

Tuning MySQL SQL Code

by Guy Harrison

MySQL has a well deserved reputation as a light-weight and efficient database server capable of meeting the requirements of even the most demanding high volume web applications. Nevertheless, as the part of your application that does virtually all the disk IO, it's going to be the source of a significant proportion of your application response time. Therefore, if you want your application to meet its full performance potential, you should not shirk on tuning the MySQL SQL code.

Poorly tuned SQL not only takes more time than it ought, it also tends to scale badly as data volumes increase. Consider the response time curve shown in Illustration 1, which shows response time for a two table join where the tables are not correctly indexed. As the sizes of the tables increase, so does the response time, but at an ever-increasing rate. If the tables grew to millions of rows, the projected response time would be measured in days!

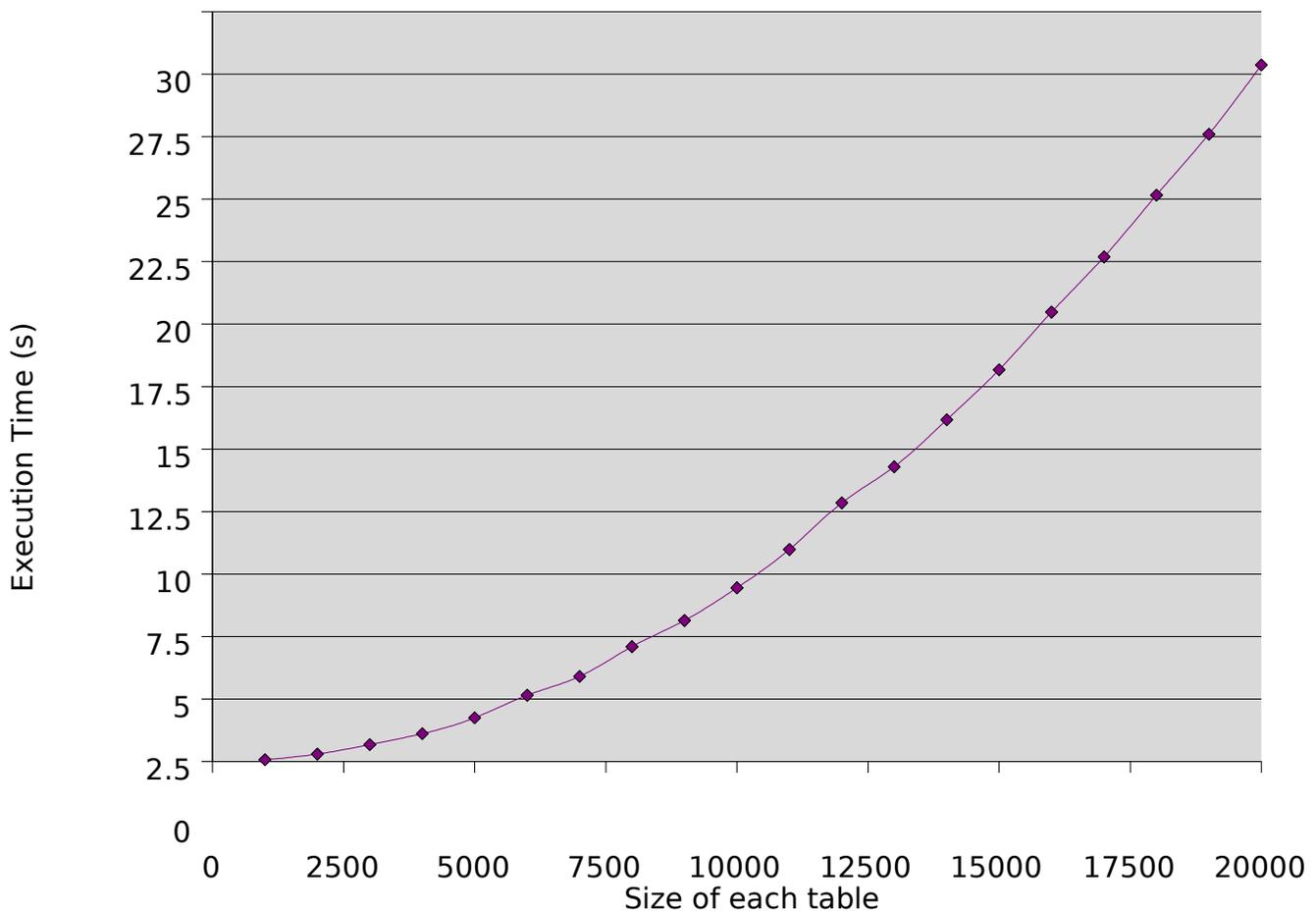


Illustration 1: Performance of unindexed join

So lets take a look at the basic tools, techniques and rules of thumb for ensuring your SQL code doesn't let you down.

Tools of the trade

There are a couple of tools that you should get familiar with if you're going to be able to systematically

tune your SQL:

The EXPLAIN command generates an explanation of how the optimizer is going to execute your SQL statement. This includes choices of indexes, expected row counts, join order and other useful information. Example 1 shows some EXPLAIN command output. We don't have time to review all the features here, but note that this output indicates the order in which the two tables are joined and the use of indexes (The `i_employees_name` index is first used to select employees and then the `i_customers_sales_rep` index is then used to join matching employees to customers).

```
mysql> explain select customer_name
->   from employees join customers
->     ON(customers.sales_rep_id=employees.employee_id)
->  where employees.surname='GRIGSBY'
->     AND employees.firstname='RAY' \G

***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: employees1
         type: ref
possible_keys: PRIMARY,i_employees_name3
          key: i_employees_name4
       key_len: 80
         ref: const,const
          rows: 15
      Extra: Using where; Using index6
***** 2. row *****
      id: 1
  select_type: SIMPLE
        table: customers2
         type: ref
possible_keys: i_customers_sales_rep
          key: i_customers_sales_rep8
       key_len: 9
         ref: sqltune.employees.EMPLOYEE_ID7
          rows: 55589
      Extra: Using where
2 rows in set (0.04 sec)
```

Example 1: EXPLAIN command output

Optimizer Hints are instructions you can embed in your SQL that are instructions that you can embed in your SQL that do not change the meaning of the SQL, but rather instruct or suggest to the optimizer how you would like the SQL to be executed.

Most of the time you will not need to add hints. In fact, hints can be dangerous because they limit the choices the optimizer has available, and if data in the tables change or if new indexes are added to the table, MySQL may be unable to adapt because of your hints. However, there are definitely situations where you will discover that the optimizer has made a less than perfect decision and you want to give the optimizer specific instructions.

Table 1 lists the commonly used optimizer hints.

Hint	Where it appears	What it does
STRAIGHT_JOIN	After the SELECT clause	Forces the optimizer to join the

		tables in the order in which they appear in the FROM clause. Use this if you want to force tables to be joined in a particular order
USE INDEX(index [,index...])	After a table name in the FROM clause	Instructs MySQL to only consider using the indexes listed. MySQL may choose to use none of the indexes if it calculates that they would not be faster than scanning the entire table.
FORCE INDEX(index [,index...])	After a table name in the FROM clause	Instructs MySQL to use one of the indexes listed. This differs from USE INDEX in that MySQL is instructed not to perform a table scan of the data unless it is impossible to use any of the indexes listed.
IGNORE INDEX(index [,index...])	After a table name in the FROM clause	Instructs MySQL not to consider any of the listed indexes when working out the execution plan.

Measuring SQL execution. When we execute an SQL statement from the MySQL command line, MySQL is kind enough to report on the elapsed time taken to execute the statement:

```
mysql> call TestProc1() $
Query OK, 0 rows affected (9.35 sec)
```

Elapsed time is a good first measurement of SQL or stored program performance, but there are lots of reasons why elapsed time might vary between runs that may have absolutely nothing to do with how well the SQL statement is optimized:

- Other users may be running jobs on the host while you execute your SQL statements; you will be contending with them for CPU, disk I/O and locks.
- The number of physical IOs necessary to execute your statement will vary depending on the amount of data cached in the operating system and within the various MySQL caches.

For these reasons, it is sometimes better to obtain additional metrics to work out if your tuning efforts are successful. Useful execution statistics can be obtained from the SHOW STATUS statement, although the level of detail will vary depending on your storage engine, with InnoDB currently offering the most comprehensive selection of statistics.

Generally, you will want to compare before and after variables for each statistic and – because the statistics are sometimes computed across all sessions using the MySQL server – ensure that your session has exclusive use of the server while your statement runs.

The **Slow Query Log** allows us to identify SQL statements that might be eligible for tuning. You can do this by adding the following lines to your MySQL initialization files:

```
| log_slow_queries  
| long_query_time=seconds
```

This will cause MySQL to write any queries that exceed an elapsed time exceeding *seconds* execution time to a log file. The logfile can be found in the MySQL data directory and is named *hostname-slow.log*. In MySQL 5.1, the log can also be accessed as a table within the *mysql* schema. For each SQL statement identified, MySQL will print the SQL statement along with a few execution statistics.

Using indexes to retrieve data

A common mistake made by those new to SQL tuning is to assume that it is always better to use an index to retrieve data. Typically, an index lookup requires three or four logical reads for each row returned. If we only have to traverse the index tree a few times, then that will be quicker than reading every row in that table. However, traversing the index tree for a large number of rows in the table could easily turn out to be more expensive than simply reading every row directly from the table.

For this reason, we generally want to use an index only when retrieving a small proportion of the rows in the table. The exact break even point will depend on your data, your indexes, and maybe even your server configuration, but I have found that 5-10% is a reasonable rule of thumb.

The MySQL optimizer predicts when to use an index based on the percentage of data from the table it expects to retrieve given a specific *WHERE* clause. The optimizer chooses to use the index for small intervals, while relying on a full table scan for large intervals. This basic algorithm works well when the volume of data is evenly distributed for the different indexed values. However, if the data is not evenly distributed, or if the statistics on table sizing are inaccurate, then the MySQL optimizer may make a less than perfect decision.

Illustration 2 shows the elapsed time for retrieving various proportions of rows when forcing an index scan or a full table scan, or when allowing MySQL optimizer to make that decision. In this example, MySQL switched from index to full table scan when the rows returned represented approximately 7% of the total. However, in this case, the index outperformed the table scan until about 17% of the rows were retrieved. So although MySQL made the correct decision in most cases, there were a few cases where forcing an index lookup would have improved performance. If we wanted to override the MySQL optimizer, we could force an index lookup using the *FORCE_INDEX* hint mentioned earlier.

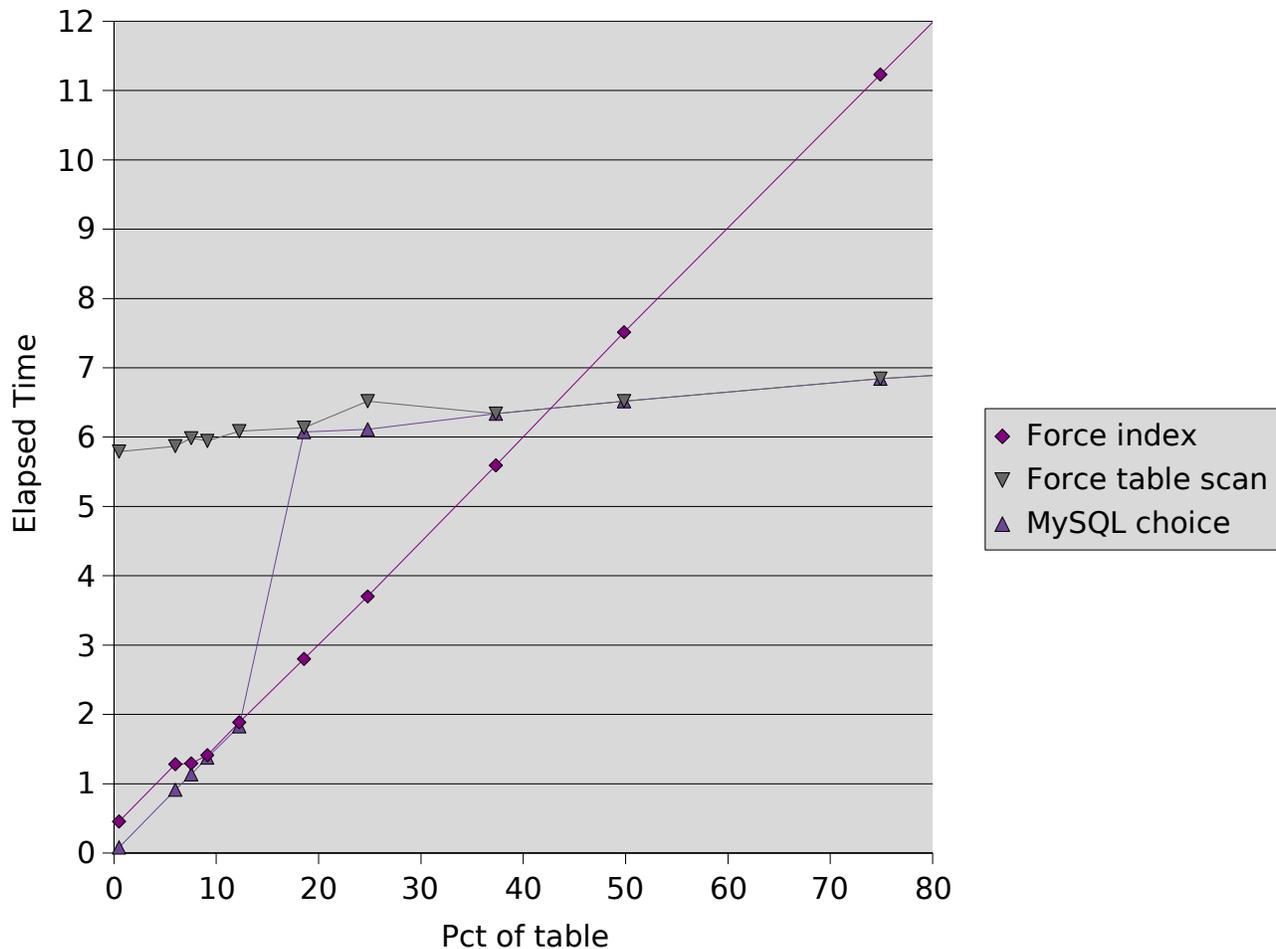


Illustration 2: Full table scan versus index lookup

Concatenated Indexes

A concatenated index – often called a “composite” index - is an index that is created on multiple columns. For instance, if we frequently retrieve customers by name and date of birth, we might create an index as follows:

```
create index i_customers_first_surname_dob on
customers(contact_surname, contact_firstname, date_of_birth);
```

There is very little chance that two customers would have the same first name, surname, and date of birth, so use of this index would almost always take us to a single, correct customer. If you find that you frequently need to query against the same set of multiple columns’ values on a table, then a concatenated index based on those columns should help you optimize your queries.

We can use a concatenated index to resolve queries where only some of the columns in the index are specified, provided that at least one of the “leading” columns in the index are included.

For instance, if you create an index on (surname, firstname, date_of_birth) you can use that index to search on surname or on surname and firstname, but you cannot use it to search on date_of_birth. Given this flexibility, organize the columns in the index in an order that will

support the widest range of queries. Remember that you can rarely afford to support all possible indexes because of the overhead indexes add to DML operations – so make sure you pick the most effective set of indexes.

Covering indexes

Creating a covering index is a very powerful technique for squeezing the last drop of performance from your indexes. If there are only a few columns in the SELECT clause that are not also in the WHERE clause, you can consider adding these columns to the index. MySQL will then be able to resolve the query using the index alone, avoiding the I/Os involved in retrieving the rows from the table. Such an index is sometimes called a “covering” index.

For instance, consider the following query:

```
select count(*), sum(quantity)
  from sales
 where customer_id=77
    and product_id=90
    and sales_rep_id=61
```

To satisfy the WHERE clause, we would create an index on `customer_id`, `product_id` and `sales_rep_id`. However, if we add the `quantity` column to the index our query can be resolved from the index alone – with a significant IO savings.

Avoiding accidental Table scans

There are a few circumstances in which MySQL might perform a table scan even if a suitable index exists and perhaps even after you instruct MySQL to use an index with the FORCE INDEX directive. The three main reasons for such “accidental” table scans are:

- You modify an indexed column in the WHERE clause with a function or an operator. You can usually avoid this by rewriting the SQL to manipulate the search value, rather than the column value.
- You are searching for a substring within an indexed column. Indexes can be used to search for wild cards, but only if the leading portion of the string is specified.
- You are using only some of the columns within a concatenated index, and the order of columns in the index does not support searching on the columns you have specified. To use a concatenated index, you must include at least the first column in the index in your WHERE clause.

In each of these cases, the EXPLAIN command will reveal that an index was not used and you can either rewrite the query or create more appropriate indexes.

Joining tables

To get predictable and acceptable performance when joining tables, you need to create indexes to support the join. Generally, you will want to create concatenated indexes based on any columns in a table that might be used to join that table to another table. However, you don't need an index on the

first (or “driving”) table’s columns; that is, if I am joining CUSTOMERS to SALES in that order, then my index needs to be on SALES – I don’t need an index on both tables.

Creating an index on the join column not only reduces execution time, but also prevents an exponential increase in response time as the tables grow in size. Our very first example – see Illustration 1 - shows how performance degrades as tables increase in size when an index is not present.

By far, the most important factor in the optimization of MySQL joins is to ensure that each successive join is supported by an index. Beyond that, we should:

- Ensure that any rows to be eliminated by WHERE clause conditions are done so as early as possible in the join.
- Pick an optimal join order. A good rule of thumb is to join tables from smallest to largest.

Generally, the MySQL optimizer can be relied upon to pick a good join order. However, if you need to change the join order, you can use the STRAIGHT_JOIN hint to ensure that the tables are joined in the order in which they appear in the FROM clause.

Tuning ORDER and GROUP BY

GROUP BY, ORDER BY and certain group functions (MAX, MIN, etc.) may require that data be sorted before being returned to the user. If there is sufficient memory, the sort can be performed without having to write intermediate results to disk. However, without sufficient memory, the overhead of the disk based sorting will often dominate the overall performance of the query.

There are two ways to avoid a disk-based sort:

- Allocate more memory to the sort. When MySQL performs a sort, it first sorts rows within an area of memory defined by the parameter SORT_BUFFER_SIZE. If the memory is exhausted, it writes the contents of the buffer to disk and reads more data into the buffer. This process is continued until all the rows are processed; then, the contents of the disk files are merged and the sorted results are returned to the query. The larger the size of the sort buffer, the fewer the disk files that need to be created and then merged. If the sort buffer is large enough, then the sort can complete entirely in memory. You can allocate more memory to the sort by issuing a SET SORT_BUFFER_SIZE statement
- Create an index on the columns to be sorted. MySQL can then use the index to retrieve the rows in sorted order.

Illustration 3 shows how increasing the amount of sort memory, or creating an index, reduces the elapsed time for sorting.

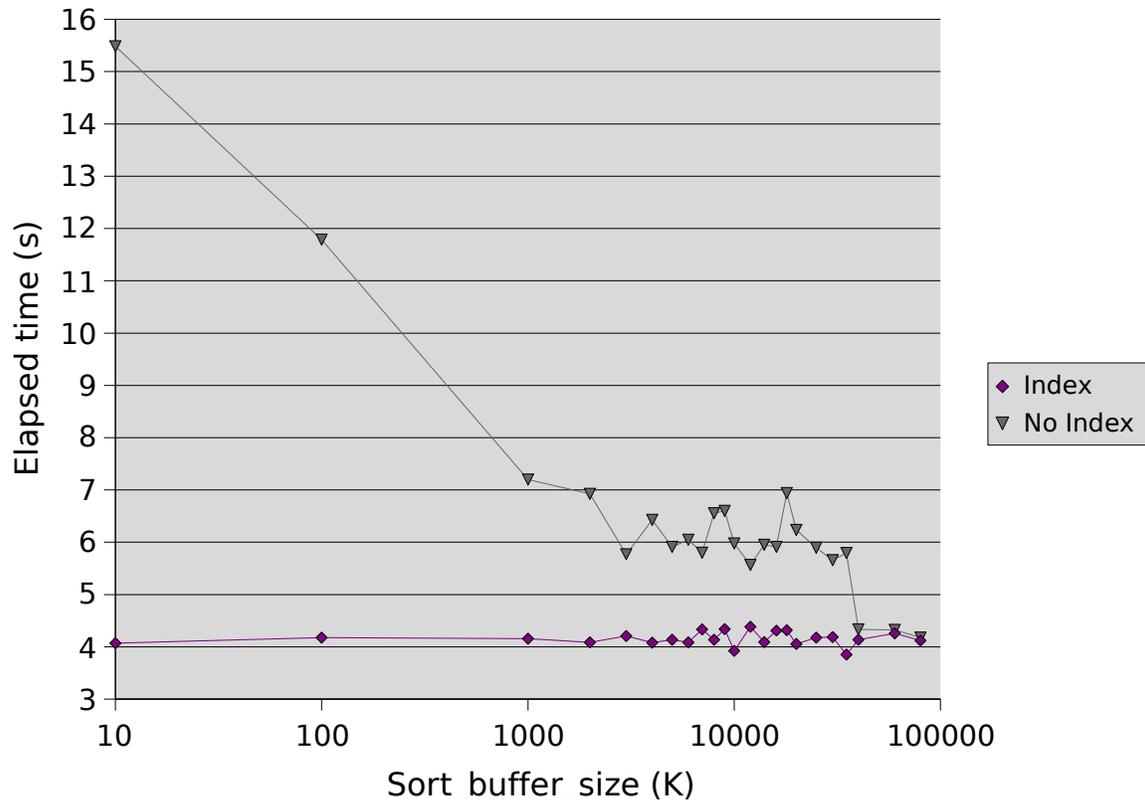


Illustration 3: Optimizing ORDER BY through increasing sort buffer size or creating an index

Tuning DML (INSERT, UPDATE, DELETE)

The first principle for optimizing UPDATE, DELETE, and INSERT statements is to optimize any WHERE clause conditions used to find the rows to be manipulated or inserted. The DELETE and UPDATE statements may contain WHERE clauses, and the INSERT statement may contain SQL that defines the data to be inserted. Ensure that these WHERE clauses are efficient – perhaps by creating appropriate concatenated indexes.

The second principle for optimizing DML performance is to avoid creating too many indexes. Whenever a row is INSERTed or DELETED, updates must occur to every index that exists against the table. These indexes exist to improve query performance, but bear in mind that each index also results in overhead when the row is created or deleted. For UPDATES, only the indexes that reference the specific columns being modified need to be updated.

Batching inserts

The MySQL language allows more than one row to be inserted in a single INSERT operation. Batching insert operations in this way can radically improve performance. shows how the time taken to insert 10,000 rows into table decreases as we increase the number of rows included within each INSERT statement. Inserting one row at a time, it took about 384 seconds to insert the rows. When inserting 100 rows at a time, we were able to add the same number of rows in only 7 seconds.

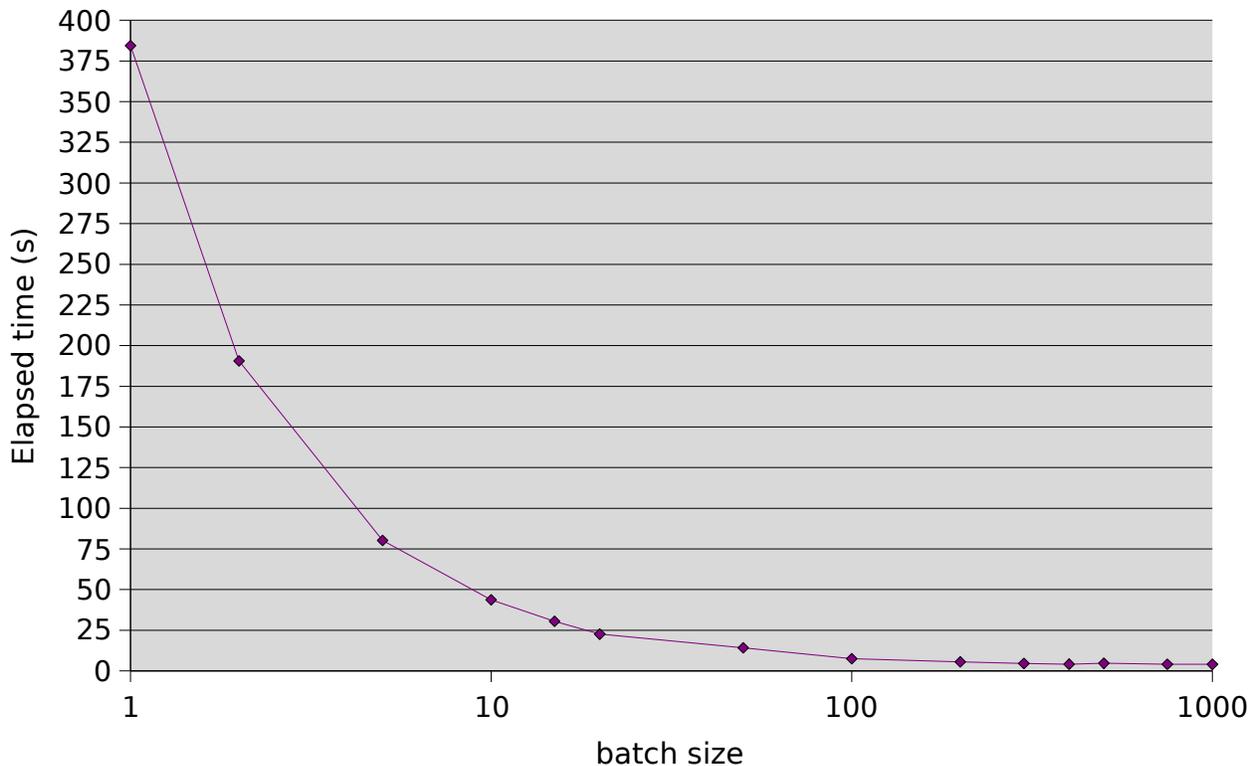


Illustration 4: Performance improvements from multi row inserts

Stored programs

In a previous article (cross-reference to my article on stored procedures) we discussed the performance implications of stored procedures (procedures, functions and triggers). To quickly summarize:

- Stored procedures can sometimes improve the performance of complex SQL by allowing a “divide and conquer approach”.
- Stored programs can also reduce network overhead of multi-SQL transactions.
- Stored programs are not, however, suitable for computationally expensive “number crunching”.
- As with MySQL *prepared statements*, stored procedures can reduce the overhead of the initial SQL statement parsing and optimization.

Conclusion

There's a lot more to SQL tuning than there was room for in this article, but I hope this gets you started. There's lots more information to be had on the web. The MySQL site itself (www.mysql.com) is a good start and there's a MySQL forum on performance at <http://forums.mysql.com/list.php?24>.

Guy Harrison is chief architect for Database Solutions at Quest Software (www.quest.com). This article was partially extracted from his book *MySQL Stored Procedure Programming* (O'Reilly 2006; with Steven Feuerstein). Guy can be contacted at guy.harrison@quest.com.

