SalesCube: Milestone III

*Let me begin by saying I had significant challenges trying to clear the cache to get “cold runs.” My process consisted of disconnecting from the data-base, potentially reloading some tables, querying large other tables, etc. Considering that this is production technology and optimizing storage of cache for future utility helps to improve the product as a whole, I found that it was very hard to use alternative measures. Regardless, I believe I was able to gain an understanding of the material that was taught and apply it in this homework.*

Query 1:

Without Index: Sort (cost=860.29..876.93 rows=6656 width=83) (actual time=41.149..41.540 rows=1000 loops=1)

**CREATE INDEX** sales\_cust\_id\_index **ON** sale(customer\_id);

Query did not leverage index, subsequently the query performance did not increase. **Final Recommendation:** do not implement an index.

Query 2:

Without Index: Sort (cost=598.11..600.76 rows=1060 width=83) (actual time=35.236..35.263 rows=50 loops=1)

This query appears to be more optimal than the previous one, which is surprising since there is an additional join operation. Let us test the customer index again:

**CREATE INDEX** sales\_cust\_id\_index **ON** sale(customer\_id);

Query did not leverage index, subsequently the query performance did not increase. Considering we are joining on states as well, consider implementing a state index.

**CREATE INDEX** cust\_state\_id\_index **ON** customer(state\_id);

Query did not leverage index, subsequently the query performance did not increase. The fact that we were using the state foreign key did not improve performance. **Final Recommendation:** do not implement an index.

Query 3:

Without index: Sort (cost=189.65..189.65 rows=1 width=40) (actual time=4.637..4.638 rows=1 loops=1)

This query seems to be very optimal already, with very quick processing time and low cost. Potentially, there might be a way to implement an index but it is doubtful to improve performance.

**CREATE INDEX** sales\_cust\_id\_index **ON** sale(customer\_id);

Sort (cost=68.95..68.95 rows=1 width=40) (actual time=0.292..0.292 rows=1 loops=1)

The sales index on the customer\_id foreign key had a very large impact on both cost and processing time. While I believe there was likely some caching influence on this (and I did my best to mitigate it), the performance increase in processing time was very large. This was the best query output across both databases. **Final Recommendation:** implement index immediately.

Query 4:

Without index: Sort (cost=1534.09..1559.09 rows=10000 width=130) (actual time=56.346..56.841 rows=1000 loops=1)

**CREATE INDEX** sales\_cust\_id\_index **ON** sale(customer\_id);

Query did not leverage index, subsequently the query performance did not increase. Considering we are joining on product as well, consider implementing a product index.

**CREATE INDEX** sales\_prod\_id\_index **ON** sale(product\_id);

Query did not leverage index, subsequently the query performance did not increase. **Final Recommendation:** do not implement an index.

Query 5:

Without index: GroupAggregate (cost=1243.24..1593.24 rows=10000 width=130) (actual time=64.649..78.703 rows=735 loops=1)

**CREATE INDEX** sales\_cust\_id\_index **ON** sale(customer\_id);

Query did not leverage index, subsequently the query performance did not increase. Considering we are joining on product as well, consider implementing a product index.

**CREATE INDEX** sales\_prod\_id\_index **ON** sale(product\_id);

Query did not leverage index, subsequently the query performance did not increase. **Final Recommendation:** do not implement an index.

Query 6:

GroupAggregate (cost=1057.50..1060.50 rows=100 width=44) (actual time=62.832..63.065 rows=18 loops=1)

**CREATE INDEX** sales\_cust\_id\_index **ON** sale(customer\_id);

Query did not leverage index, subsequently the query performance did not increase. Considering we are joining on product as well, consider implementing a product index.

**CREATE INDEX** sales\_prod\_id\_index **ON** sale(product\_id);

GroupAggregate (cost=1011.22..1014.22 rows=100 width=44) (actual time=71.495..71.777 rows=18 loops=1)

Query did leverage the index but did not improve the performance. Increased the processing time while marginally reducing the cost. Hard to imagine the performance would increase with higher volume data sets. **Final Recommendation:** do not implement an index.