Introduction

This document serves as a guide for my work sample for a position with AWS as a Sr. Data Architect, Big Data. I'll begin with my **considerations** about the problem space and thoughts about the different directions one could go in search of a solution. Following that will be a verbal description of the **general approach** I've decided to implement here. After that, we'll dig into the **technical details** of the implementation, the **opportunities for improvement**, and finally, a **summary** to tie everything together.

Considerations

Our client (Big Wind, LLC.) desires to query high resolution weather data for use in its products. Specifically, it desires to query data from the UKMET office mogreps-g dataset that is produced every 6 hours. Key requirements are that data must be available (in CAP theorem terms, we are willing to sacrifice some level of Consistency and Partition-tolerance in order to guarantee maximum Availability), queries must be fast, and new data must be available quickly.

My initial thought was that this is a problem well-suited for a distributed document datastore like DynamoDB. The biggest danger of split-brain scenarios in a distributed system are dirty reads and stale reads that happen because a transaction hasn't propagated to all nodes in the system. When you consider that there is no user updating data (i.e., data is only ever modified by the ETL process we are creating) then you can mostly dispense with the P in CAP theorem for this scenario. This again leads me down the path of a DynamoDB solution or something similar. Perhaps the best solution is a distributed database with characteristics that are excellent for time series like Cassandra or good for search indexing like ElasticSearch.

On the other hand, the initial specifications were to process a ~26MB file and make it queriable in a timely fashion while upholding the needs for robustness of process and resilience of data.

So my thought process went like this:

If we store the data as is in some form of document, time series, or search index, we're using a lot of space rather quickly—just to stay current. The client is in the transportation and planning industry, and as much as we hope this doesn't occur, accidents happen, and lawsuits often follow. It would be a benefit to at least keep some amount of archival data.
Looking closer at the n-dimensional structure of the data, it's obvious that there's an enormous amount of duplication. Every "cube" is self-contained and repeats all the data necessary to analyze the cube without reference to any other underlying data.

Putting all of that together leads me to go against my initial inclination. You have multi-dimensional data with lots of duplication, a need for high availability and fast queries, a need for a scalable ETL process to provide new data quickly, and a need to keep costs down. I've inferred the need to keep costs down because that is a fundamental tenet of building trust with your clients. If your first response is always the most expensive one, that's a hard relationship to keep.

In keeping with the importance of building trust, I'm going to infer a new requirement: this needs to be not only *fast* to query, but it also needs to be *easy* to query.

Just put a pin in those pie-in-the-sky assumptions for a minute. Because after I implemented things based on these ideas, I discovered that everything else I'm about to demo in this doc was deeply and badly wrong.

The General Approach

The previous chain of reasoning leads me to a relational solution in the style of a data warehouse. It's not a strict, Kimball-style fact and dimension structure because this isn't really a strict data warehouse. But the Kimball design principles can be effective here.

Let's get the design differences out of the way first. Most obviously, a weather "data warehouse" doesn't model a business process. Unless you are willing to call "querying weather data" a business process (and there are some academic papers that do this). But that's a misplaced desire to conform to design principles. If you were modeling that process, you'd warehouse a bunch of query logs, not create the queries themselves.

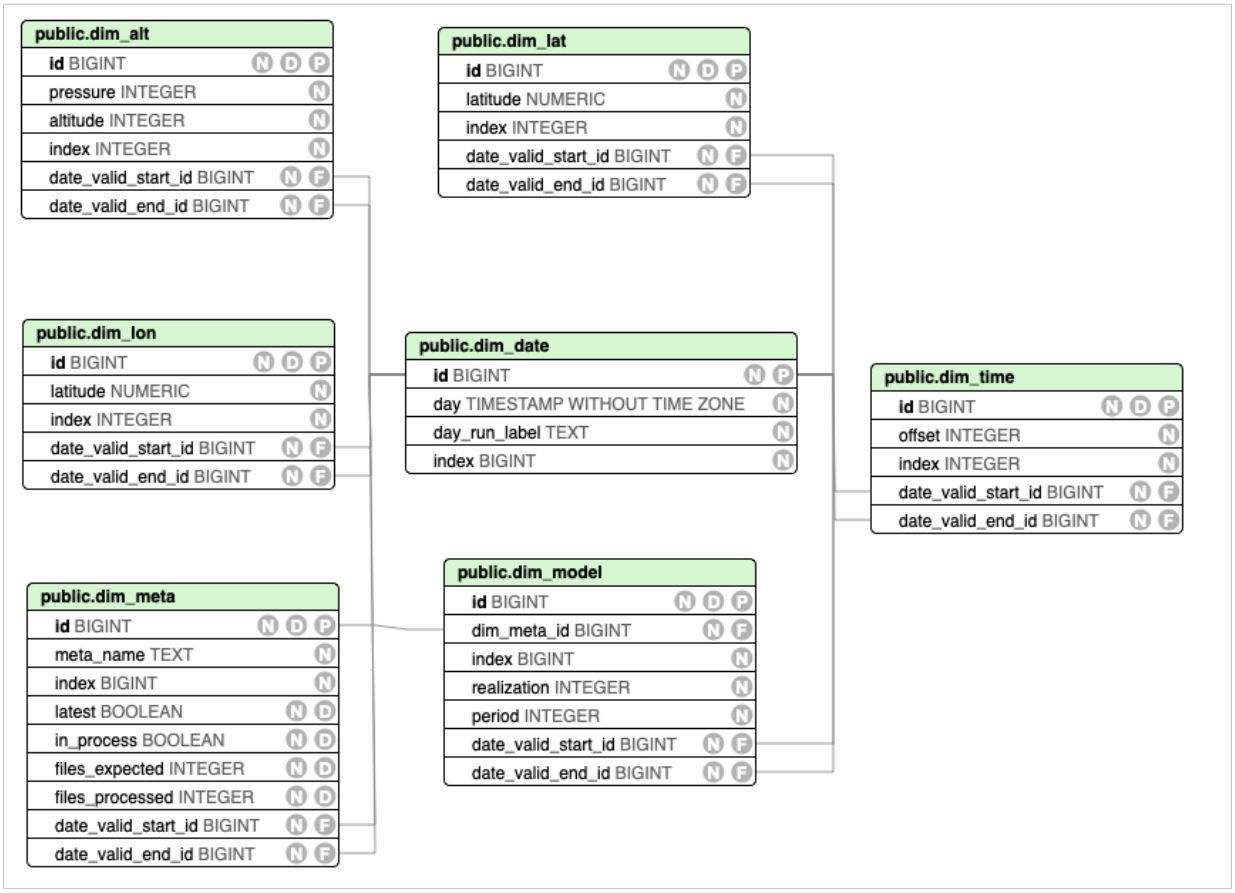
Second, in a strict Kimball model, dimensions cannot/should not be joined directly. You'll end up with a cartesian product, and it will be awful. In this model, I allow for some dimensions to join directly to others via foreign key (so that we avoid that cartesian product issue) in service of easier maintenance of the ETL process overall, an easy-ish way to manage slowly changing dimensions, and the ability to generate some top-line status summaries gracefully.

A good example of this (and to be fair Kimball doesn't totally hate this) is storing valid date ranges in the rows of your dimensions. I have dim_lat and dim_lon in this model. What if the UKMET office changes the resolution of this model at some point in the future? At one point in the past, I think it was 50km grids. It's currently 33km grids. What if they move to 15km grids? By setting start and end dates on the rows in those dimensions, we can adapt to model changes and still query historical events simply by changing the date we are looking for.

What if we want to incorporate *other* predictive models? This is all handled easily with a few practical modifications to the strict data warehouse structure.

Technical Details

One thing we *can* do that fits with normal data warehouse techniques is define the *grain* of the data we are going to capture. We want the maximum level of resolution here, so we will define the grain as recorded value by model by realization by period. I diagram is a good thing here to reduce how many words I'm writing:



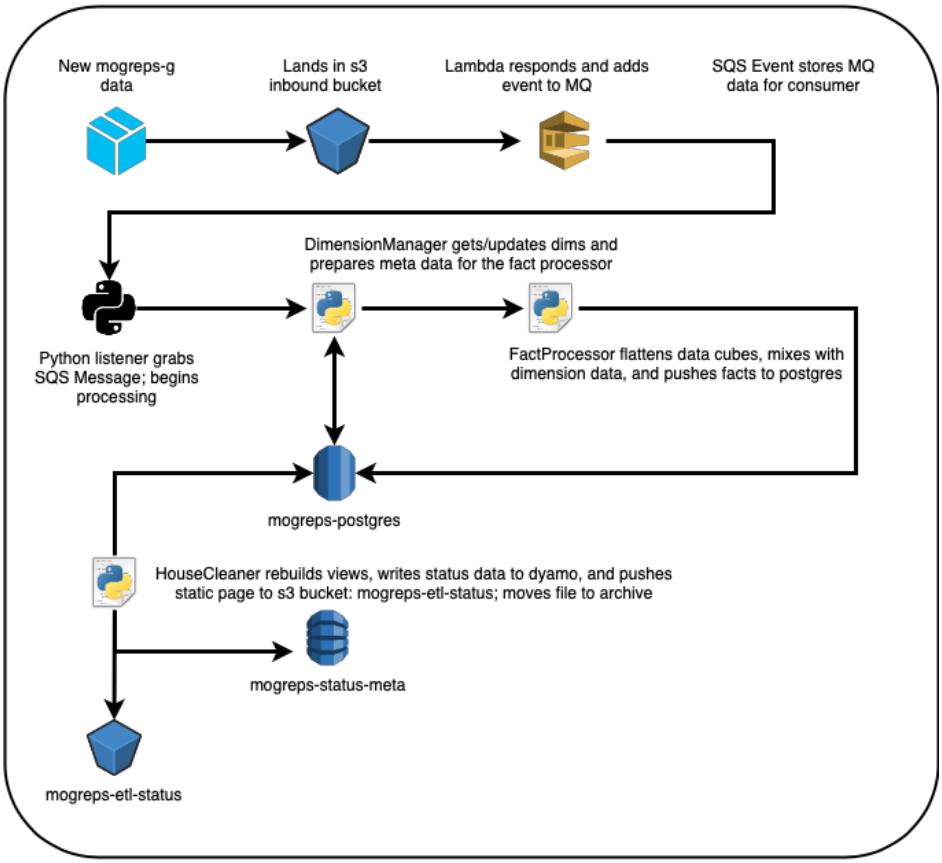
Most of these are very obvious.

dim_model is by far the most atypical dimension that I've defined. It contains the realization and period information, but it also has a foreign key to the meta dimension because we need to know without joining to a fact, how many of the model files have been processed, and how many are expected, etc.

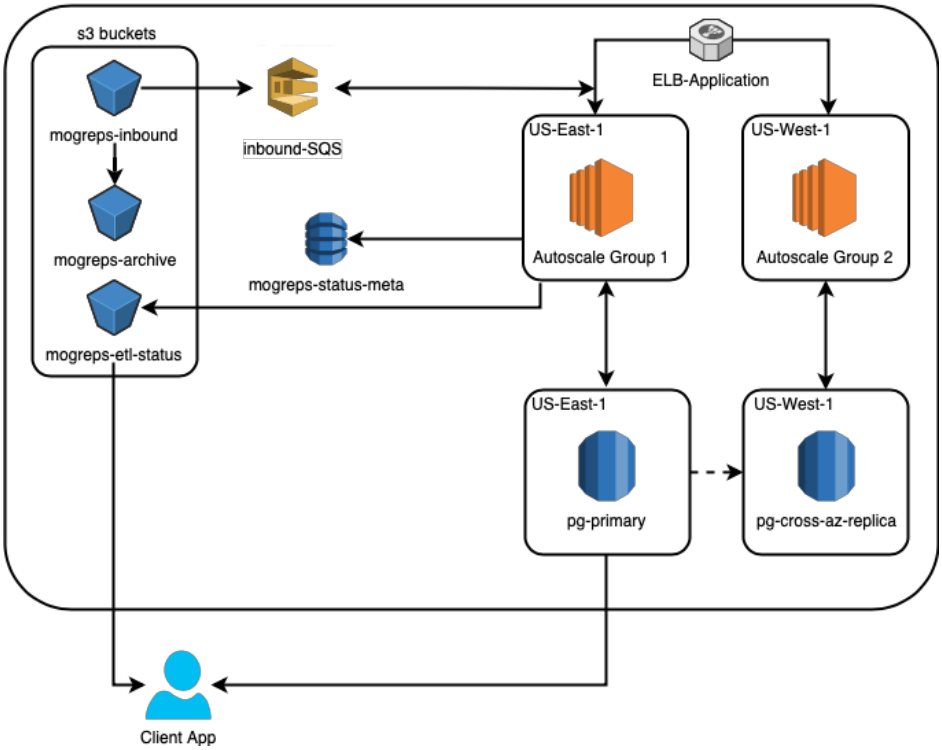
And speaking of meta dimension, it has all the things you want to know about the current status of your data. Is it the latest available? Is it complete? How complete is it? How far back does it go?

This is low-level stuff. How about the big picture? Let's go back to a diagram again.

MOGREPS-G ETL Data Flow



MOGREPS-G ETL Infra



This all boils down to a very straightforward ETL process supported by a scalable, available, and resilient infrastructure. I'm using core AWS services where I can to not invent my own wheels, and it sort of makes sense. Even though I haven't implemented all of it yet.

Strengths and Weaknesses

The strength of this approach is that the code works at a very high level, it could be easily handed off to a client with even barely minimal skills for maintenance and support. It's really simple ETL. I like the code organization and structure, and it's not hard to understand. It's also *completely useless*. It's too slow by 3 orders of magnitude. Here's why:

While this work sample talks about only one ~26MB file arriving every few minutes, the mogreps model as a complete entity is larger by several orders of magnitude. Like most forecasting models, this is initialized every 6 hours. It has a control run plus 21 perturbations of initial conditions, and it produces a file for each of those for every 3 hours up to 7 days. The UKMET office also recommends that the best forecast results come from a time-lagged combination of the last two model runs. The total dataset consists of (22*56*92*2)MB for a total of ~226GB of data that needs to be queried, with half that arriving every 6 hours. And those files are packed extremely densely inside numpy arrays inside a data wrapper called iris. And we need to process that within a few minutes, if I understand the requirements correctly.

This structure I've described generally in this document completely fails the requirement of making the data available fast. I keep a rack of Dell servers in my office that I can use to test my assumptions about scalability. Some of them do other things, but one of them is usually available to throw big jobs at. And since I work a lot in Python, I know just exactly how much an autoscale group is going to speed things up vs. my dev laptop. Not much. Not nearly enough, for this use case.

I could waste enormous amounts of your time explaining all the ways that you can optimize raw Python for this kind of job, you can strip away abstraction layers, engage c and Fortran hooks, etc. Or, I could just admit that this is *fundamentally* the wrong approach. Unrolling n-dimensional cubes with tight loops is fundamentally the wrong way to do this. What we need to do is map items in those arrays to low-cost objects and store those. Let's talk about all the things I haven't had a chance to implement yet: PySpark, EMR, and some other miscellany.

Opportunities for Improvement

As I said above, rolling out data cubes via tight loops is bad. It's not just bad for Python, it's bad everywhere. What we want is to treat each array as a hash table (which we can with Spark), and map hash locations to low-cost objects (in this case Pandas arrays), and push them to a datastore.

When I started this work sample, I wanted to eschew ec2 altogether. I wanted to use the ecosystem to its fullest and rely on messaging from one service to another. Well, getting loopy on things didn't help that, and I had to abandon Lambda because I couldn't even get a single fact table processed inside the 5-minute limit.

I'm really getting close to out of my time here for my two-week work sample, so I've had to make a lot of compromises. I would like to make another diagram. But I'm bad at those, and they take me a long time, so here it is in text:

File → s3 Bucket → Lambda(partition manager)→EMR→Postgres→Status

The basic idea here is that instead of the s3 event sending a message to SQS as it does now, and the ec2 downloading and processing the file . . . s3 pushes an event to a Lambda that breaks up the datacubes from the file and pushes them to EMR for parallelized processing—most importantly here, mapping and reducing—not just a faster loop. And then collating the results and pushing them to postgres.

What I will say about this that is clearly up for a good debate is whether or not RDS is the right choice at all. My initial opinion was that it was the right choice for this type of data. I think it needs relationships. But there are also possibly strong arguments that says, "Hey, this is time series data! By definition! Postgres is a bad idea from the start, and Cassandra is where you should've started."

That's a real possibility at the hundreds-of-GB-per-5-minutes type of big data scale. And almost everything has better write characteristics than postgres, if it's given that you are relaxed about certain guarantees.

But again, I would go back to the argument that this needs to be easy to query for the client, so. I guess I'm of two minds here.

Summary

I wish I could go back in time and undiscover what I now know about weather data. That would make this document a lot easier to write, and make me a lot happier about my code. I've provided the most basic implementation possible, and it fails miserably on time constraints. But I hope I've also provided some kind of meaningful information about what it means to work with me day-to-day on a project that involves truly large data: I have a lot of opinions, and I'm willing to stick with them until I know they are really, really bad ideas. And then I hate them.

-andrew