Insight for DBAs Insight for Developers MySQL

Subscribe to RSS Feed

Full Table Scan vs Full Index Scan Performance

November 23, 2012 Stephane Combaudon

Earlier this week, Cédric blogged about how easy we can get confused between a covering index and a full index scan in the `EXPLAIN` output. While a covering index (seen with `EXPLAIN` as `Extra: Using index`) is a very interesting performance optimization, a full index scan (`type: index`) is according to

the documentation the 2nd worst possible execution plan after a full table scan.
If it is obvious that a full table scan is not good for performance, how much can we expect if we can switch to a full index scan? In other terms, is a full table scan always the worst possible execution and should it be avoided at all costs?

Let's take the employees database, and slightly modify the employees tables:

```
mysql> ALTER TABLE employees ADD INDEX idx_first (first_name),ENGINE=InnoDB;
```

And then let's consider this query:

```
SELECT * FROM employees ORDER BY first_name;
```

This query can of course by executed by running a full table scan, but we could also take advantage of the idx_first index, which will translate to a full index scan.

Let's see how the optimizer will execute it:

```
mysql> EXPLAIN SELECT * FROM employees ORDER BY first_name\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 300363
        Extra: Using filesort
```

Surprising? The optimizer preferred a full table scan, and it did not even consider scanning the idx_first index as a relevant choice (`possible_keys: NULL`).

What do we get if we force the optimizer to use the index?

```
mysql> EXPLAIN SELECT * FROM employees FORCE INDEX(idx_first) ORDER BY first_name\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: index
possible_keys: NULL
          key: idx_first
      key_len: 16
          ref: NULL
         rows: 300363
        Extra:
```

Honestly, it looks better: the number of rows is the same, but a full index scan instead of a full table scan and no filesort. But predicting how a query will perform by only looking at the execution plan is not enough, we must run both queries and compare execution time.

First case: the employees table does not fit in memory
With the full table scan, the query runs in about **4s**.
With the full index scan, it runs in about **30s**.

So the optimizer was right after all. But why? Simply because all access patterns are not equal. When we are doing a full table scan, we are doing sequential reads, which are quite fast even with slow disks. But when we are using the index, we first have to do a full index scan (fast sequential reads, no problem) and then lots of random reads to fetch rows by index value. And random reads are orders of magnitude slower than sequential reads.

The optimizer has this knowledge, so it is able to pick the right execution plan.

Second case: the employees table fits in memory
With the full table scan, the query runs in about **3.3s**.
With the full index scan, the query runs in about **2.6s**.

We can see here a limitation of the optimizer: it does not know on which kind of media data is stored. If it is stored on spinning disks, the assumption that random reads are much slower than sequential reads is correct, but it is not the case anymore if data is stored in memory. That's why when execution plans look similar, you should always execute the query to really see which execution plan should be chosen. Here if we know that the table will always be in memory, we should add the `FORCE INDEX` hint to ensure optimal response time.

Now let's modify the query by selecting only the first_name field instead of selecting all the fields:

```
mysql> explain SELECT first_name FROM employees ORDER BY first_name\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: index
possible_keys: NULL
          key: idx_first
      key_len: 16
          ref: NULL
         rows: 300584
        Extra: Using index
```

The optimizer chooses the full index scan. It is a good choice because the index now covers the query, which means that reading the index is enough to get the results.

Conclusions:

- For a non-covering index, the difference between a full table scan and an execution plan based on a full index scan is basically the difference between sequential reads and random reads: it can be close if you have fast storage or it can be very different if you have slow storage.
- A full index scan can become interesting when the index is covering.
- Don't forget to measure response time when you are trying different execution plans. It is too easy to get focused on having a good-looking execution, but the end user only cares about response time!

About the Author

Stephane Combaudon
Stéphane joined Percona in July 2012, after working as a MySQL DBA for leading French companies such as Dailymotion and France Telecom.In real life, he lives in Paris with his wife and their twin daughters. When not in front of a computer or not spending time with his family, he likes playing chess and hiking.

Share This Post!

17 COMMENTS Oldest

Peter Laursen
12 years ago

So shouldn't we request an option for the slow log to 'log queries not using a covering index'?

+ 0 —

Stephane Combaudon Author
12 years ago

Peter,

Why not? But that would probably be lots of queries!
So it should rather be 'log queries using a full index scan but not using a covering index' 🙂

+ 0 —

Sergei Golubchik

12 years ago

“When we are doing a full table scan, we are doing sequential reads,”

No, we are not. It’s InnoDB storage engine, the data are stored in a B-Tree, by the primary key. There is no way to do a sequential table scan in InnoDB, a “table scan” is always an index scan, a scan by the primary key.

+ 0 —

Peter Laursen

12 years ago

My point was that the option currently available (‘log queries not using an index’) does not distinguish covering and non-covering indexes. It could make sense to log those using a non-covering index and not log those using a covering index. At least in some scenaarios. This is how I understand results from this blog of yours.

Besides – the logging options available (threshold for when a query is considered slow, option to log queries not using an index) are OR’ed. There is no way to AND the conditions as I see. This result in – if both options are used – a lot of ‘noise’ in the log (what will need to be filtered client side when analyzing logs), unnessary I/O and excessive disk usage

+ 0 —

Justin Swanhart

12 years ago

I'd never expect to see type:index outperform type:all UNLESS you see "using index". This is because type:index without "using index" defeats the purpose of type:index, which is to only scan the index and not access the table data. If you force the optimizer to use type:index to access columns that aren't in the index you are asking for more work. It doesn't matter if the data is sequentially accessed or not. A FTS on InnoDB is a scan over the primary key. Scanning an entire table via the secondary key is a scan over the secondary key plus an o(log(n)) lookup for every row to actually get the data. That is WAY more expensive (as seen in your tests).

The only case where type:index is faster w/out 'using index' is when count(*) is used. This is because count(*) doesn't REALLY access every column in the table, so the index can be used for the count without actually going back for the table data, so it is "using index" implicitly.

Here are some examples.
http://sqlfiddle.com/#!8/25865/8

+ 0 —

Peter Zaitsev Admin
12 years ago

Sergei,

Indeed Innodb table may be fragmented so full table scan will not be sequential, yet same is true for MyISAM which over time will get fragmented rows. Freshly created Innodb table will most likely result in a lot of sequential reads.

Though even in the most fragmented case there is a major difference in primary index scan vs other index scan. The data is at least "clustered" in primary key so you will scan all data from 16K page in order. If you're doing index scan changes are the same 16K index page will contain 500 pointers causing up to 500 different pages to be fetched. For row size of 100 you might be looking at 10-100 times of difference in number of iops

+ 0 –

Jordan Ivanov
12 years ago

Is there option for full table scan in physical order?

I mean to scan table sequential and to ignore internal nodes.

+ 0 –

Peter Zaitsev Admin
12 years ago

Jordan,

Not for Innodb. It is not as simple as it might sound without table level locks as while you're scanning the table the new data might be well inserted in the portion you already scanned. There have been talks though about using such table scan for quick dump functionality when table can be temporary made read only, though emerging SSD technologies make random IO less and less issue.

+ 0 –

Stephane Combaudon Author
12 years ago

Peter Laursen,

Yes, being able to distinguish full index scans using a covering index and full index scans not using a covering index could be nice. I first thought you wanted to log ALL queries that are non covered by an index.

Regarding the logging options, if you enable log_queries_not_using_indexes, it will log all full table scans and full index scans, even if the execution time if below long_query_time.

0

Stephane Combaudon Author
12 years ago

Justin,

I agree with you, but that's counter-intuitive (at least in my opinion), that's why I decided to write this post.

0

Cédric
12 years ago

Interesting, thanks for the post.

0

Dan Harkness
12 years ago

Justin / Stephane,

I am a bit confused. If I'm reading Justin's comment correctly, it is saying that outside of COUNT(*) the use of a non-covering index is always worse than a full table scan? While this may be true in a case where you are selecting all data (or even a significant portion in some cases), wouldn't using the index still typically be faster in other cases due to the elimination of the need to fetch all rows? Please tell me I simply misread Justin's comment.

Thanks

+ 0 −

Justin Swanhart
12 years ago

Hi Dan,

For most queries with filter conditions, b-tree indexes provide fast binary search which is faster than scanning the whole table.

Type:index is only used when there is no criteria that can be used as filter conditions, but an index still contains all the columns used by the query. The index can be scanned instead of the table. This saves on cost because (at a minimum) there should be less total data in the index pages compared to the actual table pages.

create table t (
c1 int primary key,
c2 int,
c3 int,
…,

c10 int
key(c3, c10)
)

— should use type: index because we would need to FTS
— BUT we have an index covering all of the requested columns
select distinct c3 from t where c10 = 5;

— will use type:ref (non-unique index lookup) NOT type:index
select distinct c10 from t where c3 = 5;

— will do a index lookup (type:ref, but not use using:index)
select distinct c2 from t where c3 = 5;

— will do a FTS (type:index would be slower because the c2 column is not in the index)
select distinct c2 from t where c10 = 5;

+ 0 —

Dan Harkness
12 years ago

Justin,
Thanks for the response. That helped confirm what I thought. I just misinterpreted your first post. However, I think that in your last example, the reason for the FTS is not because the c2 column isn't in the index, but rather because the where condition is filtering on c10, which is not the first column of any index (after all the c2 column is in the previous query, which you correctly noted would have type:ref).

If I'm not mistaken this all also depends on the uniqueness of the values in the c3 and c10 columns, as if c3 = 5 in 99% of the rows then using the index isn't going to be very preferable anyway (at least certainly not if it's a non-covering index)

+ 0 −

shenglin
12 years ago

Hi Peter,

Could you please help explain how can caused the fragment with full table scan? I have a case, the query "select count(*) from table(innodb);". Checking the plan, it uses primary index scan. however, it will do the sequential read only 2-3M read/sec and 100-300 read/sec. I used blktrace to check and find it read 32 block each time, not sequential read. n other environment, the similar query can do over 200M read/sec. Could you please help me explain what can cause this?

Thanks

+ 0 −

Sergio
10 years ago

Nice article, thanks to it I now understand why the optimizer chooses to do a Full Table scan when using a index seems to be the better choice.

+ 0 −

Ankur Rathore

9 years ago

thanks for the above article… I have a question please give some insight…

You use a save point during a transaction when
a) writing a compensating transaction
b) working with a temporal data set
c) there is a low probability of failure
d) there is a high probability of failure
e) multiple users are updating the same table

what will be the best option and correct option for above question..

+ 0 —