Common View & Join Design Issues

Granularity

top

When designing a view you will have to consider the level of granularity of the view.

It is quite common in parent-child transaction databases to encounter facts of differing granularity. On an order, for example, there may be a shipping charge that applies to the entire order that isn't available at the individual product-level line item in the operational system. The designer's first response should be to try to force all the facts down to the lowest level. We strive to flatten the parent-child relationship so that all the rows are at the child level, including facts that are captured operationally at the higher parent level, as illustrated below. This procedure is broadly referred to as allocating. Allocating the parent order facts to the child line-item level is critical if we want the ability to slice and dice and roll up all order facts by all dimensions, including product, which is a common requirement.

If the shipping charges and other header-level facts cannot be allocated successfully, then they must be presented in an aggregate table for the overall order. We clearly prefer the allocation approach, if possible, because the separate higher-level fact table has some inherent usability issues. Without allocations, we'd be unable to explore header facts by product because the product isn't identified in a header-grain fact table. If we are successful in allocating facts down to the lowest level, the problem goes away.

We shouldn't mix fact granularities (for example, order and order line facts) within a single fact table. Instead, we need to either allocate the higher-level facts to a more detailed level or create two separate fact tables to handle the differently grained facts. Allocation is the preferred approach. Optimally, a finance or business team (not the data warehouse team) spearheads the allocation effort.

Indexes

top

A common issue when designing a reporting view for an OLTP Database is that the indexes used for managing transactions are inappropriate for report writing.
Whereas an index for transactions may be on the primary key for reporting these may be on multiple dimensional attributes. This can create a conflict of requirements that will determine your overall data architecture strategy depending on the size and complexity of your underlying database. As a result you may have to extract data from your underlying tables into specific reporting tables in the form of a data mart or OLAP cube to ensure optimised reporting.

Outer joins

top

When creating a join with inner joins the "direction" of the join is not relevant, but when creating an outer join the direction does matter. A join rule in Yellowfin is that an inner join cannot reside on the discretionary end of an outer join.

For example this will work:
CASE_PARTIES inner join CASE STAGES outer join TEAMS

But this will not work:
TEAMS outer join CASE_PARTIES inner join CASE STAGES

If you require complex joins like this you may have to use a virtual table, a SQL view or a hard coded view on the database. Even though the view above would only require simple insertion of brackets, Yellowfin does not cater for complex nested joins such as:
A inner join B outer join (C inner join D) outer join E outer join (F inner join G outer join H)

top