Cloud Spend Data Warehouse Deficiencies and Proposed Architectural Solution

White Paper

Author: Mary Welsh – Data Engineer

March 26, 2025

[Overview 1](#_Toc204349123)

[Summary Recommendation 2](#_Toc204349124)

[Current Data Architecture Areas for Improvement 2](#_Toc204349125)

[Challenges with Historical Data 3](#_Toc204349126)

[Structural Fragility 3](#_Toc204349127)

[Scalability 4](#_Toc204349128)

[Performance Issues 5](#_Toc204349129)

[Future Users 6](#_Toc204349130)

[Architectural/Strategic Proposal 7](#_Toc204349131)

[Challenges to Address 7](#_Toc204349132)

[Conclusion 9](#_Toc204349133)

[Summary of References 10](#_Toc204349134)

## Overview

The current Cloud Spend data warehouse architecture faces some significant challenges. We've already encountered issues that have led to a system that's become complex and difficult to scale. This complexity also makes it challenging to fully understand and implement even fundamental data warehouse features. This paper aims to highlight these areas and propose an architectural strategy grounded in industry best practices. The goal is to establish a more robust foundation for future development, enhance performance, and streamline our processes. Addressing these architectural concerns is crucial for us to move forward effectively.

## Summary Recommendation

Based on my research and over a decade of experience in the data domain, with a particular emphasis and training in data warehouse architecture, I recommend adopting a traditional star schema dimensional modeling approach, leveraging the Kimball methodology. This approach can be used to account for around 80-90% of business use cases. The other ~10% can be addressed with a complementary OBT (One Big Table) architecture for specific use cases and/or relevant data marts built on top of the underlying star schema for reporting/dashboard purposes. (<https://redhacherif.substack.com/p/demystifying-data-modeling-which>)

Key Takeaways

* Current Cloud Spend data warehouse architecture lacks many key features and best practices that will allow for ease of use for both developers and end users, improve performance, increase scalability, enable a robust foundation for future feature development, and more.
* The Kimball star schema approach can solve most of the issues we are currently facing and more.
* The OBT (One Big Table) approach and/or use of data marts can be implemented on top of the star schema to account for the rest of the possible business use cases that are not solved with the star schema alone.

## Current Data Architecture Areas for Improvement

The current data architecture has several areas where fundamental data warehouse design components and best practices haven't been adequately implemented or are missing altogether. Some of these include:

* Lack of unique/surrogate keys in tables to keep track of individual rows
* No reliable “updated\_at” columns to track CDC (change data capture)
* Insufficient implementation of SCD (slowly changing dimension) functionality to track historical data
* Unclear boundaries of data architecture pattern phases of modeling e.g. staging > cleansing > core > data marts (see Medallion Architecture: <https://www.databricks.com/glossary/medallion-architecture>)
* Complex business logic throughout the entire data pipeline
* Table structures are not optimized for any specific architectural approach causing performance issues
* Current architecture does not easily allow for outside user (analyst/data scientist/AI/ML) analysis

## Challenges with Historical Data

The ability to provide historical data has been a challenge to implement due to the lack of SCDs (Slowly Changing Dimensions) and several other key components that are required for such a feature. Historical data is a foundational feature of data warehouse architecture, and in a traditional dimensional modeling approach is implemented using SCDs. The way these work is by tracking changes to each row in the dimensional attributes and, in a typical type 2 SCD implementation, add a row for each new change while also tracking the start and end dates of the rows for when that data was valid. See more about how SCD type 2 works here: <https://en.wikipedia.org/wiki/Slowly_changing_dimension#Type_2:_add_new_row>

The current data warehouse architecture is missing several components to allow for such a feature.

The absence of unique/surrogate keys will prevent the ability to uniquely identify rows to keep track of what has potentially changed for each row.

The lack of a reliable “updated\_at” column means the only way the system can determine if a change was made to a particular row is to check each individual attribute for each row to detect changes, which is extremely taxing and basically non-performant. With such an “updated\_at” column, it will note the date that the record was last updated so the system can just look at this column and only update records that have been “updated\_at” more recently than the last date. (My current solution going forward is to use a metadata field in the lower level models. This field can potentially serve as a substitute for the updated\_at column, given the way the team currently ingests data is not the typical overwriting method.)

Dbt, which is the tool our team uses, already has extensive support for SCD functionality. Although dbt calls them “snapshots”, the logic and process work the same for standard SCD type 2. See the dbt documentation here: <https://docs.getdbt.com/docs/build/snapshots>

Another important note is that without well-architected proper dimension tables, SCD will not be as performant as it otherwise would be when implementing best practices.

Furthermore, a key benefit of historical data implemented and architected in this way is the ability to see how data changes over time to analyze trends, enable forecasting, and gain greater visibility into how changes can affect the data over time. Additionally, it can also provide valuable insights into the overall value of the data warehouse product with user adoption.

## Structural Fragility

The team's dbt codebase has become increasingly cumbersome to manage. This is understandable, given the inherent complexity of the business logic, which varies significantly across different offerings and even evolves within those offerings over time. We've also observed issues arising from the frequent use of pre- and post-hooks in dbt. Because these hooks aren't visible in the compiled dbt code, their impact can sometimes go unnoticed, potentially leading to unexpected behavior in other parts of the codebase. This makes debugging a more complex task, and fully grasping the nuances of the business logic becomes quite difficult without comprehensive documentation, which is still under development.

A screenshot of a computer

AI-generated content may be incorrect.

The above image displays a small portion of our current dbt flow of table dependencies. It's worth noting the sheer volume of tables, most of which couldn't even be captured in this screenshot, and this number continues to grow with each new offering or feature. Comprehending the intricacies of this data flow is becoming increasingly challenging, and it presents a clear indication that this approach may not be sustainable in the long run.

A traditional star schema can greatly simplify the code base by consolidating the business logic to a single modeling layer. Whether that logic layer happens before or after creation of the star schemas has yet to be determined as more research is needed in this area. Once the business logic can be sorted out and consolidated, the traditional modeling approach will offer all the benefits of stability, data integrity, historical data capabilities, greater user accessibility, scalability, and much more.

## Scalability

Related to the issues in the section above, due to the sheer complexities of our current architecture and code, adding to the codebase can be an intimidating task to ensure there are no unseen dependencies that may be impacted by new changes. It has been recommended to implement an audit to ensure numbers align between deploys. While this is a wise move for the current setup, having a well-architected data warehouse will lessen the necessity for such a feature as the confidence in the codebase will increase and put developers’ minds at ease when making changes.

Not only this, but having a well-architected data warehouse ensures future features can be implemented with greater ease and speed and can provide a much more robust structure for all users involved – developers, analysts, end users, etc.

## Performance Issues

Drawing from my experience as a DBA, I've observed several performance-related issues within our data warehouse. These appear to stem from the current table structures and the complexity of our underlying logic.

While Redshift is a columnar database and can be very performant with large volumes of rows of data, performance declines sharply with the addition of too many columns in tables. This is one of the many reasons OBT is advised against for columnar databases (see <https://www.thoughtspot.com/data-trends/data-modeling/data-modeling-best-practices-for-analytics-and-data-engineers>) Many of our tables have above 100+ columns which can greatly degrade query performance and additional table/model creation. Furthermore, it appears that there needs to be greater emphasis and attention to the proper use of sort and dist keys.

Similarly, our dbt code seems to be getting pushed to its limits due to the intricate business logic and numerous layers of modeling phases. Our team has frequently found itself needing to retroactively implement some of dbt's fundamental features, such as incremental loading, to achieve performance improvements.

Fortunately, I have noted that there are many ways in which both of our main tools – Redshift and dbt – are specifically designed and optimized to handle star schema architecture.

Redshift is a columnar oriented RDBMS which works extremely well with the overall structures of fact and dimension tables. Fact tables, having many more rows than columns are naturally very performant in a columnar database. Alternatively, dimension tables, having many columns but fewer rows and potentially many repeated values can take advantage of Redshift’s compression algorithms as well as not being too row-intensive to take a toll on the performance. Redshift also has several articles to aid in performance tuning for star schemas. “While Amazon Redshift automatically detects star schema data structures and has built-in optimizations for efficiently querying this data, you can further optimize your data model to improve query performance.” (<https://aws.amazon.com/blogs/big-data/optimizing-for-star-schemas-and-interleaved-sorting-on-amazon-redshift>) (<https://aws-samples.github.io/aws-dbs-refarch-edw/src/star-schema>)

Dbt also has many built in features related to relational data modeling. We have already touched on a few above in the Historical Data Capabilities section such as SCDs (or as dbt calls them “snapshots’). Dbt also has pre-coded libraries and shortcuts to help with relational/star schema architectural setup such as a surrogate key generator (<https://github.com/dbt-labs/dbt-utils/blob/main/macros/sql/generate_surrogate_key.sql>), a macro to automatically generate the entire date dimension (<https://github.com/calogica/dbt-date/blob/main/macros/get_date_dimension.sql>), as well as several guides for how to implement this architecture from start to finish using dbt (<https://docs.getdbt.com/blog/kimball-dimensional-model>)

Clearly, both of the primary tools we are using – Redshift and dbt – are already primed and ready to support a dimensional star schema architecture and it is apparent that they are frequently used in this way.

## Future Users

One potential feature that has not yet been discussed in depth is the possibility of exposing more of our data warehouse data to users such as analysts, data scientists, and AI/ML applications. We are currently exposing a version of some of our data through an API for users to consume but this data is more limited in scope, and few are able to access it. Typically, data warehouses can be used to provide insights to a much broader audience. As it is now, our data is very limited in how it is being consumed. The potential to open this data up for other users could increase visibility into important trends, enable more creative and innovative analysis, and elevate the value of the entire platform.

The implementation of a broader data exposure can be done in many ways. One of the most common methods is to expose the data using data marts designed for specific users to give them a more focused view on the data relevant to their use cases. These data marts are often just views on the underlying data warehouse data to protect the underlying data from any changes that may accidentally occur on the view.

Another approach is the recently popular OBT (One Big Table) schema. This method just joins all data together into one giant denormalized table. This approach has been lauded as most useful for applications of AI/ML ingestion and can be beneficial for simple queries. Problems arise, however, with scalability, loss of data granularity, data integrity and maintenance, and performance. (See <https://redhacherif.substack.com/p/demystifying-data-modeling-which>) Because of these issues, it is recommended that this approach be used only in very specific circumstances.

## Architectural/Strategic Proposal

Due to the issues we are facing listed above, I have started on a proposal to architect our current data in a way that will utilize best practices and hopefully greatly improve our processes by simplifying logic, optimizing performance, facilitating the creation of future features, and increasing robustness and scalability.

From the recommendations in the documentation from The Data Warehouse Toolkit (Ralph Kimball, Margy Ross) among several other sources, we would have a resulting star schema consisting of 2 facts: spend and usage. These are the measurements we care about aggregating and analyzing.

The dimensions are all the attributes about each of the record in both fact tables. The date dimension (dim\_date) is a data warehouse fundamental table which includes all date-related attributes we may need to view the data by. The dim\_spend and dim\_usage tables are both junk dimensions providing additional metadata about the individual spend and usage records in the fact tables respectively. We would then have dimensions for more of the attributes for where the spend is coming from, which will be hashed out with more of the business logic.

We also have options for a few different types of fact tables here. Fact\_spend can be a traditional transactional fact table or an accumulating snapshot table to track more changes in how the spend is allocated. We could even explore supporting both. More research is needed to discover the best approach going forward.

Further explanation of a traditional transactional vs accumulating snapshot fact table can be found here: <https://www.holistics.io/blog/the-three-types-of-fact-tables>

## Challenges to Address

The most difficult challenge of implementing this new architecture will be addressing the incredibly complex business logic for each of our offerings. There are several options to explore.

* Load the fact\_usage table as a typical transactional table. Place business logic into separate macros before loading the final fact\_spend table. The final fact\_spend table will need to base the calculations on the fact\_usage data, hence this dependency.
  + This moves the business logic code into macros where the logic can be consolidated, simplified, and reused as necessary
  + If possible, transfer to using weighted spend calculations using percentages based on usage.
* Keep the fact tables transactional and move business logic to the data mart layer. Can potentially have a data mart per offering, but if going that route would need a data mart to union all the data marts together as needed for reporting.

A diagram of a business process

AI-generated content may be incorrect.

More research is required to pinpoint a solution that would best fit our business needs while also optimizing for performance, simplicity, and other such factors.

Regardless of the approach we choose, it is vital to consolidate and simplify the business logic to a single layer in our processes to reduce overall complexity and improve scalability. The overall flow should follow a similar pattern as is outlined in the Medallion Architecture here: (<https://www.databricks.com/glossary/medallion-architecture>)

Going forward, I recommend a phased approach to implementing these changes. We have already started looking into converting our current metadata “dimension” tables into SCDs and we should aim to consolidate these tables as much as possible (grouping by similar business logic/functions).

Once these dimensions are properly architected and implemented as SCDs, we can then start creating the proper fact tables as well as any remaining junk-type/auxiliary dimensions. Again, more investigation will need to be done as to how to implement these fact tables accounting for the complex business logic.

After both dimension and fact tables have been completed, we will then need to address the question of what the future of the product should look like and what goals it needs to accomplish. These factors will ultimately influence the architecture we should use. For example, if we want the possibility of exposing more data in the data warehouse for direct user/application consumption we could explore the data mart or OBT options. Both/either of which would then sit on top of the existing data warehouse facts/dims. We could even potentially use such data marts/OBTs as data sources for the QuickSight dashboards.

## Conclusion

After extensive research into architectural best practices and the options and tools available, I highly recommend using a star schema approach in our data warehouse as it has many benefits that will greatly outweigh the efforts to implement it. It is always wise to have a solid architectural plan in place before development, but unfortunately many teams are simply not afforded the opportunity to do so or are even aware of the best practices to begin with. As stated in the following blog, (<https://www.ssp.sh/blog/data-modeling-for-data-engineering-approaches-techniques>) “Data modeling is easy to neglect; assessing the consequences can take time and effort. The image below illustrates that if you initially ignore poor data modeling and architecture decisions, you’ll likely notice problems in the last mile, thinking they might be due to the tools or insights. However, the fundamental issues primarily originate in the first part of the data analytics cycle.”

A diagram of a person's problem

AI-generated content may be incorrect.

Additionally, the following article warns of the cost of accumulating similar technical debt over time: (<https://medium.com/booking-com-development/measuring-technical-debt-to-avoid-the-boiling-frog-syndrome-c44eb48b3ce1>)

Cartoon of cartoon of men wearing helmets talking to each other

AI-generated content may be incorrect.

It states, "The trick with technical debt is that it accumulates very slowly by taking many small shortcuts now and then." This gradual accumulation, where prioritizing short-term ease over long-term considerations leads to mounting issues, is described in the article as the boiling frog metaphor. The team is at a point where these issues are already adding up, and we want to proactively avoid a critical situation. It's clear that the current architectural setup presents several red flags that are already impacting the quality and usability of our product and could severely hinder it if not addressed promptly. Implementing sound architecture and best practices will not only help us avert potential crises but also enable us to build future features on a strong, reliable foundation, ensuring integrity, performance, scalability, and much more.

## Summary of References

* <https://redhacherif.substack.com/p/demystifying-data-modeling-which>
* <https://www.databricks.com/glossary/medallion-architecture>
* <https://en.wikipedia.org/wiki/Slowly_changing_dimension#Type_2:_add_new_row>
* <https://docs.getdbt.com/docs/build/snapshots>
* <https://www.thoughtspot.com/data-trends/data-modeling/data-modeling-best-practices-for-analytics-and-data-engineers>
* <https://aws.amazon.com/blogs/big-data/optimizing-for-star-schemas-and-interleaved-sorting-on-amazon-redshift>
* <https://aws-samples.github.io/aws-dbs-refarch-edw/src/star-schema>
* <https://github.com/dbt-labs/dbt-utils/blob/main/macros/sql/generate_surrogate_key.sql>
* <https://github.com/calogica/dbt-date/blob/main/macros/get_date_dimension.sql>
* <https://docs.getdbt.com/blog/kimball-dimensional-model>
* <https://redhacherif.substack.com/p/demystifying-data-modeling-which>
* <https://drawsql.app/teams/a-1435/diagrams/maverick>
* <https://www.holistics.io/blog/the-three-types-of-fact-tables>
* <https://www.databricks.com/glossary/medallion-architecture>
* <https://www.ssp.sh/blog/data-modeling-for-data-engineering-approaches-techniques>
* <https://medium.com/booking-com-development/measuring-technical-debt-to-avoid-the-boiling-frog-syndrome-c44eb48b3ce1>