LIQUENT InSight for Analytics

Reporting Schema Design Approach

Copyright

Contents

Reporting Schema Design Approach

Design Approach

The schema is at this point still a bit of a work-in-progress. This is because customers may want/need different things. The goal in the LIQUENT InSight 6.0 release was to make the data *available*. The ongoing goal post LIQUENT InSight 6.0 is to improve the *convenience* of accessing the data, where possible. We have made some strides in the convenience arena by reorganizing the data along the lines of a star schema. For more information on general star schema design and use, see http://en.wikipedia.org/wiki/Star_schema

At this time, the entire schema is implemented in views, rather than materialized. It is understood that views may be slower than materialization, but since materialization is a complex endeavor the focus and priority was placed on getting the schema as usefully designed as possible. The benefit of using views is that the data can always be assumed current.

No attempt has been made, at this point, to capture any sort of audit-like history of what each row looked like at any point in time. That is something that can be explored in the future, should the need arise. One exception is with PDS data, where a history of event-related changes is an integral part of the production data. The entire history of any PDS detail is available in the reporting schema.

An additional benefit to the view implementation is that smaller, less busy databases can benefit directly from the simplified schema without need for complex ETL procedures. In the future, if the need for ETL and materialization arises, the hope is that the view definitions can be converted to tables and maintained using ETL tools. That way, the same reports and query interface can be used regardless of whether the data is accessed via views or via materialized tables that look like the views.

As to the possible performance implications of using views, the performance seems to be better than what is seen today with previous generations of LIQUENT InSight reports, for the following reasons:

- There are no stored procedure calls used in the reporting schema. The superior capabilities of the reporting tool provide many ways of formatting the data on output, so it is not necessary to depend upon the database to provide pre-formatted data.
- The same queries are simpler in the reporting schema. Queries in general, are easier to construct. Simpler queries usually mean better performance.
- Out-of-the-box reports have been designed to be very focused in their reporting intent. With the reporting schema, it is easier to write efficient, optimized queries to answer specific questions. This eliminates the need for gigantic single queries that are meant to answer many different questions, and never answer any of them efficiently.

View Groups

The views are organized into groups, identifiable by the prefixes on the views.

Dimension Tables

There are a large number of dimension tables. Dimension tables have the prefix V**D**_<*description*>*.* A dimension refers either to a “thing” in LIQUENT InSight or to a relationship between “things” in LIQUENT InSight. The latter is important because it is not entirely conventional.

Examples of “thing” dimensions are “application” (**vd_application**), “registration” (**vd_registration**) and “event” (**vd_event**).

Examples of “relationship” dimensions are “event_country” (**vd_event_country**) and “registration-packageset-country” (**vd_reg_packset_country**).

Each dimension has a column that is the dimension key, which is usually named as “<dimension name>_id”. See “Naming Conventions” in this document for more information and some exceptions. Dimension tables contain all of the “information” about a “thing” or a “relationship”. Examples: the name of a “thing” such as an application, or the status of a “relationship”, such as an event_country.

Fact Tables

The conventional fact table philosophy is to reorganize, pre-join and pre-calculate business data in ways that the business wants to analyze the data. For example, if you wanted to obtain a simple list of all registered package sets and registered medical devices in LIQUENT InSight, you would have to navigate two different, complex paths through the data. The reporting schema now provides a harmonized view of all registered products (**vf_registered_product**). There are only a few of these, as they are supposed to be targeted towards very specific perspectives on the data. In time, we would expect to see these being refined a bit, perhaps some added or deleted or combined. These fact tables do the “heavy lifting” of pulling data together. They have the prefix **vf**_*<description>.*

Since LIQUENT InSight is marketed to many customers with many needs, it is clear that, at least in our first iteration of the reporting schema, we may not have every possible perspective on the data covered. Furthermore, there will be a permanent need for absolutely customized reports.

As a means to this end, a second set of fact tables are referred to as “context” fact tables are provided. Fact tables of this type have the prefix **vf_ctxt**_*<entity name>.* These are simply a way to create the dimension hierarchy for each LIQUENT InSight entity. These fact tables are simple, narrow tables that give the id numbers of the dimensions that comprise the context of entity within the LIQUENT InSight system. The granularity of this set of fact tables is 1:1 with the LIQUENT InSight entity for which the fact table is providing context.

For example, **vf_ctxt_event** will list the columns event_id, application_id and product_family_id, and there will be one row for every row in the LIQUENT InSight event table. This facilitates a report with a row corresponding to each event, with application and product family information (dimensions) added. The notion of having a fact table that is 1:1 with the dimension table seems at first a bit curious. This is because a tacit attempt is being made to build a reporting model that can accommodate possible reorganizations of the LIQUENT InSight data hierarchies in the future.

Join Methodology

Dimension-related information is pulled into the query by joining the fact table to the relevant dimension by the column on the fact table that has the same name as the dimension key.

Example –

Fact table **vf_ctxt_event**: application_id, product_family_id, event_id

Dimension tables: **vd_application, vd_product_family, vd_event,**

Where Clause:

where **vd_application**.application_id = **vf_ctxt_event**.application_id

and **vd_product_family**.product_family_id = **vf_ctxt_event**.product_family_id

and vd_event.event_id = vf_ctxt_event.event_id

The PDS Dimensions

PDS Dimensions warrant special attention. They are named **vd_pds_**<*detail type*>**_hist**. Each of the pds history tables is modeled as a separate dimension. Each pds table has a different set of attributes, and as such, is impossible to harmonize into any sort of single pds history table. The entire history for each detail is included in each table. The event is listed on the row as event_id. To get event information, join to the event dimension (**vd_event**). This allows one to report on the pds detail state at any point in time.

To get current approval information, go directly to these views and select the rows where row_currency = 'CA' to get current approved status. To get rows with pending changes, select rows with row_currency = 'C'.

An alternative approach to querying pds history is to use these views in conjunction with the fact table **vf_pds_detail_summary** and join in the last_approved/last change_id to the history dimensions. However, this approach is tricky as the joins should be done as separate queries for each detail type.

Multi-Valued Attributes

Some LIQUENT InSight entities have attributes that can have multiple values. Examples might be product routes of administration or family substances.

In previous LIQUENT InSight reporting schemes, these types of attributes have been presented as delimited lists. It was thought to be desirable to eliminate the csv concept, due to performance implications. Most of these attributes in LIQUENT InSight are simple relationships with no attributes on the relationships themselves.

For now, for the sake of simplicity, multi-valued attributes are stored in their own separate view with the naming convention **vb**_<*parent entity*>_<*attribute name*>.

Data Row Security

Data row security is implemented through two views, **vb_fam_security** and **vb_app_security**. These are simple views that are fed by the normal LIQUENT InSight data security logic. The family view provides 1 row per allowable family/user combination. The application view provides 1 row per allowable application/user combination.

All that is necessary to use security in any query is to have an application id or product family id available, and join to one or the other of these views. It is suggested that one or the other be used. The data from which these views are built comes from multiple sources that sometimes contain contradictory information. Best practice is to use application security, unless the granularity of the intended query is above or outside of the application context, which means essentially product or product family.

It is important to note that there IS NO DATA SECURITY without joining these tables into the query. Reports created and run by any user will return all records unless these tables are added. Security is implemented via queries, not on “raw” dimension or fact tables. This is similar to the LIQUENT InSight implementation. Security is implemented when the data is queried, but if someone connects to the database outside the application, there is no security.

Date Handling

A physical table with hard-coded values call **d_date** is provided for use in date-partition type queries. It contains rows for every day from 1951 to 2101. For each day, it provides the day number in the month, the long and short names for the month, the month number and the year.

To use this table, join the full_date column to whatever date column in a view is being evaluated. Note that the full_date column contains only month, day and year information. There is no clock/timestamp information. That means that relational operators must be properly used on date columns.

It is best to use <=, >=, >,< operators. Avoid the trunc function and the = operator. The trunc function performs best if indexed to facilitate its use, which we do not provide on every date column, and = will almost never return anything. This is a drawback of using a non-materialized schema. We do not want to over-index the underlying database, because in some cases, it may be the actual production database. If the schema were fully materialized, we could add many more indices, and even add truncated and non-truncated versions of columns.

Column Naming Conventions – Dimensions

Almost all columns in the dimensions have the EXACT name that they have in the underlying ism views. If conflicts arose in joins, clashing columns were prefixed or suffixed, to make it obvious what they mean.

In most cases, where the name of the dimension is vd_<dimname>, the unique identifier for that dimension is called <dimname>_id.

Example – ***vd_app_country****.app_country_id*

Exceptions:

vd_cta_shared_data.cta_shared_id (this is unintentional and may be corrected in future releases)

vd_manuf_function.function_type_id (this is unintentional and may be corrected in future releases)

vd_numerator_unit_type.concentration_unit_type_id (this is intentional and due to the remapping of the former concentration unit types to the new evmpd numerator units (and subsequent elimination of the concentration unit type table without renaming the many columns in the data that reference it. It is not out of the question that this might be changed in future releases).

Vd_status_code.status_id (this is intentional and due to the fact that all of the tables that have status codes call it status id and the thinking was that vd_status_code was more meaningful than vd_status; that may be a questionable idea at this point, so it is not out of the question that **vd_status_code** could be renamed to **vd_status** in the future).

All **vd_pds*_hist** dimensions have a unique identifier column called change_id. This is intentional because that is how we refer to it internally in the system. Typically, it can be assumed that the order of the change_id values within a specific detail is consistent with the order that events modified the pds. Additionally, all **vd_pds*hist** dimensions have a column called header_id, which is the internal unique identifier for the detail to which the change is attached. Fact tables that reference the change_id usually have columns called something like last_change_id or last_approved_change_id.

Column Naming Conventions – Fact Tables

*_id columns that reference fact tables have the exact name as the unique identifier column of the dimension. (See the dimension naming conventions, above). This provides a simple and understandable join between fact and dimension like **vf_ctxt_application**.application_id = **vd_application**.application_id.

Column Naming Conventions – General

When a column is a reference in the underlying data, but it is not a row key/identifier column, there will be two columns present: the original <xxxxx>_id column and an additional column called <xxxxx>_name.

> *Example* – vd_event.notify_type_id and vd_event.notify_type_name. The latter column is the name/string identifier for the _id column. Most of the time there is no need to even consider the id column in a query, just use the name column. This occurs mostly in dimension tables, but rarely, in fact tables as well. If an *_id column appears without a *_name column, that is a good indication that a corresponding dimension table exists, and a join should be made to the dimension to get the data.

Physical Implementation

All views (vd_, vf_*, vb_*) as well as the single table (d_date), are implemented in the dm schema.

Any custom views or reports should use ONLY data that is presented in the dm schema. It is preferred that custom views be stored in a separate, customer-defined schema. It will be necessary to grant select on the dm schema views and table(s) to the owner of the customer-defined schema.

Many LIQUENT InSight data admin entities are not represented in the reporting schema at this time. Important larger entities, such as substances, manufacturers, and indications (usually entities with many attributes) are included as dimensions. DA entities with only a name attribute are pre-linked into the relevant dimensions. Exceptions occur when we needed to provide data for drop-down lists in the out-of-the-box reports built in the reporting tool. We will add more as the need arises.

A workaround for the missing data admin entities is to construct the drop-down lists from the pre-resolved values that are found on the views. All data admin _id columns have been resolved into their meaningful names taken from the data admin tables. Of course, values that are not yet represented in the data will not appear when using this approach.

Insight Audit Data Views

There are two sets of views that present LIQUENT InSight Audit data. ID columns for data admin values are provided, but they have not been pre-resolved.

VH_* views

These views are the raw audit data. The extra fields columns are joined in. Since they are stored in a separate table, the entity_change* columns must be calculated.

If the change was triggered by a change in the extra fields, the entity_change_source will have an E. If the change was triggered by a change in the main entity columns, the change_source columns will have an X. If it is not possible to determine (i.e. the change dates on the main entity and on its extra fields are the same), the change_source will be null.

DF* views

These views are dynamic views. The first row of any query applied against these views will have all of the starting values of the earliest row returned by the query. Subsequent rows will only have a value for a given column if the value for that column changed since the last row. If there has been no change in the value of the column since the previous row, the value for the column will be null.

Note: if a value changes from non-null to null, there is no way to distinguish this from the normal situation of having no change in the value. This is a difficult technical problem to solve and will be looked at in the future.