizer to make the most cost effective and efficient decisions when preparing and formulating a query it needs accurate and current statistics. Update Statistics:

- After large loads.
- When there have been significant table changes e.g. lots of inserts and deletes done.
- After an index has just been created.
- For queries that are running slower than normal.
- At least weekly for a highly used database.
- As part of the database reorganize.

It is easy to forget, so run it as a part of normal maintenance.

To check how fragmented your database is, log on to the database as sysadm user and issue the following query:

```
"select name, Percent=@round((extentpagecount/rowcount)*100,2),
      rowcount, extentpagecount
from systables
where extentpagecount>0 and system = 'N'
and rowcount > 0 order by 2 desc"
```

If a lot of tables show in your results or say (15 – 20%) of a table has extent pages then a reorganization procedure needs to be done.

Note, performance degradation could occur if there are redundant indices defined. This may make the process of selecting the best plan very lengthy and could exceed the time required to execute the chosen plan.

Update statistics places many locks both on user and system objects so it is important to commit when done.

### ***Forcing Statistics***

This is a useful way to model your database using a small database. It provides a way of studying behaviour of applications during development.

**Accurate  
statistics are the  
key to good  
query  
performance**

### Example of forcing statistics:

```
SELECT * FROM P WHERE color = 'Red' AND weight = 19;
```

Original query plan:

Outer Tbl	Ind_Used-O	Inner Table	Ind_used-I	Result Tbl	Join Method
		P	PXWEIGHT	RESULT	

After forcing statistics:

```
UPDATE STATISTICS ON TABLE p SET rowcount=1000,  
pagecount=100,rowpagecount=100;
```

```
UPDATE STATISTICS ON INDEX pxcolor SET distinctcount(color)=900,  
leafcount=5,height=2,clustercount=100;
```

Outer Tbl	Ind_Used-O	Inner Table	Ind_used-I	Result Tbl	Join Method
		P	PXCOLOR	RESULT	

There is a host of other useful information that can be found in the Systems Catalogs

For example, in systables you can find the page-count (number of data pages in a table) and extent page counts. In sysindexes the number of overflow pages allocated for an underlying table using a hashed index is shown.

### **REORGANIZE procedure**

Over time a database can get fragmented if it does not have contiguous disk space and tables can get fragmented due to modifications to the data. We need to defragment!

It is advisable not to use the actual REORGANIZE command which automates the 'unload', initialize database and 'reload' process! It is better to run your own script that mimics the 'reorganize' and adds performs other useful functions. It also runs 2-5 times faster and is much safer in the event of hardware failure.

Using the process SQLBase will collate all row pages with their extent pages leading to lesser fragmentation and also identifies any corruption problems.

**Run timely  
procedures to  
defragment the  
database**

Below is a sample script in the order you should perform it:

```
CONNECT <databasename>
UNLOAD ... ON SERVER
DISCONNECT ALL
SET SERVER servename/password
DROP DATABASE
CREATE DATABASE
CONNECT <databasename>
SET RECOVERY OFF
LOCK DATABASE
LOAD ... ON SERVER
UPDATE STATISTICS
UNLOCK DATABASE
SET RECOVERY ON
DISCONNECT ALL
```

Remember to specify the 'on server' so the database unload is placed at the server destination. An extra step could be to Defragment your disks before create and load.

### ***What about Bulk Operations?***

Better to turn off Referential Integrity. and drop indexes and recreate them after the bulk operation.

After bulk deletes, be sure to run your REORGANIZE scripts (as above), as the pages are not returned to the free page pool leading to incorrect statistics.

### ***Locking the Database***

For DBA related activities e.g. 'load', 'check database' etc. a single user can have an exclusive lock on the entire database. No new connections would be allowed.

A database lock is acquired by executing the LOCK DATABASE statement in SQLTalk. It puts the database in single user mode. As the single user has exclusive access to the database the lock manager is disabled and performance can improve significantly.

A database lock is released by executing the UNLOCK DATABASE statement.

### ***Timeout mechanism***

SQLBase's locking strategy uses a timeout mechanism for a transaction waiting for a lock resource that is already held.

The default timeout period is 300 seconds (5 minutes) and is configurable at the transaction level.

The permissible timeout values are:

1 – 1800: specifies timeout period in seconds (1 second to 30 minutes).

0: specifies no waiting. If the lock is not immediately available, a timeout occurs.

-1: specifies an indefinite timeout period, the transaction will wait forever (theoretically) to acquire the lock.

Strategies to avoid timeouts due to lock contentions are:

- Spread out the data.
- Clustered Hashed Indexes.
- Larger PCTFREE value on table creation (see above on this affecting retrieval performance).
- Applications access tables in the same order.
- Do not allow user interaction during a transaction
- Use Release Locks isolation level.
- If contention is due to statement execution time on DML, consider removing indexes (that are being updated) that have little value or are infrequently used. Conversely, find and add an index that improves query performance.

For a detailed look at the timeout and locking activity you can use the 'START AUDIT', category 8 feature' which provides an audit file of lock manager, deadlock and timeout activity. Note, other 'Audit' categories are available for tracing including monitoring who logs on to a database, what tables they access, or record command execution time.

The auditing will have its own effect on performance so use judiciously.

You can also use the SQLBase server process screen to check server status, process and system activity for up to 4 levels of tracing.

### ***Precompiled queries in the database***

Performance can be gained by utilizing pre-compiled commands that are frequently used by your applications.

### **Stored and Chained Commands**

A query or data manipulation (DML) command or commands (chained), is pre-compiled and stored in the database for later execution.

### **Stored Procedures**

A sequence of GUPTA Team Developer SAL statements that can be assigned a name, compiled, and used immediately or (optionally) stored in SQLBase.

**Server side  
compiled  
procedures can  
dramatically  
improve  
performance and  
reduce  
maintenance  
effort**

The logic uses SAL procedural statements for flow control and a subset of SAL Sql\* functions with the ability to accept input and output parameters and call external functions. Note, an instance of a procedure is associated with a single application cursor and is executed, fetched and closed using the same cursor.

### **Performance advantages of Procedures:**

- Improved runtime performance because the procedural logic and (optionally) the SQL statements are pre-compiled.
- Reduced client/server network traffic improves performance.
- By preparing and executing the SQL statements on the server, the client application need only call the procedure and wait for results.

Stored procedures can be particularly effective because all complexity and specifics of a transaction can be hidden from an application developer. Maintenance effort is also reduced.

### **Types of Procedures**

Stored versus Non-stored versus Inline

Non-stored procedures - compiled and executed directly without storing in database

In-Line procedures - used by triggers when using the `INLINE` clause

Static versus Dynamic Procedures

You need to specify whether a procedure is Static or Dynamic at the time of creation. "Static" is faster as the query is precompiled and 'stored'; you are also able to use Triggers. 'Dynamic' should be used if you actually need to perform Dynamic SQL statements within the procedure and when you need to perform DDL e.g. 'Create table @tablename'.

### **External Functions**

User-defined functions, which reside in an external DLL, can include SAL functions (such as string manipulation functions). These are invoked from with a SQLBase procedure and can be written in C, C++, etc.

For optimal performance, frequently used DLLs should be pre-loaded.

### **Triggers**

Triggers can improve performance because the processing is done by stored or inline procedures and all of the processing invoked by the trigger is executed on the server without any client/server traffic.

When using triggers, use the 'ALTER TRIGGER' command to enable/disable individual triggers. This does not change or drop the definition of a trigger and is useful when you need to bypass errors

when a trigger references an object that is unavailable or to load and reload databases quickly.

### ***Trace statement***

The Trace statement prints the value of one or more *variables* to the Process Activity screen on the server. Embed Trace statements in the Actions section of a procedure.

Using Trace statements could aid to diagnosing a performance problem. The trace can indicate the SQL statement(s) or external calls causing the problem, especially if time stamping is turned on, and the log option used to write the process activity output to a log file. Remember to turn off both time stamping and trace logging after use.

In SQLTalk you can "SET TRACE ON | OFF" which will display ALL statements in the procedure's Actions section on the process activity screen before execution.

## **Tools and Tuneable settings**

Tuning SQLBase may involve optimizing memory allocation, minimizing I/O requests and reducing contention for SQLBase resources. The information needed to make the decisions to change tuneable elements comes from assessing overall performance. Tools such as SQLConsole help greatly.

### ***Monitoring Tools***

#### **SQLConsole**

Gives a picture of the whole server. Can be used to monitor cursor and locking activity, virtual: Physical I/O Ratios, Cache Hit Ratio and many other useful figures.

Some examples to aid performance assessment are:

- How many processes are running concurrently? If >100 the server is probably overloaded.
- Are process switches > 100? This means that the server is CPU-bound.
- What isolation levels are being used? Most often RL.
- How many cursors are attached? If >400 then server probably overloaded.
- Are there any 'old' cursors not marked as 'inactive'? Old cursors that may be in 'fetched' or other operation state that have been 'hanging around' mean the command has not been committed and could be holding locks.

**SQLConsole is invaluable in analyzing the actual behavior of transactions and other activity on the server**

## Read-only mode is a huge performance 'hog'

- Are there any large SQL Costs shown? If so, the query is poorly designed so check query plans. SQLPerformance tool can aid to check frequency of query.
- What isolation level is being used? This should normally be in Release Locks (RL) mode. Check for other isolation levels – for example, Read Repeatability (RR) may show for a SQLTalk user who has not set the default isolation level for a session.

### SQL.INI settings

Check if readonly=1 set? If so, do you really need any database on the server to operate in read-only mode? If not, remove the line. It can hugely affect performance due to needing to maintain 'history' files for transactions. If you need to have one database operate in read-only mode, you can set this for that database only using the SQLTalk 'set readonly on' command.

At connection time SQLBase will search each comdll= statement until it finds the database or until it runs out of protocols. Are there out of order or extraneous protocols? Put the most-used protocol as first in the list and get rid of any protocols not used. For example, if you normally access the databases using TCP/IP, make sure that any comdll statements for Anonymous Pipes (apipe) appear after the comdll for TCP/IP (sqlws32).

You also need to check that all client DLL versions match the server version, e.g. check versions of SQLWNTM.DLL and SQLBAPIW.DLL.

Remember to set cache and sortcache settings as described earlier in the paper.

### groupcommit

Specifies the maximum number of *COMMIT*'s that SQLBase groups together before physically writing them to disk. SQLBase tries to economize resources and increase transaction rates by grouping *COMMIT*'s from several transactions into one transaction log entry then performing them all in one physical write before returning control to the user. It does this, by pausing for a fraction of a moment, in anticipation of other incoming *COMMIT* request. Commits are performed when:

The number of grouped commits exceeds the value of *groupcommit* (*default=2*)

or

The number of ticks that have elapsed since the last commit is greater than the value of *groupcommitdelay* (*see below*).

### groupcommitdelay

Specifies the maximum number of system ticks that SQLBase waits before performing commits. The duration of a system tick is platform-dependent, but is approximately 1/20 of a second.

## Sortcache

Specifies the number of 1K (1024-byte) pages of memory to use for sorting. See earlier section on setting this value and when it comes into effect.

## optimizefirstfetch

Used to set the optimization method for the first fetch. When set, the keyword instructs the optimizer to pick a query execution plan that takes the least amount of time to fetch the first row of the result set.

The valid values for this keyword are 0 and 1. When *optimizefirstfetch* is set to 0, SQLBase optimizes the time it takes to return the entire result set. When *optimizefirstfetch* is set to 1, SQLBase optimizes the time it takes to return the first row.

## logfileprealloc

Enables and disables transaction log file preallocation. See earlier section explaining how setting this value can help increase performance. The default setting is (0). Setting *logfileprealloc* on (1) means that SQLBase creates log files at full size (preallocated) and they do not grow.

## inmessage

The input message buffer holds the data coming back from the server. There is one input message buffer per client connection handle. The default size is 2000 bytes. Despite the *inmessage* value, SQLBase dynamically allocates more space when necessary as it will automatically size to hold at least one row.

When fetching data, as many rows as possible are stored into one input message buffer. Each fetch command reads the next row from the input message buffer until the end of the message buffer is reached. The SQL/API transparently fetches the next input buffer of rows (the actual behavior depends on the current isolation level)

A large input message buffer can improve performance because it reduces the number of messages between the client and server. However, it can consume lots of memory. In some cases, it may be wise to size the message buffer and network data packet size to match.

It is recommended that *inmessage* be set to 32767 to minimize network traffic.

## outmessage

The output message buffer holds the output from the application such as the SQL statement to compile or rows of data to insert. Again, it is dynamically allocated if needed. The setting does not affect

performance because the buffer only needs to be large enough to hold the largest SQL command to compile or the largest row of data to insert

## The behavior of and use of input and output message buffers is affected by the isolation level set

A large output message buffer can allocate space unnecessarily and will not reduce network traffic unless bulk execute is on where more than one statement is sent to the server in one message.

The default size is 1000 bytes and output message buffer is automatically resized to process rows greater than its current size.

It is recommended that outmessage be set to 32767 to minimize network traffic.

### **Fetchthrough**

When fetchthrough is enabled, no data buffering occurs as the input message buffer is bypassed and the most current information is retrieved directly from the database. Regardless of input message buffer size, a row is fetched one at a time from server.

Ensures most up-to-date information from the database but increases response times due to no buffering.

Fetchthrough provides functionality similar to the Cursor Stability isolation level but without maintaining locks on the current page.

The feature is only applicable to queries against base tables and will not work if the query involves ORDER BY, GROUP BY, JOINS, UNIONS, etc.

### **Application specific considerations**

The major improvements in performance will come from understanding how your applications need to access and manipulate data and through sound database design.

Other areas we can look into are:

- Improve performance of specific SQL statements.
- Improve performance of a specific application by assessing locking strategies and transaction semantics.
- Using SQLConsole and studying the system tables to identify queries not using indexes and creating appropriate indexes to support the query.
- Optimizing network traffic between server and client and configuring the client to get maximum performance.

Check if the application is being run remotely? If so, move the application .exe file and all client SQLBase DLLs to the client PC. Don't run this from a network file server.

**Optimize SQL access based on how the optimizer will treat different constructs**

## **Query performance**

The SQLBase optimizer will restate the query, to reduce processing time, following the rules of query transformation.

Convert or flatten sub-queries into joins to improve performance. Use techniques with temporary tables and/or indexes to flatten the sub-query.

Study entries in the PLAN TABLE to check if temporary tables are being created during 'conversions'. This indicates that the optimizer is sorting the intermediate results so try adding an index to avoid the sort process.

Other tips:

- Avoid use of 'UPDATE ... WHERE CURRENT OF'. It is better to use Release Lock isolation level checking for ROWID.
- Take care with queries with LIKE and bind variables:

**For example, Fieldname like '%def' will not use an index.**

Salary > :Bindvariable probably won't use an index

Salary > 1000 will have accurate selectivity factor

Use constants or add another clause with equality predicate.

- Avoid using more than four union sub-queries. Consider transforming these into a stored procedure.
- Modifying predicates forces the optimizer not to use indexes:

For example,

Change 'where sysdate > tbl\_date\_column + 4 and ...'

to 'where sysdate - 4 > tbl\_date\_column and ...'

- Do not use ORDER BY clause without having appropriate indexes otherwise the clause will result in expensive sorts.
- Do not use DISTINCT and ORDER BY clauses in same sub-query. DISTINCT does a sort, so order the columns in the select list to match the order required.... Duplicated effort.
- There are problems forcing non-duplicate result sets (DISTINCT) with Union's. It is better to make the sub-queries unique and then use UNION ALL.
- Use sub-queries in UPDATE ... SET clause. This results in less network traffic and thus is more efficient.

- Check that SQLPrepare statements are done before a loop is set up to do the SQLExecute i.e. only do the 'Prepare' once instead of SQLPrepareAnd Execute inside the loop. Do not use SqlImmediate in GUPTA Team Developer SAL language as it forces a database connect, compile and disconnect each time the statement is executed.

## **Indexes**

Maintaining correct and efficient indexes is the key to optimal performance.

### **Why use indexes?**

- Force uniqueness of primary keys.
- To enforce constraints such as uniqueness and foreign keys.
- To optimize queries by providing a means of locating rows faster than a table scan.

**Use of indexes can have a big impact but can have a negative effect on performance if used incorrectly**

After initial creation, index efficiency and usefulness could decrease due to a number of reasons:

- The index could become unbalanced.
- For heavily modified tables fragmentation could lead to many extent pages overriding use of indexes.
- The index could suddenly stop being used due to application and data changes.
- Also, either force index usage or drop index if not appearing in execution plans.
- If UPDATE STATISTICS has not been run after index creation the index will not be considered when formulating the query.
- Deleting data can cause balancing problems in the tree structure and fragmentation of the index blocks. If you suspect this is the case you should drop and recreate the index.

Do not load data in sorted order by the designated index. Data values should be randomly distributed.

You may notice that performance is really slow when deleting from a table that has foreign keys. This is due to the fact that foreign keys need to be established **after** an index exists on the joining column in each child table. For each table with the foreign key:

- 1) Drop the foreign key.
- 2) Create an index on the column used for the key.
- 3) Recreate the foreign key.

## ***Clustered Hashed Indexes***

Clustered Hashed Indexes (CHI) can also be used to specify how row data should be positioned for efficient access. Table rows are stored in locations based on their key value (clustering). CHI's must be created on a table before any data is inserted. They perform best for random row access on static tables. When the index is 'created', space is pre-allocated in the table for the specified number of rows.

You cannot update the columns (of a table) that make up a clustered hashed index.

Any rows that don't fit in the pre-allocated pages are stored in overflow pages. This will degrade performance so care must be taken to choose the size judiciously.

## ***Composite indexes***

Based upon more than one column. They can be defined in conjunction with any of the other index types (for example, a clustered hashed index based upon one function and one column).

It may be worthwhile to create a composite index combining several columns. SQLBase chooses composite indexes only for queries that make use of at least the first column of the index. The best performance is achieved by placing the column that usually has the most restrictive selection criteria first. For example, if col2 is the most restrictive:

```
CREATE UNIQUE INDEX ix5 ON tb1(col2, col1);
```

## ***Choosing an Index***

### Good Candidates

- Primary key columns: primary keys must have unique indexes for referential integrity.
- Foreign key columns: can be used to perform joins by the server, additionally can be used for referential integrity enforcement.
- Columns referenced in WHERE predicates or has GROUP BY or ORDER BY clauses. Prevents the creation of intermediate result tables.
- Columns that are subject to aggregate functions (involving SUM, AVG, MIN, MAX). For MAX() based queries and "latest" value queries make sure that your index has DESCENDING keyword on the column. For MIN() based queries and for "earliest" queries, ensure that index has ASCENDING on the column.

## Poor Candidates

- Small tables: If the table has less than five pages, overheads are more than benefits
- Large batch updates and inserts: The indexes need to constantly update thus degrading performance. Alternative is to drop the indexes before the batch operations and recreate them.
- Non-uniqueness of keys: In cases where the cardinality is high, but there are a large number of rows with the same value, there may be a performance penalty.
- Low cardinality: The optimizer would avoid these indexes because of low selectivity.
- Many unknown values: Null values skew the distribution hence could lead to performance penalties.
- Frequently changing values: There is index maintenance overheads, as well as locking contention on the index structure

## ***Transaction semantics***

Transactions are committed:

- When the Isolation Level is changed
- When recovery mode is changes
- When a transaction is terminated (either explicitly or implicitly)
- When Committed

Avoid long running transactions:

- Many locks acquired and held.
- Log files remain pinned down.
- Need to increase COMMIT frequency.

## ***AUTOCOMMIT***

When autocommit is enabled, each statement is committed immediately so there is no concept of multi-statement transactions and the scope of a rollback is only one statement.

- Autocommit must be explicitly enabled (at the cursor level).
- When autocommit is enabled for a cursor, it will affect all other cursors that are part of the same transaction.
- Autocommit can improve performance, especially in network environments as it reduces the communication between the client and the server.
- Autocommit can improve availability and reduce contention because locks are quickly released (see Locks, next).
- Enabling and disabling autocommit as required is a possibility, but it can lead to coding bugs that are difficult to track down.
- Generally, autocommit is worthy of consideration for interactive sessions with ad-hoc query and reporting tools.
- Generally not good for large number of DML statements which could be put in one transaction e.g. 'load': A transaction can be terminated implicitly in certain situations.

**Need to be careful with transactions initiated via ad-hoc tools such as SQLTalk**

## ***Cursor Context Preservation (CCP)***

By default, a COMMIT statement destroys all active result sets in the committed transaction and all locks acquired by the transaction are released. Result sets are destroyed.

Enabling CCP maintains locks held and results sets even after a COMMIT. The application can continue to process a result set on one

cursor and execute DML statements on another cursor in the same transaction. Typically, the DML statements manipulate the same table that was queried to produce the result set.

- CCP also supports maintaining results set on a ROLLBACK, if the RL isolation level is used and no DDL operation were performed.
- CCP is configurable at the cursor level.
- By default it is not enabled.

## ***Locking***

SQLBase uses Shared locks (S), Update locks (U) and Exclusive locks (X). S and X locks behave as 'read locks' and 'exclusive write locks'. A U lock is treated as an 'intended update' or 'intended exclusive' lock.

A U-lock is compatible with S-locks but not other U-locks or X-locks. This means that a U-lock can be placed on a page that already has S-locks on it, but not on a page that has a U-lock or an X-lock on it.

U-locks help reduce the possibility of a deadlock situation, especially with positioned UPDATES and DELETES.

Although S-locks and U-locks can co-exist on the same page, all S-locks from other transactions must be released before a U-lock can be upgraded to an X-lock.

When any kind of lock is placed on a row page, the same type of lock is applied to any extent pages and LONG VARCHAR pages associated with the row.

When an index is used, either because of Declarative Referential Integrity (DRI) or to select rows for processing, the index pages containing the associated index nodes are locked with the same type of lock applied to the row page.

For indexes, the entire index page is locked, even if only one index node is the subject of the lock.

If a transaction has the sole S-lock on a page, it can upgrade it to an X-lock.

## ***SQLBase Isolation Levels***

Isolation levels are used to determine the duration of S-locks in a transaction. The selected isolation level can also have side effects, such as affecting how the input message buffer is used.

Isolation levels have no effect on the duration of X-locks or U-locks.

A transaction has ONE isolation level. Isolation levels apply to the whole transaction and a change in isolation level will perform a COMMIT and change all new cursors to the new level.

### ***Read Repeatability (RR)***

RR is the default isolation level (better safe than sorry)

The RR isolation level makes effective use of the input message buffer, filling it with row data at the server, then sending the buffer to the client. This is suitable for transactions that require a high degree of consistency throughout the transaction.

In a multi-user environment, use of RR can lead to timeouts and deadlocks, so applications may need elaborate COMMIT and/or restart logic.

### ***Cursor Stability (CS)***

With Cursor Stability a S-lock is maintained on the **current** page you are processing.

As you fetch rows after executing a query, the page of the current row is held with an S-lock. You are guaranteed stability at the cursor (cursor in this context is the current row of the result set). When the page changes as rows are fetched, the S-lock on the current page is released and an S-lock for the next page is acquired.

CS makes sub-optimal use of the input message buffer, sending only one row at a time from the server to the client.

CS is suitable for applications that do row at time processing with positioned updates.

CS is also worthy of consideration if DB2 compatibility is desired.

### ***Read-Only (RO)***

To accommodate RO, SQLBase creates history files of the data as it is modified by other transactions. This uses considerable system resources (disk space for history files, additional overhead of writing and managing history files, etc.).

Note that due to the overhead, SQLBase must be explicitly instructed to allow cursors to use RO. This is accomplished with the SQLTalk 'set readonly on' statement or on rare occasions, the **readonly** keyword in the server's SQL.INI definitions when 'all' databases on a server are allowed to be connected to in a readonly mode.

This isolation level uses no locks and is not blocked by locks. If an X-lock is on the page that contains rows applicable to a query, the history file provides the rows instead.

RO guarantees data consistency based on the time the transaction started but does not guarantee the currency of the data. It is like working with the database from a snapshot in time.

When RO is used, the input message buffer is used optimally. The buffer is filled at the server first, and then sent to the client.

RO is suitable for complex reports that require point-in-time consistency of the database.

Performance impact of this isolation level is huge. You also must stop and start processes periodically to clean up history files!

### ***Release Locks (RL)***

RL provides a low degree of data consistency and a high degree of user concurrency. RL places S-locks on data as it builds a result set.

Once the result set is complete and the server is ready to return control to the client, all S-locks are released. As rows are fetched from the result set, an S-lock is briefly placed on the page of the row to place it in the input message buffer.

Unlike with CS, this S-lock is immediately released. Before control returns to the client, all S-locks are guaranteed to have been released.

The RL isolation level makes optimal use of the input message buffer, filling it with row data before returning it to the client.

RL is the recommended isolation level for client/server applications, especially those that provide data for browsing and updating. This provides the best performance.

Isolation Level	Consistency	Contention	Concurrency
RR	High	High	Low
CS	Medium	Medium	Low
RL	Low	Low	Low
RL + rowid	Medium	Low	Medium
RO	High	Low	Low

### ***Locking strategies***

#### Pessimistic Locking

**Most multi-user systems will use optimistic locking with "Release Locks" isolation level**

The RR isolation level is suitable for use with pessimistic locking.

When a pessimistic locking strategy is used, applications should be liberal with the timeout period, allowing ample time for other transactions to complete.

Pessimistic locking is not well suited to highly interactive, multi-user applications, typical of client/server applications.

Pessimistic locking can be suitable for 'batch-mode' applications, applications that run in periods of little or no concurrent activity e.g. often automated, scheduled jobs that run after hours.

### Optimistic Locking

RL is the ideal isolation level to use with an optimistic locking strategy, though CS will also work.

Optimistic locking is well suited to highly interactive, multi-user applications, typical of client/server applications as the fetched rows from a result set are maintained in local storage on the client.

For updates and deletes – we need to use the ROWID as a Time Stamp to operate in an optimistic locking scenario.

```
UPDATE T SET C1 = :1 WHERE ROWID = :2
```

If no other transaction has affected the specified row, the UPDATE or DELETE attempts to acquire an X-lock and make the respective change. Otherwise an 806 error is returned (Invalid ROWID).

If the client application is using result set mode, it can simply attempt to re-fetch the row. (Important: Enable Fetchthrough: see the topic earlier in this chapter) The fetch indicator value will either be 2 (row was updated) or 3 (row was deleted).

If the client is not using result set mode, the client can execute another SELECT based on the primary key of the row to retrieve new ROWID, if it still exists.

## **Result Set Mode**

SQLBase's result set mode provides a backend, scrollable, bi-directional result set that is persistent for the life of the transaction (and sometimes longer). By default, result set mode (also called SCROLL mode) is enabled so rows can be fetched in forward or reverse order, or in a random access order.

The advantage of using SQLBase's result set mode over maintaining a front end result set, is that SQLBase can alert you when you retrieve rows that have been updated or deleted since the query was executed.

By maintaining back-end result sets, SQLBase can dynamically update result sets when the ADJUSTING clause is used with the INSERT statement. Here, the row is added to the end of the results without invalidating the current result set.

Use of result set mode is required for positioned UPDATEs and positioned DELETEs.

Copyright © 2004 Gupta Technologies LLC. GUPTA, the GUPTA logo, and all GUPTA products are licensed or registered trademarks of Gupta Technologies, LLC. All other products are trademarks or registered trademarks of their respective owners. All rights reserved.