# The City Lit Institute

##### **Department of Computing**

## Keeley Street, Holborn, London WC2B 4BA

**MySQL**

**(with Apache server)**

OPTIMIZATION

**LECTURER** : **ALEXANDER ADU-SARKODIE**

MSc. Telematics (IT & Telecom), MSc. Eng., Dip. Russ. Lang., Teach. Cert, AMIAEng (UK), MBCS (UK), MIfL(UK)

Blog: [**http://www.blogger.com/profile/14800490193632788559**](http://www.blogger.com/profile/14800490193632788559)

**Email:** [**aadusarkodie1@gmail.com**](mailto:aadusarkodie1@gmail.com)

**Optimizing MySQL**

1. **Normalization**

Database normalization is the process of **organizing** the fields and tables of a **relational database** to minimize **redundancy** and **dependency**. Normalization usually involves **dividing large tables into smaller (and less redundant) tables and defining relationships between them.** The objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships.

The logical design of the database, including the tables and the relationships between them, is the core of an optimized relational database. **A good logical database design can lay the foundation for optimal database and application performance. A poor logical database design can hinder the performance of the whole system**.

**Benefits of Normalization include:**

* **Reasonable** normalization frequently improves performance. When useful indexes are available, the SQL Server query optimizer is efficient at selecting rapid, efficient joins between tables.
* **Faster** sorting and index creation.
* **Narrower** and **more compact** indexes.
* **Fewer indexes per table**. This improves the performance of the INSERT, UPDATE, and DELETE statements.
* **Fewer null values** and less opportunity for inconsistency. This increases database compactness.

Note as normalization increases, the number and complexity of joins required to retrieve data also increases. **Too many complex relational joins between too many tables can hinder performance. Reasonable normalization frequently includes few regularly executed queries that use joins involving more than four tables.**

1. **Queries and Indexes**

More often than not you realise your database is just too **slow**. **Queries** are **queuing** up, **backlogs** growing, users being **refused connection**. Management is ready to spend millions on "**upgrading**" to some other system, when the problem is really that MySQL is simply not **being used properly**. **Badly defined** or non-existent MySQL **indexes** are one of the primary reasons for poor performance, and fixing these can often lead to **phenomenal improvements**. Consider an extreme example:

CREATE TABLE employee (  
 employee\_number char(10) NOT NULL,  
 firstname varchar(40),  
 surname varchar(40),  
 address text,  
 tel\_no varchar(25),  
 salary int(11),  
 overtime\_rate int(10) NOT NULL  
);

To find employee Fred Jone's salary(employee number 101832), you run: SELECT salary FROM employee WHERE employee\_number = '101832';

**MySQL has to search to find this record. It has no clue where to find this record. It doesn't even know that if it does find one matching, that there will not be another matching one, so it has to look through the entire table, potentially thousands of records, to find Fred's details.**

A MySQL **index** is a **separate file** that is sorted, and contains **only the field’s you're interested in sorting on**. If you create an index on employee\_number, MySQL can find the corresponding record very **quickly** (Indexes work in very similar ways to an **index in a book**. Imagine paging through a technical book (or more often, a scrambled pile of notes!) looking for the topic "Optimizing MySQL". An index saves you an immense amount of time!

1. **Repairing queries**

Using **EXPLAIN**:

EXPLAIN shows how your queries are **being used**. By putting it **before** a SELECT statement, you can see whether **indexes** are being used **properly**, and what kind of **join** is being performed.

For example:  
EXPLAIN SELECT employee\_number,firstname,surname FROM employee WHERE employee\_number = '10875';

We can illustrate the query above schematically below;

+----------+------+---------------+------+---------+------+------+------------+  
| table | type | possible\_keys | key | key\_len | ref | rows | Extra |  
+----------+------+---------------+------+---------+------+------+------------+  
| employee | ALL | NULL | NULL | NULL | NULL | 2 | where used |  
+----------+------+---------------+------+---------+------+------+------------+

So what are all these things?

* ***table*** shows us which table the output is about (for when you join many tables in the query)
* ***type*** is an important one - it tells us which type of **join** is being used. From **best** to **worst** the types are: system, const, eq\_ref, ref, range, index, all
* ***possible\_keys*** Shows which possible indexes apply to this table
* ***key*** And which one is actually used
* ***key\_len*** give us the length of the key used. The **shorter** the **better**.
* ***ref*** Tells us which column, or a constant, is used
* ***rows*** Number of rows mysql **believes** it must examine to get the data.
* ***extra*** Extra info - the bad ones to see here are "using temporary" and "using **filesort**"

Looks like our query is a shocker! There are **no possible keys to use**, so MySQL has to go through all the records (only 2 in this example, but imagine a really large table). Now let us add the index we talked about earlier.

If we re-run the **EXPLAIN**, we get:

+----------+-------+---------------+---------+---------+-------+------+-------+  
| table | type | possible\_keys | key | key\_len | ref | rows | Extra |  
+----------+-------+---------------+---------+---------+-------+------+-------+  
| employee | const | PRIMARY | PRIMARY | 10 | const | 1 | |  
+----------+-------+---------------+---------+---------+-------+------+-------+

The query above is a good one. The type of "join" is "**const**", which means that the table has **only one matching row**. The primary key is being used to find this particular record, and the number of rows MySQL thinks it needs to examine to find this record is 1. All of which means MySQL could have run this query thousands of times in the time it took you to read this little explanation.

1. **More optimisation techniques:**

*Making indexes* ***constants*** *in complex operations*

Some **knowledge** of how **indexes work** allows you to use them more efficiently. Firstly, note that when you **update** a **table** with an **index,** you have to **update the index table as well**, so there is a performance price to pay. **But unless your system runs many more inserts than selects and the inserts need to be quick, and not the selects, this is a price worth paying**.

What about if you want to select on more than one criteria? (As you can see, **it only makes sense to index those fields you use in the WHERE clause**.) The query:

SELECT firstname FROM employee;

makes no use of an index at all. An index on firstname is useless. But,

SELECT firstname FROM employee **WHERE** surname="Madida";

would benefit from an index on **surname**.

Let's look at some more complex examples where EXPLAIN can help us improve the query.

**We want to find all the employees where half their overtime rate is less than £20**.

Knowing what you do, you correctly decide to add an index on **overtime\_rate**, seeing as that's the column in the where clause.

ALTER TABLE employee ADD INDEX(overtime\_rate);

Now let's run the query.

EXPLAIN SELECT firstname FROM employee WHERE overtime\_rate/2<20;

+----------+------+---------------+------+---------+------+------+------------+

| table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----------+------+---------------+------+---------+------+------+------------+

| employee | ALL | NULL | NULL | NULL | NULL | 2 | where used |

+----------+------+---------------+------+---------+------+------+------------+

Not good at all! Every single employee record is being read. Why is this? The answer lies in the "**overtime\_rate/2"** part of the query. Every overtime\_rate (and hence every record) has to be read in order to divide it by 2. So we should try **and leave the indexed field alone, and not perform any calculations on it.** How is this possible? This is where your school algebra comes to the rescue! You know that **'x/2 = y' is the same as 'x = y\*2'**.We can **rewrite** this query, by seeing if the **overtime\_rate is less than 20\*2**. Let's see what happens.

EXPLAIN SELECT firstname FROM employee WHERE overtime\_rate<20\*2;

+--------+-------+---------------+---------------+---------+------+------+----------+

|table | type | possible\_keys | key | key\_len | ref | rows |Extra |

+--------+-------+---------------+---------------+---------+------+------+----------+

|employee| range | overtime\_rate | overtime\_rate | 4 | NULL | 1 |where used|

+--------+-------+---------------+---------------+---------+------+------+----------+

Much better! MySQL can perform the 20\*2 calculation once, and then search the index for this constant**. The principle here is to keep your indexed field standing alone in the comparison, so that MySQL can use it to search, and not have to perform calculations on it.**

**You could have constructed the query to read where overtime rate is less than a number. Simply make comparison operands or inputs constants!**

In the example above **ref** is **NULL** because there is no matching row. It is a **range**.

Ordering by surname is a common requirement, so it would make sense to create an index on surname. But in this example our employee table could consist of **thousands** of people who could have a surname say "Dlamini". So we need to index on firstname as well. The good news is that MySQL uses **leftmost prefixing, which means that a multi-field index A,B,C will also be used to search not only for a,b,c combinations, but also A,B as well as just A.**

In our example, this means that an index of the type

ALTER TABLE employee ADD INDEX(surname,firstname);

is used for a queries such as

EXPLAIN SELECT overtime\_rate FROM employee WHERE surname='Madida';

as well as

EXPLAIN SELECT overtime\_rate FROM employee WHERE surname='Madida' and firstname="Mpho";

## The Query Optimizer, OPTIMIZE and ANALYZE

i) The magic inside MySQL that decides **which keys**, if any, to use to in the query, is called the **query optimizer**. It takes a **quick glance** at the **index** to see which **indexes are the best to use.**

RDBMS is **non - procedural** because you **specify what needs to be done or achieved and not how to do it**. You don’t **need to define paths**. The system **automatically** finds it. The system has **optimisers** to find the data. The optimisers will **automatically search** for the **most efficient path to retrieve the desired results without any intervention from the user**.

By running a query like;

ANALYZE TABLE tablename;

**This stores the key distribution** for the table (running ANALYZE is equivalent to running myisamchk -a or myismachk --analyze).

ii)Many **deletes** and **updates** leave **gaps** in the table **(**especially when you're using **varchar,** or in particular **text/blob** fields**).** This means there are **more unnecessary disk I/O's,** as thehead needs to **skip** over these gaps when reading. Running

OPTIMIZE TABLE tablename

solves this problem. Both of these statements should be run **fairly frequently** in any well looked after system.

iii) Another factor that most people don't use when indexing is to take advantage of **short indexes**. You don't have to index on the entire field. Our surname and firstname fields are 40 characters each. That means the index we created above is **80 characters**. **Inserts** to this table then also have to write an additional 80 characters, and **selects** have 80 character blocks to manoeuvre around. Disk I/O is the primary hardware bottleneck. Try reducing the size of your index - in the example above, rather use.

ALTER TABLE employee ADD INDEX(surname(**20**),firstname(**20**));

Now our updates write to an index **half** the size, and selects have a smaller index to search. Both will be faster (**unless you make the indexes too short** - imagine a book index, instead of giving the full word, only contained the first letter of the word).

The same applies to the **original field definitions**. In these days of ample disk space, we don't often worry about space. But smaller usually means faster, so defining our surname and firstname fields as CHAR (255) would be a mistake if the biggest firstname is never more than 20 characters! **You don't want to cut names off, but remember that you can ALTER the field later if conditions change, and you need to allow for more characters**. I also suggest using **VARCHAR** rather than **CHAR** (variable length characters rather than fixed length characters), even though many don't recommend this as they are more subject to fragmentation. I overcome this by using OPTIMIZE often.

**5. Optimising inserts**

i) Most systems need to be highly **optimized for selects** - take a news site which performs millions of queries per day, but where the data arrives in large batches of text files. So for parts of the day, inserts need to be optimal, without noticeably affecting the millions trying to access the data.

The best way to insert the data is to use MySQL's "**LOAD DATA INFILE**". This is much faster (**20 times according to MySQL**),

The syntax is simple, and the code becomes a lot simpler too:

**$db->query("LOAD DATA INFILE 'datafile.txt' INTO TABLE employee (employee\_number,firstname,surname,tel\_no,salary) FIELDS TERMINATED BY '|'");**

LOAD DATA INFILE has defaults of:

FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'

And, just as with an ordinary insert, you need to **specify** a field list if the **order** of the fields is **different**, or, as in the example above, you're not inserting data for every field. Always specifying a field list is good practice for all queries anyway - if someone adds a field to the table at a later stage, you don't want to go back and have to fix all your previous INSERT and SELECT \* statements.

**If you can't get this to work properly, have a look at the format of your text file – most problems encountered with LOAD DATA have been because of a corrupted text file. Every field in every row must be delimited correctly!**

ii) **LOW PRIORITY** and **INSERT DELAYED:**: You may not always be inserting from a text file - perhaps your application needs to do many unrelated inserts continually. There are ways to make sure the mass of users selecting are not badly affected. The first is to use **INSERT LOW PRIORITY.** This **waits until there are no more reads waiting to happen, waiting for the gap, and not pushing in as it were**. Of course, if your database is a rush hour special, there may never be a gap, and the client performing the **INSERT LOW PRIORITY** may start to grow cobwebs! An alternative here is **INSERT DELAYED.** The client is immediately freed, and the **insert put into a queue** (with all the other INSERT DELAYED's still waiting for the queue to end). This means that there can be **no meaningful information passed back to the client**, (such as the auto\_increment value), as the INSERT has not been processed when the client is freed.

**Also, be aware that a catastrophe such as an unexpected power failure here could result in the queued INSERT's being lost. For neither of these methods do you have any idea when the data will be inserted, if at all, so I suggest you use with caution.**

## Conclusion

It's not only getting the data **in** that needs to be quick - sometimes you need to get it **out** quickly too. (Say you've accidentally loaded yesterday's classified ads, for example). Don't do a:

DELETE FROM classifieds;

Rather, use:

**TRUNCATE** TABLE classifieds;

The difference here is that **DELETE drops records one by one**, and that can be 1 million one by one's too slow! Note that this does not apply before version 4.0 of MySQL. If you do a DELETE FROM tablename on a non-empty table, and get 0 records back as a result, you're running an earlier version. To fix this problem, MySQL made DELETE remove records one by one so as to return the number of records deleted, but TRUNCATE still did the quick delete. Also, earlier versions than 3.23.33 used TRUNCATE tablename, not TRUNCATE TABLE tablename)

Don't forget **EXPLAIN**! Often the best way is to try and **rewrite the query in as many different ways as possible, and see which one runs more efficiently**. You can learn a lot by trying to figure out why one alternative ran faster than the other.

**7. The 15 Commandments!**

1. **Proper use of indexes** improve performance
2. Do not perform **calculations on an index** (eg: if you have an index for a column called salary, do not perform calculation such as salary \* 2 > 10000)
3. “**LOAD DATA INFILE**” is the fastest way to insert data into MySQL database (20 times faster than normal inserts)
4. Use **INSERT LOW PRIORITY** or **INSERT DELAYED** if you want to delay inserts from happening until the table is free
5. Use **TRUNCATE TABLE** rather than **DELETE FROM** if you are deleting an entire table (DELETE FROM delete row by row, whereas TRUNCATE TABLE deletes all at once)
6. Always use **EXPLAIN** to examine if your select query is efficient
7. Use **OPTIMIZE TABLE** to reclaim **unused space** (Note: Table will be **locked** during optimization, so only do it **during low traffic time**), and **ANALYZE** to trigger key distribution of queries.
8. **Better to have 10 quick queries than 1 slow one**
9. Use **caching** to reduce database load
10. **Normalize** tables to ensure data consistency
11. Use **persistent** connections
12. Don’t query columns you don’t need, **avoid using SELECT \* FROM**
13. MySQL can search on prefix of indexes (ie: If you **have index INDEX (a,b),** you don’t need an index on (a)). **Leftmost prefixing**
14. Don’t use **HAVING** when you can use **WHERE**
15. Use **numeric** values **rather than alphabetical values** when performing a join.