Efficiency Gains in Inbound Data Warehouse Feed Implementation

Simon Eligulashvili

simon.e@gamma-sys.com

Introduction

The task of building a data warehouse with the objective of making it a long-term strategic platform represents many technical and business challenges. One of the determining factors of a successful implementation of the warehouse system is its ability to expand on the number of inbound data feeds over time with the pace required by the business. Besides the commonly recognized challenges presented by expanding the data inventory, such as ensuring system scalability and acceptable performance levels, capacity planning, and continuous adherence to enterprise guidelines, a critical part of this process lies in the methodology of implementing inbound feeds. While other aspects of the data warehouse lifecycle are equally important, it is imperative to work out an efficient inbound feed implementation methodology that will help reduce costs, time-to-market, and risks associated with development and maintenance of data feeds as the data warehouse evolves.

Conventional Inbound Feed Implementation Approach

Frequently, during the initial stages of data warehouse implementation, the emphasis is made on delivering the first set of inbound data feeds as quickly as possible, without giving a proper amount of consideration to the overall inbound feed architecture. As a result, the initial set of feeds is often implemented as monolithic modules that are costly to maintain, lacking the foundation that would allow reducing efforts of implementing subsequent feeds into the warehouse. As the data warehouse evolves and a need arises to source in more data feeds, each data stream gets implemented as a standalone process again and again, with a limited ability to effectively leverage previously built feeds, causing the costs and efforts to maintain the warehouse to grow unreasonably high.

In order to understand where efficiencies can be gained in respect to the inbound feed implementation, let us take a look at technical activities comprising implementation of each inbound feed. Typically, the feed development process consists of the following phases:

1. Data extraction and staging
2. Delta processing
3. Feed-specific exception checks and validations
4. Source-to-target transformations (translations, maps, aggregations, filters, normalizations/denormalizations, etc…)
5. Natural key map into existing generated (surrogate) DW keys
6. Surrogate key generation for new records
7. Referential integrity (RI) checks against parent warehouse tables
8. History management

It is noticeable that as data flows from sources to target warehouse structures, the nature of activities taking place changes from being source specific to target specific. For example, while the source-to-target transformations can be unique for each feed, phases like RI validation and history management are more of the target warehouse requirements. Given that it is common in data warehouse environments for disparate data streams to write into the same target structures, it can be concluded that the target specific processing phases (steps 5 to 8 in the list) can be reused across many feeds that write into the same tables. By reusing these phases, we have an opportunity to reduce costs and time of development by 30% to 50% on each inbound feed.

Considering this observation, a question arises of how to architect a solution that will eliminate the need to re-implement target-specific phases for each inbound feed and provide means to link source-specific logic to reusable self-contained target-specific processing modules.

Solution Geared Towards Maximizing Reusability

In the world of conventional application programming, a similar problem of encapsulating and reusing processing steps can be commonly solved by introducing the concept of application programming

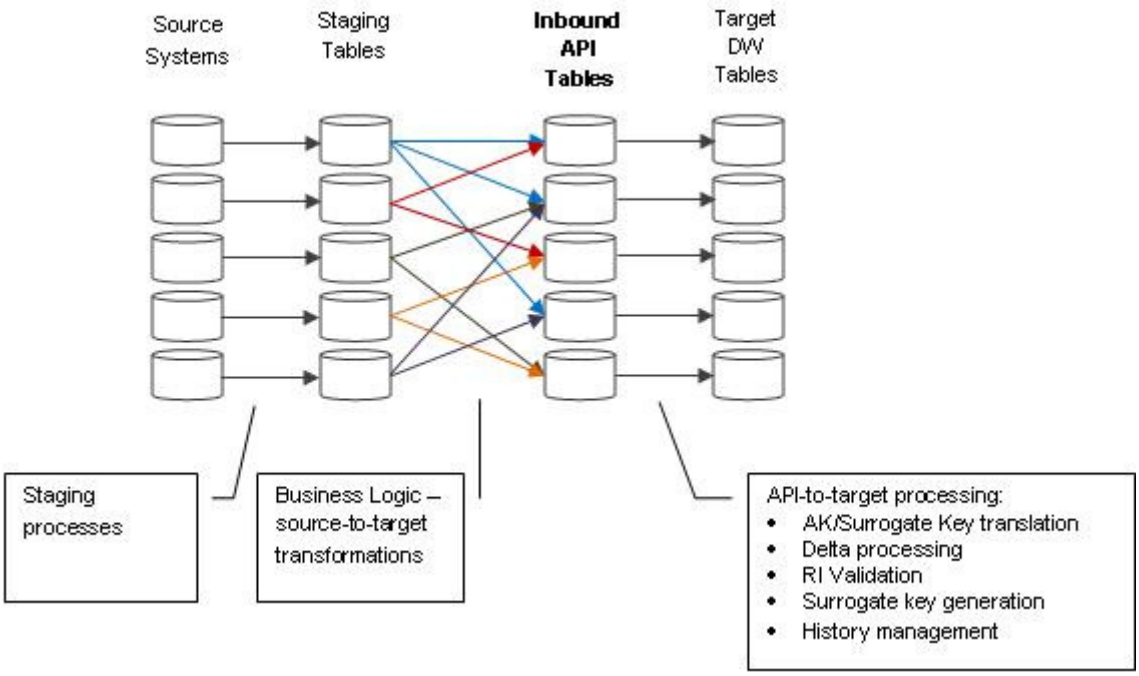
interface (API). In the data warehouse environment, an analogous concept can be utilized very effectively as well – APIs can be created to abstract the internal data management, validation, and storage rules of the warehouse from developers coding the inbound feeds. While in the application programming domain APIs may be in a form of functions with specific signatures, the inbound feed APIs in the data warehousing environment can be represented by a set of tables to hold incoming business and control values, and generic processes that move data from the API layer into the target warehouse structures.

Inbound Feed API Layer

As with the conventional API, a data warehouse architect has to determine the optimal design of the inbound API layer, balancing its flexibility, manageability, and ease of use. A well-balanced solution should present processing modules granular enough to support the flexibility requirements, but large enough to be self-contained and fully encapsulate the warehouse-side processing phases. Given that in data warehouse environments it is quite common for a single data stream to get dispersed into multiple target tables, and for a single warehouse table to receive data from multiple streams, it makes sense to create a separate abstraction point (e.g. API table) for each core warehouse table.

As a result of introducing the API layer, processing of inbound feeds from the staging area onwards will be split into two phases – 1) propagating data from the staging area into the API layer and applying only the source-specific transformation logic along the way and 2) moving data from the API layer into the core warehouse structures by invoking reusable modules that handle the warehouse-side requirements (depicted in diagram 1).

DIAGRAM 1 – Inbound Feed Architecture Utilizing Inbound API Layer

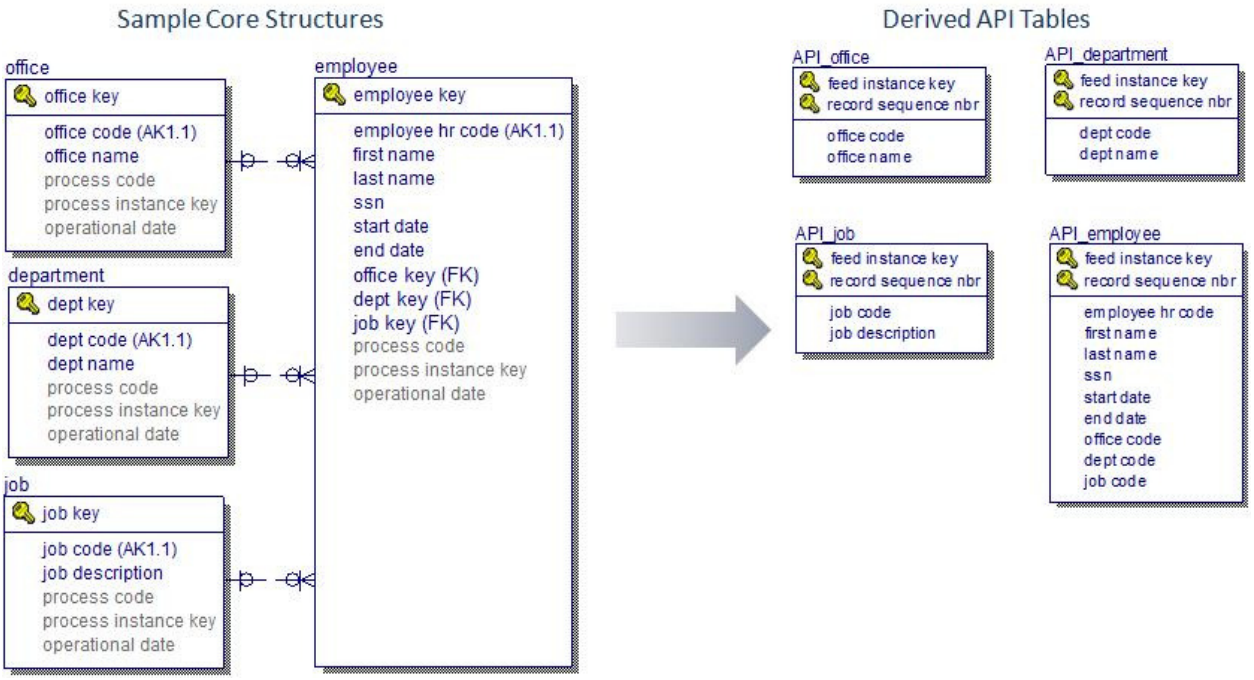


Considering that the goal of having the API layer is to abstract the feed implementation team from internal workings of the data warehouse, the following rules can be applied for creating each table in the API layer:

- All columns from a target warehouse table carrying business values are to be present in an API table, preserving data types, lengths, and nullable flags.
- An API table is to provide placeholder columns for process control, such as feed code, feed instance ID, record status, and record sequence numbers.

- Each foreign key column from the corresponding target warehouse table will be replaced by a set of columns representing the natural key on the referenced table
- The surrogate primary key column (if it is present on the target table) will be replaced by a set of columns representing the natural or alternate key of the target table
- Columns representing surrogate keys should either not be present or should be made nullable on API tables, as their values will be generated by the API-to-Target process.
- An API table representing a subtype target table can also include business-valued columns from the related super-type table, which can help to ensure proper population of super- and sub-type tables.
- API tables do not have any foreign key constraints, as they can function independently of each other.

The diagram below gives an example of how sample data warehouse tables can be transformed into the API layer tables:



Since an API table can fully abstract history management requirements for a corresponding warehouse table, there is no need to create API tables for historical tables (for example, type IV dimensions). A single API table can be used to abstract both tables with current and historical records, thus eliminating the need to write code to populate both tables for each inbound feed going into these tables.

Implementation Considerations for the API-to-Target Processes

Several approaches can be taken in implementing the processes that move data from the inbound API layer into the core warehouse tables. The choices mainly depend on the enterprise's standards and guidelines, particular requirements of the data warehouse project, and the software/hardware configuration of the warehouse system.

The most straightforward way of implementing these processes is building each API-to-Target process individually, using either an ETL tool or a native SQL. The downside of this approach is that as table structures of the warehouse change over time, one will have to synchronize individual processes manually.

Another method of propagating data from the API layer into target tables is creating a module that will generate SQL statements on the fly based on the source and target tables involved in a particular feed. While this approach provides more flexibility, this choice requires a more elaborate metadata model to support its functionality.

Regardless of the choice determined to be appropriate for a particular project, introduction of the API layer requires a framework that will allow tying all the components of a particular inbound feed into a logical unit of work.

The following functionality points should be considered when designing the API-to-Target processes in order to support basic levels of flexibility:

- Keeping track of the status of records to be moved from an API table into a warehouse table.
- Ability to control commit sizes as data gets written into warehouse tables
- Ability to restart processes from the point of failure and reprocess failed records
- Ability to invoke API-to-Target processes for specific inbound feeds and feed instances

Depending on specifics of a particular data warehouse environment, some of the points should be paid closer attention than others.

Benefits of the Inbound API Layer

There are several benefits in introducing the API layer in the inbound feed architecture.
First and foremost, the activities associated with supporting the internal data warehouse data storage and management rules do not have to be re-implemented for each feed, and instead are implemented once for each table, reducing the time-to-market, costs, and risks of implementation of each feed, and at the same time enforcing uniformity of implementation. The layer provides a clear isolation of source- and target-specific processes, making it easier to introduce new inbound feeds or replace feeds as needed throughout the life of the data warehouse. The layer ensures that the warehouse-side changes, for example, changes in history management requirements for a particular table, do not affect inbound feed implementation. Risks associated with data integrity are greatly reduced as a result of having to write and test less code that deals with the internal data warehouse management functionality.

Applications of Inbound API Layer

The concept of the inbound API layer is applicable in data warehouse environments where data originating from disparate sources gets written to the same (conformed) structures. There are other scenarios where the API layer can prove to be beneficial. If instances of the same data warehouse model are to be utilized at different sites, for example, at multiple branches of a firm, where sources are unique for each site, the model can be deployed along with the API layer and the API-to-target processes, so each site would only have to implement its own source-specific part, while the processes associated with the data validation, storage, and management rules of the data warehouse model remain unchanged. The same applies with a case of vendor-supplied data models – when deployed at distinct client sites, the model can still utilize the same pre-coded internal data management rules by introducing the inbound feed API layer.

Conclusion

Data warehouse projects, often being large and involving several technologies, offer great opportunities in respect to reusability of components and concepts. Rather than rushing into the implementation mode, it pays to go through thorough analysis and design stages, during which teams have a chance to

understand the magnitude of the task at hand and lay out the architectural base of the warehouse system. One of the objectives of the design phase of the project should be around creating a foundation that would allow teams to work on their future tasks efficiently, focusing on high-order activities and relying on the mechanisms put in place to support their activities. The inbound API layer would be a part of such foundation, as it eliminates the need to code repetitive and redundant data management and storage rules for distinct feeds going into warehouse tables.

Other essential aspects of data warehouse implementation, such as process management framework, exception handling, and outbound data extraction layers, can effectively utilize reusability techniques to enforce consistency and boost productivity across the entire data warehouse project.