CSE 291H Analytics: Techniques and Databases

Project Phase Four Report

Yuning Hui – A53091905
Zhe Wang – A53097553

Test Environment

Apple Macbook Air with Mac OS 10.11.5 and Postgres 9.5.

1 Cats

1.1 Precomputed Table Choices

The obvious precomputed table is to record the *lc* values of all pairs of users. For incremental update purpose, we maintain the 1+COUNT(*) values.

1.2 Precomputed Table Initialization

The direct join *like_activity* ⋈ *like_activity* is too slow because its huge intermediate results. The final results should be *users* * *users*, which is 5000*5000=25000000. We also measured the join result of one user's likes and the whole *like_activity* table. The time consumed improves a lot by calculating each user's *lc* values separately. Thus, we use FUNCTION in Postgres to iterate through all user_id, join them with the *like_activity* table, and perform count aggregations. It only took us about 6 minutes to compute the precomputed table *user_weights*.

1.3 Incremental View Maintenance

The IVM should combine the rules for updating joins and count/aggregate. After insertion, the join should be done by joining together the *delta plus* tuples and the *like_activity* table. The join should be performed twice for both operands of join. Because they are same, we just alter the order of columns of the *join delta plus.* Then, we just sum up all new counts and update the precomputed table *user_weights.*

In the same way, we compute the join and count aggregation results for each user. The update method for count remains the same with basic IVM rules. The *delta join* process only takes about 100s to finish.

The most time of our IVM update is spent on updating the precomputed table because it is too large. Even it benefits some from building indices on it, the time is still around 8 minutes.

1.4 Index Choices

To accelerate the join with the users and *like_activity*, we create an index on *like_activity(video_id)*. In the precomputed table *user_weights*, we often need to find the specific user *lx_user_id*, hence we create an index on *user_weights(lx_user_id).* Besides, when performing incremental updates, when need to update exactly a pair of (*lx_user_id, ly_user_id*). We also index it as a pair.

1.5 Test Results

1.5.1 Precomputation Time

With our separated computation for each user, the initialization time is 374s.

1.5.2 Maintenance Time

	Pure Insertion Time	IVM Overhead
10 tuples	18ms	1.1s
33000 tuples	1.1s	512s

1.5.3 Query Time

	Without Precomputed Table	With Precomputed Table
10 tuples	280ms	99ms
33000 tuples	327ms	165ms

1.6 Conclusion

In Cats schema, the cost of precomputation is relatively high. The initialization and IVM both take minutes to complete at large scale data. The query obviously benefits from the precomputed table. With the table, the computation for final query result only depends on the size of table *like_activity*. If the query is executed for multiple times, the the precomputation is really good for querying all users' like values.

2 Sales

2.1 Precomputed Table Choices

The query six of Sales schema first calculates two intermediate tables, which are top 20 categories and top 20 customers. Then it joins them together and perform an aggregation to get the final result. The original query execution time is 16-18s.

From the query plan, we can conclude that we should not precompute the join and aggregation in the last two steps. The reason is simple, we only have 20 tuples from each side of join, and aggregate only at most 400 tuples. It just not worth it.

Now, let's break down the computations under the two “top 20s”. To get the top 20 categories, we must first have $sales \bowtie product \bowtie category$. Then , we sum the *price_paid* by aggregation on *category_id*. From the *EXPLAIN ANALYZE* results, the join between *sales* and *product* takes 4.5s and the aggregation only consumes 100ms. The reason might be that there is only 100 categories, which makes the hash aggregation easier. Hence, we choose only to precompute $sales \bowtie product \bowtie category$. It is convincing to say that this choice can save considerable time while introduce reasonable additional overheads.

To get the top 20 customers, we only need to sum the *price_paid* by aggregation on *customer_id*. It takes 3.1s to finish. This simple computation takes too much time. Therefore we decide to precompute it. All joins afterwards only take less than 1s, which seems not too valuable to be precomputed comparing to the introduced overheads.

In conclusion, we only maintain two precomputed tables. One is $sales \bowtie product \bowtie category$, the other is sum *price_paid* by aggregation on *customer_id* in *sales*. The names of the two precomputed tables are *sales_prod_cate* and *sum_customer*. We believe they are beneficial while introducing acceptable overheads. The precomputed table creation queries are enclosed as *sales_precompute_tables.sql* and the adopted query is enclosed as *sales_new_query*.

2.2 Methodology Discussion

2.2.1 Trigger Functions

The trigger function is professional way to update precomputed tables automatically whenever there is an insert operation. First, we wrote two trigger functions for each precomputed table, then hook it with insert operations on *sales* by triggers. However in Postgres, triggers can only update one row each time. We have to perform IVM one row at each time, which brings a lot overheads. The trigger code is enclosed as *sales_triggers.sql*.

2.2.2 Simulated IVM

Another way to achieve IVM is executing IVM queries manually. First, the new tuples were inserted into another table *sales_inserted*. Then, we execute the IVM update queries manually and measure the time consumed. The simulated IVM query is enclosed as *IVM_maintenance_query.sql*.

2.2.3 IVM Method Choice

We implemented both methods and tested them on different scales of insertions. The result is listed below.

	Trigger Functions	Simulated IVM
10 Insertions	1.8s	< 1s
1000 Insertions	170s	< 1s
1000000 Insertions	> 40 mins	42.3s

Therefore, although trigger functions are really appropriate for IVM updates, its “FOR EACH ROW” execution brings significant overheads. We have to manually simulate the IVM updates.

2.3 Test Results

Using precomputed tables, we measured insertion time, query time of both original and IVM methods in small and large scales of insertions.

2.3.1 Precomputation Time

Here we present the time for initializing the two precomputed tables. There are four million tuples in *sales* table.

	$sales \bowtie product \bowtie category$	sum *price_paid* aggregated by *customer_id* in *sales*
Initialization time	6.7s	8.8s

2.3.2 Maintenance Overhead

Here we present the pure insertion time and IVM maintenance time for inserting different numbers of tuples into *sales*.

	Pure Insertion Time	IVM Maintenance Overhead
10 tuples	18ms	717ms
1000 tuples	100ms	1.9s
1000000 tuples	37.2s	7.3s
3000000 tuples	108s	10s

2.3.3 Query Time

We present the query time with and without precomputed tables after different sizes of insertions.

	Without Precomputed Table	With Precomputed Table
10 tuples	14.7s	4.3s
1000 tuples	14.7s	4.7s
1000000 tuples	18.5s	6s
3000000 tuples	33.2s	7s

2.3.4 Total Cost Comparison

The total cost of query with precomputed tables are maintenance overhead plus the query time. The cost of query without them is just the execution time itself. Here, we present both costs.

	Without Precomputed Table	With Precomputed Table
10 tuples	14.7s	5.0s
1000 tuples	14.7s	6.6s
1000000 tuples	18.5s	13.3s
3000000 tuples	33.2s	17.3s

2.4 Conclusion

We can divide the scale of the inserted tuples into three groups, which is small, medium, and large. In our Sales schema, the small scale means about 1000 tuples. The large scale means millions of inserted tuples. The medium scale is right between them. We can discuss the effects from IVM by these scales.

For small scale of insertions, the pure insertion time and IVM maintenance time can be ignored because they are too small compared with query times. The query with precomputed tables performs much better than the original one. It wins by one order of magnitude.

For large scale of insertions, the overall performance can be improved by a factor of three with precomputed tables. The reason is that although the query benefits from precomputed tables a lot, the cost of IVM maintenance makes the insertion time slower. The time IVM saves is primarily the re-computations of all intermediate results in the original query.

For medium scale of insertions, the performance is about the same. This is because the benefits from IVM cannot compensate the overheads. However, if we perform multiple times of queries, the query with precomputed tables can beat the original one easily.

One last thing is that we have tried to create an index on *sum_customer(customer_id)*. But the query won’t utilize it and it generates additional overheads when inserting. Hence we did not create any indices on Sales.