Optimized Pagination using MySQL

October 24th, 2011.

Dealing with large data sets makes it necessary to pick out only the newest or the hottest elements and not displaying everything. In order to have older items still available, Pagination navigation's have become established. However, implementing a Pagination with MySQL is one of those problems that can be optimized poorly with MySQL and certainly other RDBM systems. However, knowing the underlying database can also help in optimizing pagination queries, because there is no real copy and paste solution.

There are rattling around many alleged optimized ways on the web to do a fast pagination, but let's start with the worst query which is used very often, though:

```
SELECT *
FROM city
ORDER BY id DESC
LIMIT 0, 15
```

Which is done in 0.00 sec. So, where is the problem? Actually, there is no problem with this query and their parameters, because the primary key of the following table is used and only 15 elements get read:

```
CREATE TABLE city (
  id int(10) unsigned NOT NULL AUTO_INCREMENT,
  city varchar(128) NOT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB;
```

The real problem are clicks on sites with a large offset, like this:

```
SELECT *
FROM city
ORDER BY id DESC
LIMIT 100000, 15;
```

Which takes about 0.22 sec on my data set with about 2M rows. An *EXPLAIN* shows, that 100015 rows were read but only 15 were really needed and the rest was thrown away. Large offsets are going to increase the data set used, MySQL has to bring data in memory that is never used! We could assume that most users just click around on the lower sites, but even a small number of requests with large offsets may endanger the entire system. Facebook has recognized this and doesn't optimize the database for many requests per second but to **keep the variance small**. With this in mind, we shouldn't take this loss and should use a different approach. Anyway, with pagination queries also another information is needed for the page calculation: the total number of elements. Well, you could get the number with a separate query like this very easily:

```
SELECT COUNT(*)
FROM city;
```

However, this takes 9.28 sec on my InnoDB table. An (inappropriate) optimization for this job is *SQL_CALC_FOUND_ROWS*, which reduces the calculation and the fetch into one query. But keeping the queries simple and short doesn't result in a performance gain in most cases. So, lets see how this query performs, which unfortunately is used in some major frameworks as the standard pagination routine:

```
SELECT SQL_CALC_FOUND_ROWS *
FROM city
ORDER BY id DESC
LIMIT 100000, 15;
```

Ouch, we've doubled the time to 20.02 sec. Using *SQL_CALC_FOUND_ROWS* for pagination is the worst idea, because there is no *LIMIT* optimization: ALL rows must be read for counting and just 15 rows get returned. There are also tips around to ignore indexes in order to perform faster. This isn't

true, at least when you need to sort the table. The following query takes about 3 minutes:

```
SELECT SQL_CALC_FOUND_ROWS *
FROM city
IGNORE INDEX(PRIMARY)
ORDER BY id DESC
LIMIT 100000, 15;
```

If you need further information of when to use SQL_CALC_FOUND_ROWS and when not, take a look at the article on **MySQL Performance Blog**.

Okay, let's start with the real optimization. There is a lot to do in order to optimize Pagination queries. I'll split up the article into two sections, the first covering how we can get the number of resulting rows and the second to get the actual rows.

Calculate the number of rows efficiently

In the case you want to paginate a cache table where you still use MyISAM, you can run a simple *COUNT(*)* query in order to get the number of rows. Also HEAP tables store the absolute number of rows in their meta information. More complicated, however, it is for transactional storage engines like InnoDB, where different numbers of rows exist at any time.

If you insist, that the pagination is always based on the correct number of rows, cache the value somewhere and update it periodically via a background process or when the cae must invalidated by an user action under the usage of an explicit NOT NULL index using *USE INDEX*, like:

```
SELECT COUNT(*)
FROM city
USE INDEX(PRIMARY);
```

If writes are no problem for you, you could also add an aggregate table which is maintained with INSERT and DELETE triggers or multi_query's in order to save some latency.

But do we really need an exact number of elements? Especially for really big data sets? Does it interest you if there are 38211 elements instead of 39211 on a random site you're visiting somewhere on the net? Considering this, we could approximate the number just as well and output something like "40 to 80 of over 1000" in the user interface. This naturally requires a prevention of jumping to the last page and a rethinking of the paginator layout.

If your data set is really huge, I would recommend to use a kind of infinity pagination, as I implemented it with my **jQuery Pagination plugin** (negative number of elements set it to infinity).

If you want to build a pagination for search results, it's generally the case that the important stuff should be on the first page. If this isn't the case, optimize your search quality rather than optimizing the pagination in order to allow users browsing the whole result set. In the following, I'd like to focus more on gathering a good **estimation** for a result, which also has relevance for search results.

When you need an estimation of the number of rows of an table rather than a subset, a good starting point can be a SHOW query, which executes quite quickly:

```
SHOW TABLE STATUS LIKE 'city';
```

Another idea would be using the cardinality of a column with unique elements, like an *auto_increment* column:

```
SHOW INDEX FROM city;
```

However, you don't need the whole table in most cases. And if so, it's certainly a cache table with MyISAM, where you can run a fast *COUNT(*)*. For a good estimation of only a part of the query, try to take the output of *EXPLAIN* into account, e.g.:

```
EXPLAIN SELECT *
FROM city
WHERE id < 5000
ORDER BY id DESC
LIMIT 300, 15;
```

In this example the correctness of the estimation is about 99.91%, but there are cases where an estimation can have a deviation of about 15% and more. Mark Callaghan suggested to implement a **fast COUNT function** for InnoDB as a new *ESTIMATED_COUNT()* function. I would be glad to see

COUNT(ESTIMATE *) in preference to his approach if the parser must be modified, as we already have the *DISTINCT* modifier and another flag looks quite natural.

Another estimation approach for the number of rows of a table is using the *information_schema*. I abuse this meta information schema for optimizations very heavily in the last time, as you'll see in further articles. So, if a table doesn't get deletes, we could use the *auto_increment* value as the number of rows and are done:

```
SELECT auto_increment
FROM information_schema.tables
WHERE table_schema=DATABASE()
AND table_name='city';
```

If you have gaps in your table and especially in your auto_increment range, try to figure out what percentage of gaps you have:

```
SELECT COUNT(*) / MAX(id)
FROM city
USE INDEX(PRIMARY);
```

Cache this value somewhere and use it for estimation. The following query illustrates the usage with a more complete example:

```
SELECT @rows:= FLOOR(auto_increment * $pct) AS rows,
       @estimate:= @rows - @rows % 500 AS estimate,
       @estimate < @rows AS more
FROM information_schema.tables
WHERE table_schema=DATABASE()
AND table_name='city';
```

This query returns a good estimation based on the auto_increment value and the percentage of gaps in the range. Additionally, you get the number rounded more user friendly and you'll get a column called "more", which indicates if there are further elements to show a "and more" or something like this.

Get the elements

Okay, we get to the more important part of this article, the retrieval of the page elements. As indicated above, large offsets slow down the entire system, thus we have to rewrite the queries in order to make usage of an index. As an illustration I create a new table "news", where we sort by topicality and realize an efficient pagination on it. For simplicity, we suppose the newest elements also have the highest ID:

```
CREATE TABLE news(
   id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
   title VARCHAR(128) NOT NULL
) ENGINE=InnoDB;
```

A very fast approach is using a query which is based on the last ID the user has seen. The query for the next page looks like this, where you have to pass the id of the last element on a page:

```
SELECT *
FROM news WHERE id < $last_id
ORDER BY id DESC
LIMIT $perpage
```

The query for the previous page looks similar, where you have to pass the id of the first element on a page and sort in reverse order (sure, you havee to sort the resulting rows again):

```
SELECT *
FROM news WHERE id > $last_id
ORDER BY id ASC
LIMIT $perpage
```

The problem with this approach is, it's good for a "older articles" link in the footer of a blog or when you reduce your pagination to "next" and "previous" buttons. If you want to generate a real pagination navigation, things get tricky. An idea would be getting more elements and pick out the Id's like this:

```
SELECT id
FROM (
   SELECT id, ((@cnt:= @cnt + 1) + $perpage - 1) % $perpage cnt
   FROM news
   JOIN (SELECT @cnt:= 0)T
   WHERE id < $last_id
```

```
   ORDER BY id DESC
   LIMIT $perpage * $buttons
)C
WHERE cnt = 0;
```

This way you get an offset id for every button you want to show in the user interface. A small note for my jQuery Paging plugin, I've mentioned earlier; every "block" element has an own ID, which is more or less the position of the element in the navigation. But you can also use this for an array index like so:

```
var offsets = [/* past your id list here */];
...
onFormat: function(type) {

 switch (type) {
 case "block":
  return '<a href="?offset=' + offsets[this.pos] + '">' + this.value + '</a>';
  ...
 }
}
...
```

Another big advantage over using page numbers or pre-calculated slices from the plugin is that users ever have their pagination consistent. Imagine, you publish a new article on your site. All of your articles are shifted one position ahead on the sites. This is a problem if a user changes the site while you publish something new, because she will see one article twice. With a fixed ID offset, this problem is solved as a nice side effect. Mark Callaghan published an **analogical post** where he makes use of a combined index and two position variables but the same basic idea.

If the records are relatively rigid, you could also just save the page number within the table and create an appropriate index on that column. Also a cache table for different $perpage values would be possible, which you can join for a page selection. If there is a new entry added to that table once in a while, you could simply run this query to regenerate the page number cache:

```
SET @p:= 0;
UPDATE news SET page=CEIL((@p:= @p + 1) / $perpage) ORDER BY id DESC;
```

You could also add a new pagination table, which can be maintained in the background and rotated if the data is up:

```
UPDATE pagination T
JOIN (
   SELECT id, CEIL((@p:= @p + 1) / $perpage) page
   FROM news
   ORDER BY id
)C
ON C.id = T.id
SET T.page = C.page;
```

Getting your page elements get's really trivial now:

```
SELECT *
FROM news A
JOIN pagination B ON A.id=B.ID
WHERE page=$offset;
```

With my database class, I've published another slightly different approach, which can be used for relatively small data sets, but where really no index can be used - for example in search results. On a common server the following queries took about 2 seconds with 2M records. You could use this query in the background to generate caches or as I said on smaller data sets. On limited result sets also higher concurrence should be attainable. The approach is quite simple, I create a temporary table in which I store all the Id's - which in turn is also the slowest part of this approach. But take a look at this where I sort the result by a dynamically generated random column:

```
CREATE TEMPORARY TABLE _tmp (KEY SORT(random))
SELECT id, FLOOR(RAND() * 0x8000000) random
FROM city;

ALTER TABLE _tmp ADD OFFSET INT UNSIGNED PRIMARY KEY AUTO_INCREMENT, DROP INDEX SORT,
ORDER BY random;
```

In the next step, you can execute your original paginated query like this:

```
SELECT *
```

```
FROM _tmp
WHERE OFFSET >= $offset
ORDER BY OFFSET
LIMIT $perpage;
```

BTW: If you use MyIsam for this temporary table, which is the case on MySQL - MariaDB uses InnoDB! - you could simply run a COUNT(*)-query on it to get the exact number of elements. Win-Win with the cost of one expensive copy query. But if the table isn't user specific, you could also generate such tables as cache table for all users and implement a simple table rotation.

Think different

Okay, we slowly come to the end, but one last question: Isn't this is a huge effort for a simple problem? Yeah, indeed, but it could be easier, if you don't follow the standard paradigm. Just because everyone lays out such pages in the same way doesn't mean you need to. Frank Denis has written a **great article** on this subject that page numbers should work the same way like book pages. The first pages get the smallest numbers and the last pages get the highest numbers. In that way you have an indexable permalink structure for search engines, it can be cached and the page number can be stored directly in the table as described above for the caching approach, however, there is no need to rebuild this column at any time.

I need to mention my Pagination plugin again, because it's also very easy to build a paginator which delivers the inverse number to the backend. The only difference is the subtraction in the *onSelect*-callback:

```
onSelect: function(page) {

 page = 1 + this.pages - page;

 $.ajax({
  "url": '/data.php?&page=' + page,
  "success": function(data) {
   // do something unexpected
  }
 });
}
```

On the MySQL-side the only thing is needed is this:

```
SELECT *
FROM (
   SELECT *
   FROM news
   WHERE page=$page
   LIMIT $perpage
)T
ORDER BY id DESC;
```

As you can see there are many possibilities, it depends on the specific use cases and the complexity of your sorting. If it is quite rigid, one can realize a simple and powerful pagination very quickly.

You might also be interested in the following

- **jQuery Pagination revised**
- **Optimal index size for variable text in MySQL**
- **Running Standard Deviation in MySQL**
- **Flag based COUNT using MySQL**

6 Comments on *„Optimized Pagination using MySQL”*

saaj
commented after 13 weeks
It's really good article on the subject of MySQL pagination. Though I'd also add this presentation http://www.slideshare.net/suratbhati/efficient-pagination-using-mysql-6187107

Øystein Grøvlen
commented after 3 days

The LIMIT optimization for SQL_CALC_FOUND_ROWS only applies when filesort is required. It uses a priority queue to keep track of the top N rows. The rows that are considered for insertion into the priority queue are counted. Hence, the total number of rows has been calculated during the execution of the query. The major advantage from earlier is that the entire result set is no longer sorted; only the rows needed for the result. Of course, if no sorting is required, CALC_FOUND_ROWS will still be more expensive since all rows will have to be inspected.

Robert Eisele
commented after 2 days
Øystein,

thanks for your hint about the LIMIT optimizations in 5.6. How does the query plan for such queries now look like? Or how did you actually optimized that kind of queries? Do you split the query into two seperate questions if it is indicated and merge the result at the end?
As one can discern from the article, optimizations sometimes are very tied to the SE used.

Robert

Robert Eisele
commented after 2 days
Uli,

I think some users stray around on such deep links, too. But above all bots, not blessed with too much intelligence are a more serious problem for the stability of such a system.

Killing threads is no option imco ;) I think MySQL should optimize large offsets by using an underlying index the same way we have to do it manually. I'm not that familar with the internals of MySQL, but wouldn't it be enough to optimize the cursor handling to simply skip the rows? Or better: Create heuristics about the table where the rows "could" rest and make a full scan only over a small portion of the entire set. That is assuming an index is used and we try to avoid a full index scan.

Robert

Øystein Grøvlen
commented after one day
Note that MySQL 5.6 has added limit optimization for SQL_CALC_FOUND_ROWS. See http://didrikdidrik.blogspot.com/2011/04/optimizing-mysql-filesort-with-small.html

Uli Stärk
commented after 10 hours
Users wont click all next-pages and generate high offsets - but search engines will! They cause most of the load by indexing weird data :)

"An EXPLAIN shows, that 100015 rows were read but only 15 were really needed and the rest was thrown away. Large offsets are going to increase the data set used, MySQL has to bring data in memory that is never used!"

I noticed this, too. Some board administrators simply close long threads after a load of posts for this reason ;) But why would the DB do this? The index (in your example) has all information needed to locate the result row pointers. It should not use the row-data and skip them.

Sorry, comments are closed for this article. Contact me if you have an inventive contribution.
