MySQL查看索引使用情况及优化

参考：http://www.cnblogs.com/kucha/p/4838895.html

https://www.percona.com/blog/2012/12/05/quickly-finding-unused-indexes-and-estimating-their-size/

mysql中支持hash和btree索引。innodb和myisam只支持btree索引，而memory和heap存储引擎可以支持hash和btree索引
我们可以通过下面语句查询当前索引使用情况：

```
show status like '%Handler_read%';
+-----------------------+-------+
| Variable_name | Value |
+-----------------------+-------+
| Handler_read_first | 0 |
| Handler_read_key | 0 |
| Handler_read_last | 0 |
| Handler_read_next | 0 |
| Handler_read_prev | 0 |
| Handler_read_rnd | 0 |
| Handler_read_rnd_next | 0 |
+-----------------------+-------+
```

如果索引正在工作，则Handler_read_key的值会很高，这个值代表一个行被索引值读的次数，很低值表名增加索引得到的性能改善不高，因此索引并不经常使用
如果Handler_read_rnd_next值很高意味着查询运行效率很低，应该建立索引补救，这个值含义是在数据文件中读取下一行的请求数。如果正在进行大量表扫描，Handler_read_rnd_next的数值将会很高。说明索引不正确或者没有利用索引。

优化：
优化insert语句：
1.尽量采用 insert into test values(),(),(),()...
2.如果从不同客户插入多行，能通过使用insert delayed语句得到更高的速度，delayed含义是让insert语句马上执行，其实数据都被放在内存队列中个，并没有真正写入磁盘，这比每条语句分别插入快的多；low_priority刚好相反，在所有其他用户对表的读写完后才进行插入。
3.将索引文件和数据文件分在不同磁盘上存放（利用建表语句）
4.如果进行批量插入，可以增加bulk_insert_buffer_size变量值方法来提高速度，但是只对MyISAM表使用
5.当从一个文本文件装载一个表时，使用load data file，通常比使用insert快20倍
优化group by语句：
默认情况下，mysql会对所有group by字段进行排序，这与order by类似。如果查询包括group by但用户想要避免排序结果的消耗，则可以指定order by null禁止排序。
优化order by语句：
某些情况下，mysql可以使用一个索引满足order by字句，因而不需要额外的排序。where条件和order by使用相同的索引，并且order by的顺序和索引的顺序相同，并且order by的字段都是升序或者降序。
优化嵌套查询：
mysql4.1开始支持子查询，但是某些情况下，子查询可以被更有效率的join替代，尤其是join的被动表待带有索引的时候，原因是mysql不需要再内存中创建临时表来完成这个逻辑上需要两个步骤的查询工作。

I had a customer recently who needed to reduce their database size on disk quickly without a lot of messy schema redesign and application recoding. They didn't want to drop any actual data, and their index usage was fairly high, so we decided to look for unused indexes that could be removed.

Collecting data

It's quite easy to collect statistics about index usage in Percona Server (and others) using the User Statistics patch. By enabling 'userstat_running', we start to get information in the INFORMATION_SCHEMA.INDEX_STATISTICS table. This data collection does add some overhead to your running server, but it's important to leave this running for a good long while to get a good dataset that is representative of as much of your workload as possible.
If you miss collecting index stats while some occasional queries run, you run the risk of dropping indexes that are being (seldomly) used, but are still important for the health of your system to have. This may or may not impact you, but I'd highly recommend you manually review the list of unused indexes being generated above before you simply drop them.
Depending on your sensitivity to production load, you may therefore want to run this several days, or just sample different short windows during your normal production peak. In either case, you may want to compare or repeat this index analysis, so let's setup a separate schema to do this. Its important that this index analysis is on a server with your full production dataset loaded, but it could be a master, or just a slave

somewhere (just be careful not to break replication!).

Shell

```
mysql> create schema index_analysis;
```

If our index_statistics are collecting on the same server, then we can simply get a snapshot of it into our schema with one command:

Shell

```
mysql> create table index_analysis.used_indexes select * from information_schema.index_statistics;
```

If the stats come from some other server, then you may need to dump and load a copy of that table into your working index_analysis schema.

Merging stats from several servers

In the case of this client, they had a master and several slaves taking read traffic. The index workload on these two sets of servers was different and I wanted to make sure I considered the index statistics from both of these sources. Be sure you include all relevant index stats from all aspects of your application, otherwise garbage-in, garbage-out and you risk dropping necessary indexes.

To accomplish merging multiple result sets, I gathered statistics from both their master and slave and loaded them into my schema as separate tables. Then I simply created a view of a UNION DISTINCT of those two tables:

Shell

```
mysql> create view used_indexes as
 (select TABLE_SCHEMA, TABLE_NAME, INDEX_NAME from master_index_stats )
 UNION DISTINCT
 (select TABLE_SCHEMA, TABLE_NAME, INDEX_NAME from slave_index_stats)
 ORDER BY TABLE_SCHEMA, TABLE_NAME;
```

Now I can query the 'all_known_index_usage' and see the union of both of those datasets. This, of course, can be extended to all the datasets you want.

Interpreting the data

So, this is all well and good, but how do we then easily determine a list of indexes that are **not**being used? Well, for this we need to back to the INFORMATION_SCHEMA to get a list of ALL the indexes on my system (or at least the schemas I want to consider dropping indexes in).

Let's keep using views so this dynamically updates as our schema changes over time:

Shell

```
mysql> create view all_indexes as
select
 t.table_schema as TABLE_SCHEMA,
 t.table_name as TABLE_NAME,
 i.index_name as INDEX_NAME,
 i.NON_UNIQUE as NON_UNIQUE,
 count(*) as COLUMN_CNT,
 group_concat( i.column_name order by SEQ_IN_INDEX ASC SEPARATOR ',') as COLUMN_NAMES
from
 information_schema.tables t join information_schema.statistics i using (table_schema, table_name)
where
 t.table_schema like 'sakila%'
group by
 t.table_schema, t.table_name, i.index_name;
```

Now I can query this view to see my indexes:

Shell

```
mysql> select * from all_indexes limit 1 \G
*************************** 1. row ***************************
TABLE_SCHEMA: sakila
TABLE_NAME: actor
INDEX_NAME: idx_actor_last_name
NON_UNIQUE: 1
COLUMN_CNT: 1
COLUMN_NAMES: last_name
1 row in set (0.03 sec)
```

Now I need a way to find the set of indexes in all_indexes, but not in used_indexes. These indexes (if our original index statistics are good) are candidates to be dropped:

Shell

```
create view droppable_indexes as
select
 all_indexes.table_schema as table_schema,
 all_indexes.table_name as table_name,
 all_indexes.index_name as index_name
from
 all_indexes left join used_indexes using (TABLE_SCHEMA, TABLE_NAME, INDEX_NAME)
where
 used_indexes.INDEX_NAME is NULL and
 all_indexes.INDEX_NAME != 'PRIMARY' and
 all_indexes.NON_UNIQUE = 1;
```

Note that we also want to avoid dropping PRIMARY and UNIQUE indexes since those tend to enforce important application data constraints, so we added some additional criteria to the end of our SELECT.

I can now select my droppable (unused) indexes from this view:

Shell

```
mysql> select * from droppable_indexes;
+--------------+--------------+----------------------------+
| table_schema | table_name | index_name |
+--------------+--------------+----------------------------+
| sakila | actor | idx_actor_last_name |
| sakila | address | idx_fk_city_id |
| sakila | city | idx_fk_country_id |
| sakila | customer | idx_fk_address_id |
| sakila | customer | idx_fk_store_id |
| sakila | customer | idx_last_name |
| sakila | film | idx_fk_language_id |
| sakila | film | idx_fk_original_language_id |
| sakila | film | idx_title |
| sakila | film_actor | idx_fk_film_id |
| sakila | film_category | fk_film_category_category |
| sakila | film_text | idx_title_description |
| sakila | inventory | idx_fk_film_id |
| sakila | inventory | idx_store_id_film_id |
| sakila | payment | fk_payment_rental |
| sakila | payment | idx_fk_customer_id |
| sakila | payment | idx_fk_staff_id |
| sakila | rental | idx_fk_customer_id |
| sakila | rental | idx_fk_inventory_id |
| sakila | rental | idx_fk_staff_id |
| sakila | staff | idx_fk_address_id |
| sakila | staff | idx_fk_store_id |
| sakila | store | idx_fk_address_id |
+--------------+--------------+----------------------------+
23 rows in set (0.02 sec)
```

From here I can use some clever SQL to generate the precise ALTER TABLE statements to drop these indexes, an exercise left to the reader.

Estimating the size of these indexes

But, what if we want to see if it's worth doing first? Do these indexes actually represent a significant enough amount of disk space for it to be worth our while?

We need some more information to answer this question, but fortunately in Percona Server, we have it in the INFORMATION_SCHEMA.INNODB_INDEX_STATS table and the 'index_total_pages' column. A page in Innodb is (usually) 16k, so some simple math here should help us know how much disk space an index utilizes.

Let's go update our all_indexes view to include this information:

Shell

```
mysql> drop view if exists all_indexes;
mysql> create view all_indexes as
select
 t.table_schema as TABLE_SCHEMA,
 t.table_name as TABLE_NAME,
 i.index_name as INDEX_NAME,
 i.NON_UNIQUE as NON_UNIQUE,
 count(*) as COLUMN_CNT,
 group_concat( i.column_name order by SEQ_IN_INDEX ASC SEPARATOR ',') as COLUMN_NAMES,
 s.index_total_pages as index_total_pages,
 (s.index_total_pages * 16384 ) as index_total_size
from
 information_schema.tables t join information_schema.statistics i using (table_schema, table_name)
 join information_schema.innodb_index_stats s using (table_schema, table_name, index_name)
where
 t.table_schema like 'sakila%'
group by
 t.table_schema, t.table_name, i.index_name;
```

Now we can can see index sizing information in the all_indexes view:

Shell

```
mysql> select * from all_indexes\G
...
*************************** 33. row ***************************
TABLE_SCHEMA: sakila
TABLE_NAME: rental
INDEX_NAME: rental_date
NON_UNIQUE: 0
COLUMN_CNT: 3
COLUMN_NAMES: rental_date,inventory_id,customer_id
index_total_pages: 27
index_total_size: 442368
...
```

Now we just need to update our droppable_indexes view to use that information:

Shell

```
mysql> drop view if exists droppable_indexes;
mysql> create view droppable_indexes as
select
 all_indexes.table_schema as table_schema,
 all_indexes.table_name as table_name,
 all_indexes.index_name as index_name,
 ROUND(all_indexes.index_total_size / ( 1024 * 1024 ), 2) as index_size_mb
from
 all_indexes left join used_indexes using (TABLE_SCHEMA, TABLE_NAME, INDEX_NAME)
where
 used_indexes.INDEX_NAME is NULL and
 all_indexes.INDEX_NAME != 'PRIMARY' and
 all_indexes.NON_UNIQUE = 1
order by index_size_mb desc;
```

Now we can easily see how big each index is if we dropped it (not big in this case with test data):

Shell

```
mysql> select * from droppable_indexes;
+--------------+--------------+----------------------------+---------------+
| table_schema | table_name | index_name | index_size_mb |
+--------------+--------------+----------------------------+---------------+
| sakila | payment | fk_payment_rental | 0.27 |
| sakila | rental | idx_fk_customer_id | 0.27 |
| sakila | rental | idx_fk_inventory_id | 0.27 |
| sakila | rental | idx_fk_staff_id | 0.19 |
| sakila | payment | idx_fk_staff_id | 0.17 |
| sakila | payment | idx_fk_customer_id | 0.17 |
| sakila | inventory | idx_store_id_film_id | 0.11 |
| sakila | inventory | idx_fk_film_id | 0.08 |
| sakila | film_actor | idx_fk_film_id | 0.08 |
| sakila | film | idx_title | 0.05 |
| sakila | film | idx_fk_original_language_id | 0.02 |
| sakila | city | idx_fk_country_id | 0.02 |
| sakila | film_category | fk_film_category_category | 0.02 |
| sakila | customer | idx_last_name | 0.02 |
| sakila | store | idx_fk_address_id | 0.02 |
| sakila | actor | idx_actor_last_name | 0.02 |
| sakila | customer | idx_fk_address_id | 0.02 |
| sakila | staff | idx_fk_address_id | 0.02 |
| sakila | film | idx_fk_language_id | 0.02 |
| sakila | address | idx_fk_city_id | 0.02 |
| sakila | customer | idx_fk_store_id | 0.02 |
| sakila | staff | idx_fk_store_id | 0.02 |
+--------------+--------------+----------------------------+---------------+
22 rows in set (0.02 sec)
```

Recovering filesystem space

Now astute innodb experts will realize that this isn't the end of the story when it comes to reclaiming disk space. You may have dropped the indexes, but the tablespaces on disk are still the same old size. If you use innodb_file_per_table, then you can rebuild the tablespace for your table by simply doing:

Shell

```
mysql> alter table mytable ENGINE=Innodb;
```

However, this blocks and on a large table can take quite some time. All the normal tricks and tips about doing a long blocking schema change without affecting your production environment apply here and is out of scope for this blog post.

Happy hunting for those unused indexes!