Steven Carpenter Take Home Answers

Hello Underdog team! Here are my answers to the questions from the take home project. I am also including a file called ddl.sql with the table DDLs and queries to generate the analytics tables. At the top of that file are instructions to run the queries with data. I have also included in the zip the csv files I loaded.

Approach:
The provided tables join nicely together but work best as two fact tables, one at the selection level and another at the projection level because projections can exist without any selections. Surely no one is betting on the Raiders to win the superbowl… but there is the option. These tables are selection_analytics and projection_analytics. Initially I had smashed it all into one table, but I found this to be easier to query and cleaner. With full outer joins to accommodate the projections with no entry info, it resulted in not having a non null PK. With the two table solution, they are both indexed on the projection_id for very fast joins when the query necessitates this. In several cases though it did not.

This makes querying very fast and with appropriate indexing will also scale quite a bit. To build this locally I just spun up a docker postgres container, created these tables ddl's, asked chat-gpt to generate me some test data (it did an okay job surprisingly) so I could test my queries and loaded that in my db. I then created the two analytics fact tables where at most I would need one join on indexed columns and I could preserve the primary key sanctity for selections and projections.

Query Performance:

- Explain how you would optimize performance of these datasets, if you were creating them for downstream stakeholders who would be querying them often.
    - Key and Index according to most frequently used columns for filtering to maximize predicate pushdown. Especially if there is a high IO cost associated with retrieving data over a network from an object store like S3
    - Partition the data based on the above constraints as well if this is a distributed system.
    - For a data warehouse like redshift, make use of the dist all to place small dimensions across all worker nodes for fast joins.
    - Analyze column statistics to choose the best encoding and compression in a data warehouse.
    - For object stores, optimize file size first for ~512mb-2gb file sizes and use a splittable file format and compression like snappy parquet. There are deeper parquet configurations for slice sizes that can also help optimize how much of the file is read based on the explain plan.

- How might your techniques change if you were trying to accommodate these stakeholder groups?
    - Both column representations for these dates should be indexed. You could denormalize this for the consumer by using stakeholder specific views of the fact tables. In my example with one flattened table, adding date column's derived from the respective timestamps these stakeholders use and indexing on those (less cardinality than a timestamp) would greatly increase the query performance for both groups.
    - Finance - tends to analyze data by the date the entries were created
    - Operations - tends to analyze data by the date in which the game occurred

Scalability Analysis:
Discuss how the chosen approach would handle a 1000x increase in data volume. Address potential challenges and how they would be mitigated.

- From the jump I am using string uuid's for id columns, anticipating issues using even a bigint given the potential number of selections per entry, per person, etc. This avoids issues like rolling over ints at the expense of storage space.
- I could further break out dimensions based on unique values assigned to int ids and then join to those tables from the fact tables to reduce the size of the two fact tables a bit by excluding the game data unless queried for.
- I indexed date columns in each table but for a massive increase, partitioned tables would make a lot of sense in a data lake table.

Data Refresh Strategy:
Taking into account data volume and possible resource constraints, design and explain your data refresh strategy. Please include discussion of the following elements:
I am going to make a few assumptions based on the data in this project specifically

1. There are no updated_at fields here so unless the entry in entries that has a created_at value can be overwritten or updated I have to assume this dataset is immutable.
2. There must be some mechanism to update the games table though in case the game_start_time changes
3. I am assuming that we want to preserve the raw data and for this purpose (and most purposes these days) I would be replicating the database into S3 as Iceburg or DeltaLake tables and building based on a lakehouse model.

Replication via change data capture is the ideal solution in my opinion. It is lightweight and offers flexibility. All of the raw tables from the database are replicated based on the immutable change log they expose into S3 as Iceburg tables. There are several vendor and open source solutions for this such as debezium. At fanatics, our sister team built tooling to do this in house while also fanning out that data to StarRocks. With the CDC data sunk to S3, you can choose the cadence you want to apply changes at separately. You can get very close to real-time here by using Spark Streaming or Flink. In some cases it can cause way too many small files if changes are applied liberally and the tables would require compaction.

- Incremental vs full load
    - Full Load consideration
        - A full load should only be chosen for a regular cadence if there is value in preserving a temporal state of the table. Some tools like Snowflake, DeltaLake, and Iceburg support time travel natively to accomplish this without needing to duplicate the table. Naively, this can be done at expense by partitioning snapshots of the data to a partition based on the time of the snapshot. I have done this in the past with epoch timestamp and then utilized S3 lifecycle policies to move old snapshots to cheap glacier storage and then ultimately expire the objects to save cost.
        - If it is required to read the database via jdbc, this should be done with a read replica for this purpose. You can have this spin up ephemerally as part of the workflow to unload tables.
            - I have written Spark jobs to unload tables like this before and it is a simple way to get a full snapshot of the table at a point in time, but there can be dangling relationships between the tables due to timing of the snapshot. This is possible with other methods too, but utilizing change data capture can minimize it.
        - There should be job that can do a full reload on demand regardless in case of emergency
        - Any full load process should build the table to a staging schema and/or table and then once validation is complete the tables can be swapped atomically in a commit block with whatever engine is being used to manage the table metadata (Hive, Glue, PG, MySQL, etc)
        - The metastore should point to the latest snapshot location in S3. For a data warehouse the prod table would be the latest table and you could preserve the most recent snapshot in an old schema if that is useful.
    - Incremental Load considerations
        - This is almost always the desired way to load but there are some considerations.
        - New data is easily appended. If all data for a source is always new, it can always be appended in a new partition based on the schedule without reconciling with the existing dataset or any of its partitions(considerations on that below).
        - In most situations there will be updated and deleted rows as well that need to be accounted for.
        - Applying changes
            - If possible, use an upsert. This may not be performant enough even with effective indexing for the operation.
                - Load files optimally based on the datastore eg. If a redshift cluster has 6 nodes, and each node has 2 cores, then 12 files is usually a good partitioning to maximize ingest because each core handles a thread concurrently.

        - Requires reading the existing table in some capacity but the goal should be to minimize the necessary data to load even though it may be impossible to avoid reading the whole table.
        - Some tidying up may be necessary
            - Compacting small files
            - Vacuuming tables
            - Re-bucketing
        - Define criteria to determine whether data in the incremental batch or the existing data should take precedence.
            - Could use a time based field if most recent is what counts.
            - Could always prefer data in the new incremental batch.
            - Could be some complex business logic, it just needs to be applied when reconciling.
        -
- Refresh cadence (e.g. hourly, daily, etc.)
    - This choice is entirely down to the business need. That need can be, “this would be really cool if it was sub millisecond real-time,” so long as the right stakeholders are thoroughly reviewing the design and sign off on the expected costs.
    - The schedule can impact how you partition data in S3. If a dataset is small abd partitioned hourly this can lead to the small file problem fast!
        - You can still update the dataset hourly and partition daily or monthly if that makes sense for the dataset!
    - Loading hourly or daily won’t necessarily yield cost savings and depends entirely on the data itself and the expense to load it. 24 hourly loads may cost the same as 1 daily load and then you have data hourly which is better! It could be that loading daily is much more expensive due to increased complexity in reconciling the incremental change at a daily scale (time and cost per row updated may not scale linearly).
    - Run time can play a part here as well. When thinking about the job’s SLA, it is important to have ample time for retries in case of transitive failures. Ideally I like my average runtime to be ¼ of the schedule interval. This allows for three retries without causing the next scheduled run to trample it (depending on the job that may be fine or I would utilize resource limits to limit the number of concurrent runs).
    - Downstream consumer scheduling is another factor here. Often tables could be updated once daily at UTC midnight, but that might leave the dashboard that sales managers check first thing in the morning out of date. I know from experience that it is worth at least scheduling another incremental run before they get online :).