CSE 291H Analytics: Techniques and Databases

# Project Phase Three Report

Yuning Hui – A53091905
Zhe Wang – A53097553

## Test Environment

Apple Macbook Air with MacOS 10.11.5 and Postgres 9.5.

## 1 Cats

### 1.1 Option "Overall Likes"

The query plan shows that Postgres did not make use of any indices, including the automatic indices on primary keys.

Execution time: 170.877 ms

### 1.2 Option "Friend Likes"

The query plan shows that Postgres did not make use of any created indices. However, it uses the automatic generated indices on the primary keys of tables *like_activity* and *friendship*.

Execution time: 0.238 ms

### 1.3 Option "Friends-of-Friends Likes"

The query plan shows that Postgres did not make use of any created indices. However, it uses the automatic generated indices on the primary keys of tables *like_activity* and *friendship*.

Execution time: 1.602 ms

### 1.4 Option "My kind of cats"

The query plan shows that Postgres uses the created index on column *video_id* in table *like_activity*. It replaces a sequential scan by a bitmap heap scan on *like_activity*.

Execution time before creating index: 680.296 ms
Execution time after creating index: 645.572 ms
Performance improvement: 5%

### 1.5 Option “My kind of cats – with preference

The situation here is basically the same with the last query “My kinds of cats”. The query plan shows that Postgres uses the created index on column *video_id* in table *like_activity*. It replaces a sequential scan by a bitmap heap scan on *like_activity*.

Execution time before creating index: 757.017 ms
Execution time after creating index: 737.549 ms
Performance improvement: 2.7%

As we can see, in Cats, the only useful index is *like_activity(video_id)*. But it only gives a 3% ~ 6% performance. For other index choices, either they were not adopted by the query plan or they only bring minor improvements, which can be ignored. For example, an index on *like_activity(user_id)* only improves about 0.02% of the performance in the query *friend of friend likes*. As a conclusion, we recommend no index on Cats.

## 2 Sales

### 2.1 Total Sales

The query plan makes a great use of index *sales(customer_id)*. In the original query plan without the index, it materializes the table after sorting the entire table *sales*. The hash index can replace it with two index scans.

Execution time before creating index: 15175.472 ms
Execution time after creating index: 1308.359 ms
Performance improvement: 91.4%

### 2.2 Total Sales for Each State

The query plan makes no use of any indices created, including the automatic created indices on primary keys.

Execution time: 6866.951 ms

### 2.3 Total Sales for Each Product for a Given Customer

The query makes a great use of index *sales(customer_id)*. It avoids the sequential scan on table *sales*. Instead, it uses a bitmap index scan.

Execution time before creating index: 487.220 ms
Execution time after creating index: 0.136 ms
Performance improvement: 99.97%

### 2.4 Total Sales for Each Product and Customer Order by Dollar Value

The b-tree index *sales(product_id, customer_id)* can save an external merge sort. However, it only saves about 5% overhead.

Execution time before creating index: 29747.477 ms
Execution time after creating index: 28278.425 ms
Performance improvement: 4.9%

### 2.5 Total Sales for Each Product Category and State

The query plan shows that it uses no index at all.

Execution time: 11673.393 ms

### 2.6 For each one of the top 20 product categories and top 20 customers, it returns a tuple (top product, top customer, quantity sold, dollar value)

The index *sales(customer_id)* can save a sequential scan on table *sales.*

Execution time before creating index: 15782.326 ms
Execution time after creating index: 13961.283 ms
Performance improvement: 11.5%

We have tried many index combination choices, the only index which brings significant improvements is *sales(customer_id)*.