Improving the Data Warehouse Architecture Using Design Patterns

Weiwen Yang
Colorado Technical University, Weiwen.yang@my.cs.colordotech.edu

Yanzhen Qu
Colorado Technical University, yqu@coloradotech.edu

Richard Fairley
Colorado Technical University, rfairley@coloradotech.edu

Improving the Data Warehouse Architecture Using Design Patterns

Weiwen Yang
Colorado Technical University
Weiwen.yang@my.cs.colordotech.edu

Yanzhen Qu
Colorado Technical University
yqu@coloradotech.edu

Richard Fairley
Colorado Technical University
rfairley@coloradotech.edu

ABSTRACT

Data warehousing is an important part of the enterprise information system. Business intelligence (BI) relies on data warehouses to improve business performance. Data quality plays a key role in BI. Source data is extracted, transformed, and loaded (ETL) into the data warehouses periodically. The ETL operations have the most crucial impact on the data quality of the data warehouse. ETL-related data warehouse architectures including structure-oriented layer architectures and enterprise-view data mart architecture were studied in the literature. Existing architectures have the layer and data mart components but do not make use of design patterns; thus, those approaches are inefficient and pose potential problems. This paper relays how to use design patterns to improve data warehouse architectures.

Keywords

Data warehouse, business intelligence, ETL, design pattern, layer pattern, bridge.

INTRODUCTION

In order to maintain and guarantee data quality, data warehouses must be updated periodically. The monolithic approach combines the extraction, transforming, and loading (ETL) operations into one step to pull data from sources to the data warehouse. All data comes in different formats and sizes, and some may need to be synthesized before loading into data warehouses. The one-step approach is long and complicated. Difficulty emerges not only in accommodating the changes when the sources or schema change, but also in troubleshooting and fixing problems, as the components are sizable and complex. In fact, the data sources and the output schema change once in a while. The monolithic approach does not easily accommodate changes in sources and output schema. Some operations are complicated and massive in scale and may exhaust too many resources to execute; running such a one-step, large-scale operation from the source to the data warehouse is not efficient in that case. Therefore, it is necessary to break the complicated operation into multiple steps. The high-level operation depends on lower ones. In general, the transformation depends on extraction, the validation depends on transformation, and the loading operations depend on validation. The data warehouse layer depends on the reconciled data layer, which depends on the source layer. The target system should be testable, easy to use and change, and efficient. Quality attributes are testability, performance, reliability, usability, scalability, data quality, manageability, extensibility, and modifiability. *Testability* refers to feasibility of testing a software system. *Performance* is how fast the software can run for a specific task. *Reliability* is the probability of failure in a period of time. *Usability* is how easily software is used. *Scalability* is the ability of a system to accommodate data growth. *Manageability* means how easily the system is administrated. *Extensibility* means the ability to add a new component. *Modifiability* is ability to modify a component. A *pattern* is a solution of the problem in a certain context (Alexander, 1979). A *design pattern* is a common recurring structure of the system components that solve a general design problem in a specific context, and it offers well-tested solutions for such problems experienced by many companies and academic institutions (Gamm, Helm, Johnson and Vlissides, 1993).

Golfarelli and Rizzi (2009) introduced a study of the structure-oriented layer architectures and enterprise-oriented-view data mart architectures. Sahama and Croll (2007) studied the enterprise data warehouse architecture, distributed data warehouse architecture, and data mart architecture. Bontempo and Zagelow (1998) studied the IBM distributed data warehouse architecture with a component approach based on IBM DB2 database systems with multiple ETL components including extracting, transforming, filtering, and data validation. Inmon, Strauss and Neushloss (2008) discussed the active data warehouse, star schema data warehouse, layer data warehouse, the federated data warehouse, and the data mart data warehouse. Kelly (1997) reviewed data warehouse architecture essential requirements. Watson and Ariyachandra (2005) discussed the independent data mart architecture, data mart bus architecture with linked dimensional data marts, as well as hub-and-spoke architecture. Golfarelli, Maio and Rizzi (1998) created a conceptual model with tree structures for data warehouses. Bonifati, Cattaneo, Ceri and Fuggetta (2001) proposed the identification and design of the data marts. Mazon and Trujillo (2009) described a hybrid framework for multi-dimensional data warehouses. Gardner (1998) discussed how to build the data warehouse.

In the traditional data warehouse architecture, the layers are designed in the monolithic approach. Breaking one layer leads to the failure of the whole system. Separating into many layers also increases the runtime overall. How to handle the metadata, alert, management information operation status such as job failures and completion in the data warehouse were not discussed. ACM, IEEE, data warehouse books and Internet material searches yielded no literature pertinent to design patterns in data warehouse architecture.

PROBLEM

The traditional architecture uses the layer approach or one-step approach from data source to data warehouse. The one-step approach is monolithic, includes many operations in a single step, and requires a large amount of memory and disk resources to run. If the required resources are more than actual resources the system can provide, it will not run. This one-step approach will lead to longer run-time as it is not processed in parallel. Job status may include running, completed, or aborted. An alert should be sent out to management if a job fails.. The management task may include starting a job, terminating a job, and querying *metadata* (which refers to the table schemas) and job status. User applications may have different ways to connect to the data warehouses. The existing architecture does not reveal how to implement the management and the user application component; it also fails to follow the design pattern principles. Hence, it leads to potential problems: less testability, less usability, less scalability, less reliability, and less efficiency.

METHODOLOGY AND BENEFITS OF DESIGN PATTERNS

The method herein involves applying the *layer pattern* with bridges, *publisher-subscriber pattern*, and *model-view-controller* (MVC) *model* to the data warehouse architecture. The design patterns are as follows:

1. The layer pattern with bridges is applied to the ETL component (Figure 1). One-step, large-scale operations that require more resources and have an exceptionally long runtime can be broken down into multiple operations to improve performance, modifiability, and scalability. Because it is a smaller layer with common interfaces, it is easily added or removed, thus promoting improved modifiability. Normal data flow moves from the data source layer, to reconciled layer, validation layer, and then data warehouse layer. There are two bridges from source data to data warehouse, and from reconciled layer to data warehouse. Data that is static and will not be modified is directly loaded from source layer to the data warehouse layer by passing both the reconciled and validation layer. Data not requiring validation can be loaded to the data warehouses by passing the validation layer. Passing a layer results in reducing runtime.
2. The publisher-subscriber pattern is applied to the management component. Publishers can be classified into many messages. The subscribers can receive the messages only if the subscriber subscribes to that message. The subscriber has no knowledge about the publishers. It is easy to add messages, delete messages, as well as to add and remove subscribers. This process improves scalability and flexibility.
3. The MVC model is used in the user-interface component and allows separation of the domain, actions, and presentation. The user-interface logic is separated from the business logic. The model does not depend on the view and controller, yet the view and controller depend on the model. The model, view, or controller can be built and tested independently. This allows for more client applications to increase scalability, testability, and usability.

NEW CONTRIBUTIONS

The new contributions of this paper are applying the layer pattern with bridges, the publisher-subscriber pattern, and MVC model to the traditional data warehouse architecture.

GENERAL ARCHITECTURE

The layer pattern with bridges is used in the architecture. The management component has the alert system, metadata, and management tools. The metadata is used by the ETL jobs. Management tools administer the jobs. An alert system is added to the ETL operations; then, an alert is sent out if an operation fails. Anyone who is subscribed to the alert system receives an email; this is the *publisher-subscriber pattern*. The data warehouse runs as a server, and the client can connect to the data warehouse via an internet connection. Furthermore, the bridge from the source to the validation layer is used to improve performance if the transformation is not needed. That is to say, the data of the source can be directly passed to the validation layer if the transformation is not needed. An MVC model is used in the user application layer. The target architecture is illustrated in Figure 1.

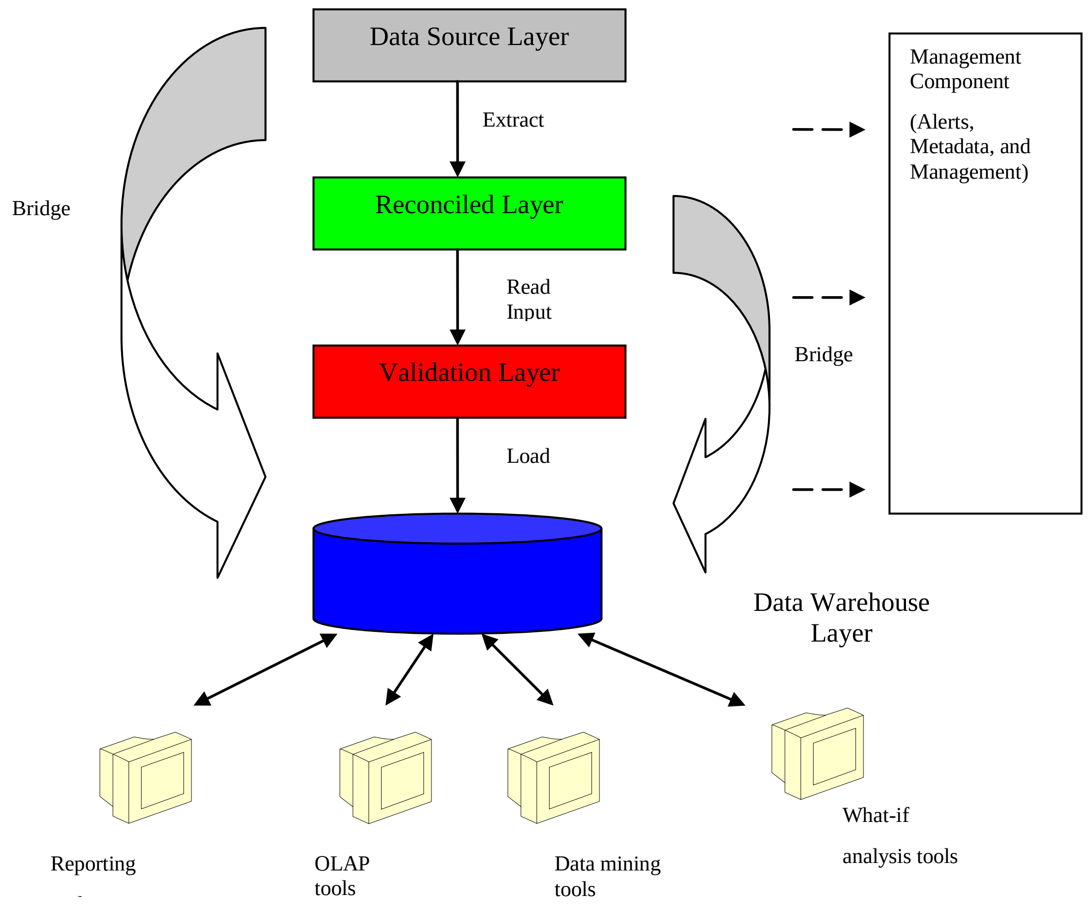

Figure 1. Modified hub-and-spoke architecture.

LAYER PATTERN WITH BRIDGES

The architectural layer pattern decomposes the tasks into groups of subtasks, where each group represents a specific level of abstraction (Buschmann, Meunier, Rohnert, Sommerlad and Stal, 1996). The context is such that the large system needs decomposition. The problem is that this system has the property of mixing high-level layers with low-level ones, with the higher level relying on the lower. A data warehouse can spread into multiple data centers and have up to millions of operations; those operations can be separated into groups, each whose operations represent a specific level of abstraction. As shown in Figure 1, the *modified hub-and-spoke architecture* includes the *source*, *reconciled*, *validation*, and *data warehouse layer* (Figure 1). The ETL process for building the data warehouse contains four layers and bridges:

1. The data source layer contains operational data, market data, business data, image, video, audio, and text—which are structured and unstructured data. The data is extracted into a table with a specific table schema in the staging area that prepares input for the reconciled layer.
2. The reconciled layer receives input from the data source layer and includes transformations. Data is transformed into a table in the staging area which will be used by the validation layer.
3. The validation layer will get input from the reconciled layer or source layer if the bridge is used. Data warehouses contain only dimension and fact tables. The dimension and fact data is validated according to some predefined business rules in the validation layer. The result will be stored in the staging area.

4. The data warehouse layer will get input from the validation layer, and the qualified data will be loaded into the data warehouse layer eventually.
5. The bridge from the source to the data warehouse layer is adopted if the transformation and validation are not needed. Sometimes the validation is simple and can be combined with the extraction. This will bypass the reconciled layer to reduce runtime, thus improving performance.
6. The bridge from the transformation to the data warehouse layer is adopted if the validation is not needed. Sometimes the validation is simple and can be combined with the transformation.

PUBLISHER-SUBSCRIBER PATTERN

The publisher-subscriber pattern maintains the state of cooperating party synchronization (Buschmann et al., 1996). The publisher notifies all subscribers if any change occurs. The alert system sends the alert to management or related personnel if the operation fails. Users who subscribe to the alert system can receive emails from the alert system. The idea is that a system needs to keep the state of cooperating parties synchronized. Data can be changed in one place, but other components depend on those data. The solution is that one component has the role of publishing, called *publisher*, and all the components that depend on the changing state of the publisher are called *subscribers*. The publisher has a registry of information on the currently subscribed components of the system. A dependent component can subscribe and unsubscribe to the publisher. The publisher may have different topics, and the subscriber subscribes to the topic of the publisher. When the publisher changes its state, it sends notification to all its subscribers. In this paper the subscriber can subscribe to the topic of invalid source data, extraction failure, transformation failure, validation failure, and loading failure of the publisher.

An alert is bound to each operation. If the operation fails, the alert will be sent to the subscriber. The first step is to check the data source based on some predefined business rules and send out an alert if it does not meet the predefined conditions. The second step is to extract the data if the data is valid and then send out an alert if extraction fails. The third step is to transform the data and send out an alert if transformation fails. The fourth step is to validate the data and send out an alert if validation fails. The fifth step is then to load the data into the data warehouse and send out an alert if the loading operation fails.

MODEL-VIEW-CONTROLLER PATTERN

The *model-view-controller pattern* (MVC) model (Figure 2) separates the application into three components representing the data, logic, and functionality (Buschmann et al., 1996). The views show the information to the users, and the controllers deal with the input and interact with the model to propagate the information to the views. The context is that an interactive application needs a flexible human computer interactive interface to present to the users. The problem occurs when a user interface receives changing requests from users. When the functionality of an application is extended, the corresponding menus need to be modified to access new functions. The user may call for a specific interface, or a system may be required to adapt to another platform with different "look and feel" criteria. Different users may input conflicting requirements to the user interface.

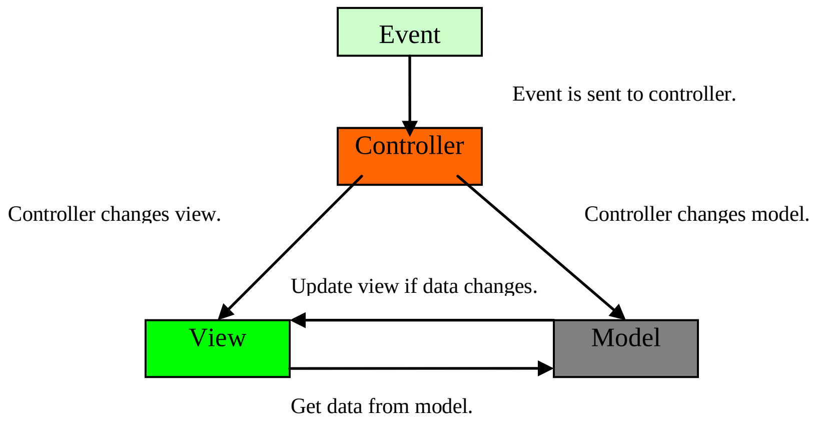
Event
Event is sent to controller.
Controller
Controller changes view.
Controller changes model.
Update view if data changes.
View
Model
Get data from model.

Figure 2. MVC model.

The MVC model was first introduced in SmallTalk-80 (Krasner and Pope, 1988). The model component encapsulates the main data and functions of the applications and is independent of specific input data or output results. The view component displays data to the user, and a view obtains data from the model. A model can have multiple views. Each view is associated with the input of the controller. The controller receives inputs or events from the mouse, keyboard, or other input devices, and the event is translated into requests. Users interact with the system through controllers. The purpose of separating the model from the view and the controller is to allow multiple views for the same model.

The structure contains model, controller, and view components. The model component includes the functional core of the user application and has the appropriate data and export procedure to process application requests. Controllers invoke the procedure for the users. The model provides functions to access the data. The change-propagation mechanism of the MVC pattern has a registry of the dependent components in the model. All views and the corresponding controllers register for informing about changes. Changing the state of the MVC model triggers the above change-propagation mechanism. The MVC pattern is based on the publisher-subscriber pattern. The MVC has components of model, view, and controller that are separated components and that communicate with each other via the registry and callback mechanism. The model is considered the *publisher*.

SUMMARY AND CONCLUSION

The data warehouse architectures were reviewed. The layer design patterns with bridges, publish-subscriber pattern, and the MVC model are added to the data warehouses to improve facets including usability, testability, performance, scalability, extensibility, and modifiability. The monolithic approach adds complication, consumes too many resources, and has excessive runtime. A smaller operation takes fewer resources and is easier to run compared a larger operation. Thus, breaking down the one-step operation into multiple, smaller operations in appropriate layers can improve performance. The layer pattern, publisher-subscriber pattern, and model-view-controller pattern are adopted to address the modifiability, manageability, performance, scalability, and extensibility in the architecture level.

The weakness is that breaking down into layers increases the number of layers for management. The failure of one layer may cause the entire data pipeline to fail. However, as discussed earlier, small operations require fewer resources and are more stable as compared to the monolithic operations. Adding the global component, validation, and alert component may increase the overhead somewhat, but they are essential to both the data quality and manageability. Overall, the approach still presents some disadvantages, yet those are outweighed by the advantages. Future research may include asynchronous, management, proxy, and communication patterns in the architecture.

REFERENCES

1. Alexander, C. (1979) The timeless way of building, Oxford University Press, 247.
2. Bontempo, C. and Zagelow, G. (1998) The IBM data warehouse architecture, *Communications of the ACM*, 41, 9, 38-48.
3. Bonifati, A., Cattaneo, F., Ceri, S. and Fuggetta, A. (2001) Designing data marts for data warehouses, *ACM Transactions on Software Engineering and Methodology*, 10, 4, 452–483.
4. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996) Pattern-oriented software architecture: A system of patterns. Hoboken, NJ: John Wiley & Sons, 25-339.
5. Gamm, E. Helm, R., Johnson, R. and Vlissides, J. (1993) Design patterns: Abstract and reusable of object-oriented design, *Proceedings of ECOOP93*, 406-431.
6. Gardner, S. R. (1998) Building the data warehouse, *Communications of the ACM*, 41, 9, 52-60.
7. Golfarelli, M., Maio, D. and Rizzi, S. (1998) Conceptual design of data warehouses from E/R schemes, *IEEE*, 1060-3425.
8. Golfarelli, M. and Rizzi, S. (2009) Data warehouse design: Modern principles and methodologies, Columbus, OH: McGraw Hill, 7-16.
9. Inmon, W. H., Strauss, D. and Neushloss, G. (2008) DW 2.0: The architecture for the next generation of data warehousing, Burlington, MA: Morgan Kaufmann, 14- 21.
10. Kelly, S. (1997) Data warehousing in action, Hoboken, NJ: John Wiley & Sons, 95-118.

11. Krasner, G. E. and Pope, S. T. (1988) A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1, 3, 26-49, August/September 1988, SIGS Publications, New York.

12. Mazon, J. and Trujillo, J. (2009) A hybrid model driven development framework for the multidimensional modeling of data warehouses, *SIGMOD Record*, 38, 2, 12-17.

13. Sahama, T. R. and Croll, P. R. (2007) Data warehouse architecture for clinical data warehousing. *Australian Computer Society*, *First Australian Workshop on Health Knowledge Management and Discovery (HKMD).*

14. Watson, H. J. and Ariyachandra, T. (2008) Data warehouse architectures: Factors in the selection, decision, and the success of the architectures, *Communications of the ACM,* 51, 10, 147.