Data Warehousing Critical Success Factors

Introduction

I've been fortunate to have been part of some successful data warehouse projects, and some not so-successful ones. As you build out your reporting solution plan, allow for time for the unexpected, testing, performance tuning, and the like. Recognize that the longer the project lasts, the more the business will move away from the original requirements, and the data warehouse project must keep up. In addition to all this, keep in mind factors that are critical to a successful implementation:

Scope

Manage scope aggressively. Implement only what is required. It's OK to design a extensible pattern, but leave it at that.

Listen to your users. Know what they want out of the system and why.

Delivering too much is fatal. Don't try to design ahead of what the users want. You will be wrong. Even your users will be wrong. Commit to the hard requirements for the here and now. Anything else is a gamble with no upside. At the same time, be ready to assimilate new hard requirements into the scope. Set expectations that new scope will increase timeframes and cost.

It's a tough dichotomy because at once, you must maintain tight scope so you can deliver. Yet, if you deliver something that's no longer needed, that's bad also. Use your experience and relationship with the users to navigate the terrain.

Design

Design the simplest possible solution that still works. Borrowing from Sun Tzu, the best reporting solution is the one that does not have to get implemented. Aside from that, keep it simple.

If the source application has adequate reporting capabilities, use them.

If you have to build a data-based reporting solution, beware of making the data flow overly complicated. The tendency is to normalize and rationalize vendors' data. Too often that leads to complicated transformations and column mappings that require extensive testing. There will a corner case that you did not anticipate. 3rd Normal Form does us an injustice by guilting us into "good" design that users do not care about. There's nothing wrong with denormalized data for the vast majority of use cases. If it takes a 100% of your brain to create the thing, you've got nothing left to figure things out when stuff goes wrong.

If you find it necessary to create a star schema data model, keep the objects intuitive and relevant. Make it so that the users can thumb through the data dictionary and recognize all the names (except for surrogate keys maybe).

Make the data model follow standard data warehouse design. Let fact tables be fact tables. Dimension tables be dimension tables. Create simple surrogate keys as primary keys on the dimensions. Refer to the them as foreign keys from the fact tables. Use simple naming conventions. Let there only be one degree of separation between dims and facts. Question whether or not slow change dimensions are even necessary. If queries are anchored in the fact tables, then dimensions can just be records. Unless there is a business requirement stating otherwise, no one cares if the database stores every version of a dim in history.

Forget the snowflake model. It is death. The leaf node dependencies force a limiter on concurrent processing. It makes for really complicated transformation rules. It may look cool hanging inside your cube, but it is death.

Think carefully about the ETL model - push or pull. In push, the data warehouse does not need to know about the source system. However, the individual data extractors may not have enough context to send a complete picture. In pull, the data warehouse has to know its source and know when data can be extracted. But, it can grab all the data that it needs.

Testing

Use a set of repeatable test cases. Permute the inputs so that corner cases can be checked. If you're reporting daily, create a way to inject the test cases into the data.

Agree on a method to reconcile the data. The method must be fast and automated. Think simple checksum.

Some users will insist on parallel testing, where the new system runs along side the old system for some period of time. This is fine as a supplemental test method. However, you and your team will grow old trying to go live if parallel testing is your only approach.

Performance

First, know how much data is involved. Next, know the capabilities of your infrastructure. Then use that knowledge as constraints. Minimize the total number of data hops. Take measurements of your system's throughput. Use that as a benchmark to size your system. Design for your constraints. Set realistic expectations with your users.

Don't assume that throughput or response times will be optimal. Count on the opposite.

Conclusion

Voice any concerns strongly and early. Don't remain quiet. If you're working with a vendor on a fixed price engagement and they won't accept any design changes because it will mess up their timeline, kick them out of the office. Whatever has been spent is a sunk cost. If the project has a bad smell, stop the whole thing and resolve the issues.