# Brag feature: Secondary Index & MySQL Procedure

## Summary

In the resource management application there are quite a few queries to find specific resources fits a certain criteria, so we sought to add secondary index to speed up those queries. During the process, we tried to build index on several attributes, and had some interesting findings: some index will speed up the query, some, surprisingly, will slow down the query. And we did a little deeper to figure out why, and we also tried to define MySQL procedure to populate test data in batch - overall it’s a good learning experience.

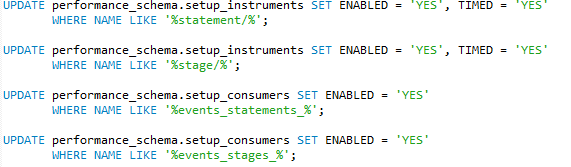
## Candidate attributes to index

Table: resource

Candidate attributes: since MySQL automatically creates index for primary keys and foreign keys, we picked other two columns used in the where predicate clause of queries: **status** (possible value 0 - available, 1 - in use, 2 - in repair), and **model** (varchar type, no limitation on possible values). The first one is used in several queries in the resource status page where we need to find out resources in different status. The second one is mainly used in the search for resource function.

## Experiment procedures with SQL code

1. Enable performance schema in MySQL



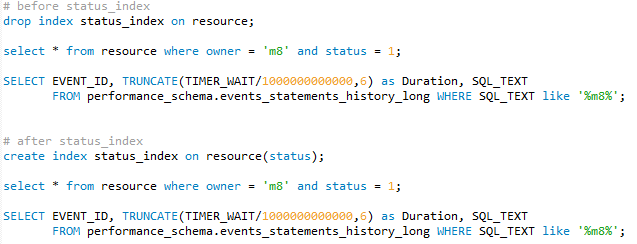
1. Disable query cache – an important step, so that the second query we run won’t gain advantage from the query cache the first query leaves behind



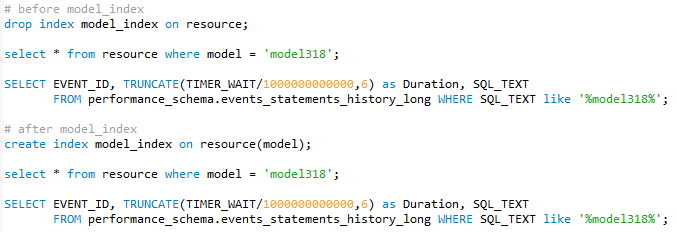
1. Populate test data – 2000 records were populated for resource table using the following procedure, with “status” column randomly filled with 0, 1, 2 and 500 distinctive values for “model” column, and all resources entries are evenly assigned to four types of users



1. Run the same query before and after creating index on “status” column, and find out query duration



1. Repeat step 3 for index on “model” column



## Results

The Duration is the average of 10 queries.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | Duration (sec) | SQL\_TEXT |
|  |  |  |
| Before status\_index | 0.000751 | select \* from resource where owner = 'm8' and status = 1 limit 2000 |
|  |  |  |
| After status\_index | 0.002894 | select \* from resource where owner = 'm8' and status = 1 limit 2000 |
|  |  |  |
| Before model\_index | 0.002785 | select \* from resource where model = 'model318' limit 2000 |
|  |  |  |
| After model\_index | 0.000482 | select \* from resource where model = 'model318' limit 2000 |

From the results above, we can see the secondary index on “model” speeds up the query as expected, however the secondary index on “status” actually slows down the query. This conclusion holds for all 10 tests. What happened?

## Explanation

We researched into InnoDB (the search engine used for MySQL) index process, and find the “status” is not a good candidate for index due to the nature of B-tree structure InnoDB is using.

Before using an index on “status”, to find resource entry with a certain “status”, InnoDB basically performs a sequential search of 2000 records to find match.

After creating the index, the index on “status” only has 3 possible values, each with ~700 entries. We now have fewer records (~700) to go through for each index value; however a sequential scanning of an index is less efficient than sequential scanning a table – the worst case possible case for a B-Tree. Plus all the overhead with index, it’s not surprising the index actually slows down the query significantly.

On the other hand, the “model” index has 500 possible values, each with ~4 entries, and by arranging them in a B-Tree, we can quickly find a match through divide and conquer approach. In other words, this B-Tree has a lot more leaves, with each leaf only with very few entries that requires a sequential search.

## Conclusion

Through the experiment, we learned that index doesn’t always speed up the query, and it will be more likely to help when the index column has enough distinctive values, and detrimental when very few values allowed in the attribute domain or several concentration of a few values in actual records.

What if it’s a case in between? Will a column with two dozen possible value good for indexing? I guess the only way to find out is to do a benchmark in advance before jumping into conclusions, and only create index whenever necessary. When doing that experiment – don’t forget to use SQL procedures.