Milestone IV: Precomputation & Indexing

**Sales (used previous schema)**

*Execution Time Comparison:*

* Original Summary: Sort (cost=224931.38..224932.38 rows=400 width=48) (actual time=6403.066..6403.217 rows=400 loops=1) - ***full plan available in Appendix***
* Precomputed Summary: Seq Scan on q6\_final (cost=0.00..18.00 rows=800 width=72) (actual time=0.096..0.622 rows=400 loops=1) - ***full plan available in Appendix***

The precomputed query runs much faster than the original query. This speaks to the importance of a scalable data model, potentially a snow-flake model where computations are stored in dimensions. The cost to build the precomputed tables was not that high, so if providing a summary on the top customers and categories is something that only needs to be done once, the original query should suffice. However, if this is an indicator that is valuable to ongoing business success, then the precomputed tables should be leveraged.

The particular precomputed tables were chosen based on the problem requirements, it made sense to store the data for both top 20 categories and customers, and the table used to store sales by product and customer. Hence, views were created based on the 3 criteria necessary to measure top categories and customer trends, which ended up being drawn from a materialized view. The materialized view, while expensive to generate/maintain, allows for substantial precomputation (no longer a sequence of aggregating functions with full table searches) to allow for very quick information retrieval (see performance increase above). Considering this was a smaller table, no indices were added to the materialized view since performance would not increase from both a cost and duration standpoint.

**Cats (used previous schema)**

*Execution Time Comparison:*

* Original Summary: Limit (cost=8597.39..8597.42 rows=10 width=12) (actual time=228.579..228.595 rows=10 loops=1) - ***full plan available in Appendix***
* Precomputed Summary: Limit (cost=12.60..348.16 rows=10 width=12) (actual time=12.105..12.387 rows=10 loops=1) - ***full plan available in Appendix***

The precomputed query runs much faster than the original query. This speaks to the importance of a scalable data model, potentially a snow-flake model where computations are stored in dimensions. However, in the case with the Cats precomputed tables, the cost to build them was quite steep – this was largely due to needing every combination of interaction likes between users. Potentially, there might be considerations to just use the original query, depending on frequency of utilization of the recommendation engine. In the long run, with lots of recommendations necessary, using the precomputed approach makes the most sense.

While this problem posed particular challenges, considering that nested queries did very well with sequential filtering, I was able to create precomputed tables that increased query performance. However, the tables themselves were very expensive to create, that is because of the requirement on being able to produce a recommendation for any given user (prepared statement). Because of this requirement, a materialized view needed to be created to store individual recommendations – in which we had a ton of users. The materialized view was quite large so a performance increase of implementing an index on the user provided a lift in computational cost and time. If necessary, the best way to update the materialized view would be use of trigger functions, noting the large cost of creation. Prior to building the materialized view, the particular precomputed tables in views were chosen based on the problem requirements: I had to create views that would develop the appropriate inner product relationship between users and then log-weight that interaction. The materialized view, while expensive to generate/maintain, allows for substantial precomputation (no longer a sequence of nested queries and all aggregating functions have already been performed – no full table scans) to allow for very quick information retrieval (see performance increase above). Depending on how many times I expect to query these recommendations, I may consider altering my approach.

Appendix

**Sales Execution Plans:**

Original Plan:

Sort (cost=224931.38..224932.38 rows=400 width=48) (actual time=6403.066..6403.217 rows=400 loops=1)

Sort Key: t20cat.category\_id, t20cus.customer\_id

Sort Method: quicksort Memory: 44kB

CTE t20\_categories

-> Limit (cost=518.19..518.24 rows=20 width=40) (actual time=34.974..35.002 rows=20 loops=1)

-> Sort (cost=518.19..520.04 rows=740 width=40) (actual time=34.972..34.984 rows=20 loops=1)

Sort Key: (sum((s\_1.price \* (s\_1.quantity)::numeric)))

Sort Method: quicksort Memory: 27kB

-> HashAggregate (cost=489.25..498.50 rows=740 width=40) (actual time=34.851..34.869 rows=32 loops=1)

Group Key: ca.category\_id

-> Hash Join (cost=62.75..364.25 rows=10000 width=40) (actual time=3.370..23.681 rows=10000 loops=1)

Hash Cond: (s\_1.product\_id = p\_1.product\_id)

-> Seq Scan on sale s\_1 (cost=0.00..164.00 rows=10000 width=40) (actual time=0.025..6.337 rows=10000 loops=1)

-> Hash (cost=53.75..53.75 rows=720 width=8) (actual time=3.321..3.321 rows=1000 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 40kB

-> Hash Join (cost=26.65..53.75 rows=720 width=8) (actual time=0.145..2.425 rows=1000 loops=1)

Hash Cond: (p\_1.category\_id = ca.category\_id)

-> Seq Scan on product p\_1 (cost=0.00..17.20 rows=720 width=8) (actual time=0.032..0.843 rows=1000 loops=1)

-> Hash (cost=17.40..17.40 rows=740 width=4) (actual time=0.074..0.074 rows=33 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 2kB

-> Seq Scan on category ca (cost=0.00..17.40 rows=740 width=4) (actual time=0.021..0.043 rows=33 loops=1)

CTE t20\_customers

-> Limit (cost=296.82..296.87 rows=20 width=40) (actual time=15.219..15.242 rows=20 loops=1)

-> Sort (cost=296.82..297.32 rows=200 width=40) (actual time=15.217..15.224 rows=20 loops=1)

Sort Key: (sum((s\_2.price \* (s\_2.quantity)::numeric)))

Sort Method: top-N heapsort Memory: 26kB

-> HashAggregate (cost=289.00..291.50 rows=200 width=40) (actual time=13.991..14.589 rows=1000 loops=1)

Group Key: s\_2.customer\_id

-> Seq Scan on sale s\_2 (cost=0.00..164.00 rows=10000 width=40) (actual time=0.009..4.413 rows=10000 loops=1)

-> HashAggregate (cost=224092.97..224098.97 rows=400 width=48) (actual time=6402.392..6402.723 rows=400 loops=1)

Group Key: t20cat.category\_id, t20cus.customer\_id

-> Hash Right Join (cost=175348.97..224020.97 rows=7200 width=48) (actual time=5508.433..6391.217 rows=13640 loops=1)

Hash Cond: ((c.customer\_id = t20cus.customer\_id) AND (p.category\_id = t20cat.category\_id))

-> Sort (cost=175334.37..177134.37 rows=720000 width=95) (actual time=5420.229..5826.211 rows=1000000 loops=1)

Sort Key: c.customer\_id, (COALESCE(sum(((s.quantity)::numeric \* s.price)), 0.0))

Sort Method: quicksort Memory: 109639kB

-> GroupAggregate (cost=79911.88..105286.88 rows=720000 width=95) (actual time=1970.655..4549.322 rows=1000000 loops=1)

Group Key: c.customer\_id, p.product\_id

-> Merge Left Join (cost=79911.88..85486.88 rows=720000 width=95) (actual time=1970.626..3325.792 rows=1009000 loops=1)

Merge Cond: ((c.customer\_id = s.customer\_id) AND (p.product\_id = s.product\_id))

-> Sort (cost=79083.49..80883.49 rows=720000 width=59) (actual time=1957.773..2371.915 rows=1000000 loops=1)

Sort Key: c.customer\_id, p.product\_id

Sort Method: quicksort Memory: 102702kB

-> Nested Loop (cost=0.00..9036.00 rows=720000 width=59) (actual time=0.048..1284.378 rows=1000000 loops=1)

-> Seq Scan on customer c (cost=0.00..17.00 rows=1000 width=51) (actual time=0.023..1.167 rows=1000 loops=1)

-> Materialize (cost=0.00..20.80 rows=720 width=8) (actual time=0.001..0.419 rows=1000 loops=1000)

-> Seq Scan on product p (cost=0.00..17.20 rows=720 width=8) (actual time=0.010..0.514 rows=1000 loops=1)

-> Sort (cost=828.39..853.39 rows=10000 width=44) (actual time=12.843..20.013 rows=10000 loops=1)

Sort Key: s.customer\_id, s.product\_id

Sort Method: quicksort Memory: 1166kB

-> Seq Scan on sale s (cost=0.00..164.00 rows=10000 width=44) (actual time=0.021..4.775 rows=10000 loops=1)

-> Hash (cost=8.60..8.60 rows=400 width=8) (actual time=51.089..51.089 rows=400 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 16kB

-> Nested Loop (cost=0.00..8.60 rows=400 width=8) (actual time=50.208..50.831 rows=400 loops=1)

-> CTE Scan on t20\_categories t20cat (cost=0.00..0.40 rows=20 width=4) (actual time=34.980..35.025 rows=20 loops=1)

-> CTE Scan on t20\_customers t20cus (cost=0.00..0.40 rows=20 width=4) (actual time=0.762..0.773 rows=20 loops=20)

Planning time: 2.436 ms

Execution time: 6424.199 ms

Precomputed Plan:

Seq Scan on q6\_final (cost=0.00..18.00 rows=800 width=72) (actual time=0.096..0.622 rows=400 loops=1)

Planning time: 0.317 ms

Execution time: 1.012 ms

**Cats Execution Plans:**

Original Plan:

Limit (cost=8597.39..8597.42 rows=10 width=12) (actual time=228.579..228.595 rows=10 loops=1)

Buffers: shared hit=1304

CTE log\_cats

-> HashAggregate (cost=4658.48..4661.96 rows=199 width=8) (actual time=55.449..55.480 rows=41 loops=1)

Group Key: l1.user\_id, l2.user\_id

Buffers: shared hit=638

-> Hash Join (cost=947.12..4192.07 rows=62188 width=8) (actual time=5.443..55.284 rows=41 loops=1)

Hash Cond: (l2.video\_id = l1.video\_id)

Buffers: shared hit=638

-> Seq Scan on likes l2 (cost=0.00..944.00 rows=49750 width=8) (actual time=0.018..26.061 rows=49990 loops=1)

Filter: (user\_id <> 1)

Rows Removed by Filter: 10

Buffers: shared hit=319

-> Hash (cost=944.00..944.00 rows=250 width=8) (actual time=5.363..5.363 rows=10 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 1kB

Buffers: shared hit=319

-> Seq Scan on likes l1 (cost=0.00..944.00 rows=250 width=8) (actual time=0.017..5.343 rows=10 loops=1)

Filter: (user\_id = 1)

Rows Removed by Filter: 49990

Buffers: shared hit=319

-> Sort (cost=3935.43..3935.93 rows=200 width=12) (actual time=228.577..228.580 rows=10 loops=1)

Sort Key: (sum(log\_cats.weight)), l3.video\_id

Sort Method: top-N heapsort Memory: 25kB

Buffers: shared hit=1304

-> HashAggregate (cost=3929.11..3931.11 rows=200 width=12) (actual time=228.020..228.250 rows=366 loops=1)

Group Key: l3.video\_id

Buffers: shared hit=1298

-> Hash Join (cost=1606.35..3866.92 rows=12438 width=12) (actual time=60.277..227.090 rows=369 loops=1)

Hash Cond: (l3.user\_id = log\_cats.o\_user\_id)

Buffers: shared hit=1298

-> Hash Anti Join (cost=1599.89..3689.20 rows=12500 width=8) (actual time=4.591..143.150 rows=49939 loops=1)

Hash Cond: (l3.video\_id = w.video\_id)

Buffers: shared hit=660

-> Hash Anti Join (cost=947.12..2581.75 rows=25000 width=8) (actual time=4.465..89.470 rows=49949 loops=1)

Hash Cond: (l3.video\_id = ll.video\_id)

Buffers: shared hit=638

-> Seq Scan on likes l3 (cost=0.00..819.00 rows=50000 width=8) (actual time=0.008..29.096 rows=50000 loops=1)

Buffers: shared hit=319

-> Hash (cost=944.00..944.00 rows=250 width=4) (actual time=4.431..4.431 rows=10 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 1kB

Buffers: shared hit=319

-> Seq Scan on likes ll (cost=0.00..944.00 rows=250 width=4) (actual time=0.017..4.419 rows=10 loops=1)

Filter: (user\_id = 1)

Rows Removed by Filter: 49990

Buffers: shared hit=319

-> Hash (cost=646.51..646.51 rows=500 width=4) (actual time=0.104..0.104 rows=20 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 1kB

Buffers: shared hit=22

-> Bitmap Heap Scan on watch w (cost=12.17..646.51 rows=500 width=4) (actual time=0.043..0.094 rows=20 loops=1)

Recheck Cond: (user\_id = 1)

Heap Blocks: exact=20

Buffers: shared hit=22

-> Bitmap Index Scan on watch\_u\_id\_index (cost=0.00..12.04 rows=500 width=0) (actual time=0.028..0.028 rows=20 loops=1)

Index Cond: (user\_id = 1)

Buffers: shared hit=2

-> Hash (cost=3.98..3.98 rows=199 width=12) (actual time=55.561..55.561 rows=41 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 2kB

Buffers: shared hit=638

-> CTE Scan on log\_cats (cost=0.00..3.98 rows=199 width=12) (actual time=55.454..55.523 rows=41 loops=1)

Buffers: shared hit=638

Execution time: 229.012 ms

Precomputed Plan:

Limit (cost=12.60..348.16 rows=10 width=12) (actual time=12.105..12.387 rows=10 loops=1)

Buffers: shared hit=345

-> Nested Loop Anti Join (cost=12.60..75112.43 rows=2238 width=12) (actual time=12.096..12.369 rows=10 loops=1)

Join Filter: (w.video\_id = p.video\_id)

Rows Removed by Join Filter: 200

Buffers: shared hit=345

-> Nested Loop Anti Join (cost=0.43..52076.34 rows=4477 width=12) (actual time=11.715..11.826 rows=10 loops=1)

Join Filter: (l.video\_id = p.video\_id)

Rows Removed by Join Filter: 145

Buffers: shared hit=323

-> Index Scan using precompute\_u\_id\_index on precompute\_likes p (cost=0.43..24749.12 rows=8954 width=12) (actual time=0.263..0.279 rows=20 loops=1)

Index Cond: (user\_id = 1)

Buffers: shared hit=4

-> Materialize (cost=0.00..945.25 rows=250 width=4) (actual time=0.005..0.571 rows=8 loops=20)

Buffers: shared hit=319

-> Seq Scan on likes l (cost=0.00..944.00 rows=250 width=4) (actual time=0.066..11.303 rows=10 loops=1)

Filter: (user\_id = 1)

Rows Removed by Filter: 49990

Buffers: shared hit=319

-> Materialize (cost=12.17..649.01 rows=500 width=4) (actual time=0.009..0.042 rows=20 loops=10)

Buffers: shared hit=22

-> Bitmap Heap Scan on watch w (cost=12.17..646.51 rows=500 width=4) (actual time=0.073..0.316 rows=20 loops=1)

Recheck Cond: (user\_id = 1)

Heap Blocks: exact=20

Buffers: shared hit=22

-> Bitmap Index Scan on watch\_u\_id\_index (cost=0.00..12.04 rows=500 width=0) (actual time=0.051..0.051 rows=20 loops=1)

Index Cond: (user\_id = 1)

Buffers: shared hit=2

Execution time: 12.627 ms