A Formal Specification of Web-Based Data Warehouses

by

Indratmo

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree

Master of Science

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

A Formal Specification of Web-Based Data Warehouses

BY

Indratmo

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

The potential of using data warehouses to support decision-making processes has attracted many researchers to propose various data models and methods to produce high performance data warehouse systems. It is a long and expensive process to develop a data warehouse system. The user requirements and the data warehouse design must be specified properly. To design a good data warehouse requires a solid understanding of the fundamental principles in database systems as the underlying technology that implements the data warehouse.

This thesis provides a formal foundation for data warehouse system design by identifying and formalising the main concepts of database systems related to data warehouses. The foundation consists of three main sections: a requirements specification, a formal model, and validation of web-based data warehouses. The specification is written in the Z specification language, and has been type checked using Z/EVES. Type checking guarantees the correctness of types and syntax used in the specification. Furthermore, rigorous proofs are presented to show that the specification has captured desired properties of the system correctly.

Acknowledgements

Firstly, I would like to thank my advisor and friend, Dr. Sylvanus Ehikioya, for his efforts, encouragements, patience, and time in guiding me through this work. During his busy days, Dr. Ehikioya always managed his time to welcome me to discuss many things regarding this thesis.

I would also like to thank the examining committee members, Dr. Peter C.J. Graham and Dr. Mary Brabston for reviewing and evaluating this thesis carefully. Their suggestions have improved the presentation of this thesis. It is my pleasure to have experts in distributed systems, and information systems on my thesis committee.

Special thanks to my host family, Victor and Mary Friesen, who welcomed me to stay in their home during my first week in Canada. Their kindness made my transition to Canada very easy.

I am very glad to have many good and helpful friends. Chunming Chen read the entire thesis and gave valuable comments. Hadi Sulistio gave me a good data warehouse book that helped me understand the subject better. Iskandar and Yulianti were kind to me like my parents. Their love made my days in Canada beautiful.

I would not have had the opportunity to study in Canada without support from my parents who managed to send me to Canada although they had a hard time in Indonesia. I am fortunate to have parents and sisters who are always supportive. I love all of you.

Finally, my love goes to Joyce Boedianto who makes my life more beautiful with her love, kindness, and understanding. Joyce always brings happiness to my days and cheers me up during my moody days. Without her support, I would not have finished my thesis on time.

For all this bliss, I thank God and Mary Mother. May God always bless all of you and your families!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The tremendous development of Internet technology has attracted many companies to deploy their applications online. An online application provides flexible worldwide access for the users. Users are no longer limited by geographical locations or office hours to access the company's services.

The trend to deploy applications on the Internet can be applied to data warehouses as well. As part of decision support systems, data warehouses play important roles in many companies. This research focuses on a formal specification of web-based data warehouses. In this chapter, I present the benefits of a web-based data warehouse, problems in data warehousing, and the scope of this research.

1.1 Motivation

Data warehousing is a relatively new research area in database systems. The use of data warehouses as decision support systems requires a new design approach. The structure of a data warehouse is different from the structure of a database that is used for transactional processing. While a transactional database is optimized to run a business, a data warehouse is built for analysing business activities and spotting trends.

A data warehouse enables business analysts to explore the historical data of a company to find certain patterns that are beneficial for the business. For example, by knowing

customers' buying behaviours, a manager can plan a marketing strategy that fits with those behaviours, thereby delivering products to the right customers at the right time.

The advantages of having a data warehouse can be expanded by providing easy access for the users. It is not uncommon for managers to travel for business negotiations. The ability to access a data warehouse remotely can increase the use of the data warehouse. While on business trips, managers can still perform various analyses to support their decisions.

Fortunately, the Internet has provided the infrastructure for deploying data warehouses online. The World Wide Web (web) has evolved from simple text and graphics applications to complex e-commerce and database applications. By using existing technologies for web-database access, we can deploy data warehouses on the web.

A web-based data warehouse offers many advantages; users only need a web browser to access the data warehouse, any changes in the data warehouse system are transparent to the users, and the network infrastructure exists widely. In addition, due to the web's popularity, a web-based data warehouse is benefits from the users' familiarity with the web interface. Internet users already know how to use a web browser.

However, a web-based data warehouse has its own limitations and is more prone to unauthorized access. Security is one of the main concerns in developing Internet applications. Furthermore, Internet connections are unreliable. These issues must be addressed properly in designing and implementing a web-based data warehouse system.

By considering the importance of information in the business world, we can categorize a web-based data warehouse as a business-critical system. A web-based data warehouse has potential to give significant improvements to a company's revenue. Therefore, we should not use an ad hoc approach to develop the system.

1.2 Problem Definition

In the highly competitive business world, it is crucial for a company to have a high quality information system. Well-informed decisions can increase the effectiveness of a marketing or promotion strategy, as well as strategic and tactical policies. Unfortunately, valuable

information is usually buried under a vast amount of data.

Operational data is an excellent data source for a decision-support system. It contains detailed transactions that have occurred in a company. Based on such data, various analyses can be performed to find hidden patterns that are important for the company.

Operational data has various forms and structures. Different subsystems in a company may use different DBMSs (Database Management Systems). A department may store its data as spreadsheets while another department may use a DBMS to manage its data. In addition, operational data may have evolved from its original design to cope with changing business requirements. These issues lead to the conclusion that we cannot use operational data directly for business-critical queries and analyses. The operational data must be extracted, organized and integrated into one consistent database, which is now known as a "data warehouse". This need for separation between operational data and data warehouse (informational data) is also identified in [14].

Building a web-based data warehouse is a complex process. End-users of a data warehouse are mostly business analysts who understand the business process well but may have limited knowledge about information technology. On the other hand, data warehouse developers may be experts regarding technical details but have little knowledge about the business process. Therefore, experts from both sides must collaborate to design and develop a data warehouse system.

Requirements gathering is a critical stage in data warehouse development. At this stage, designers and end-users must communicate to find out the requirements for the system. Any problems that may arise at this stage must be resolved properly. Expectations and limitations of the system must be stated clearly to avoid rejection of the final product. Unfortunately, end-users usually only have vague requirements in the early phases of the development life cycle. It is common that a requirements specification of a system evolves along with the development process.

End-users must validate the ongoing specification regularly. This validation is important to avoid misunderstanding between developers and end-users. Natural language, solely, is insufficient to specify requirements for a complex system unambiguously. The use of natural

language leads to ambiguity. The design should be accompanied with diagrams and/or formal notations to prevent this possibility of ambiguity. The focus, however, should not merely be on the formal notations or diagrams used, but rather the use of these methods should supplement the communication among the developers.

By looking at the scope and development process of a data warehouse, we can see that to develop a data warehouse is a long-term and critical project. People from multidisciplinary areas are involved in a data warehouse development. Fundamental principles in data warehousing, database systems, and software engineering need to be identified and integrated in a data warehouse design.

To design and implement a data warehouse properly, developers must have a solid understanding of the system. During the design stage, the system must be presented at an appropriate level of abstraction, so the developers are able to find the essential properties of the system without being buried under implementation details. The need for an abstract model is where a formal specification fits in the development life cycle. To write a formal specification, it is necessary for the designers to abstract and think clearly about the system. This way of thinking promotes a better design, which is crucial in a long-term, critical project such as data warehousing.

Even though formal methods offer many advantages, they have limitations as well. For example, it is hard to model user interfaces using formal methods. This suggests combining the use of formal methods with other software engineering methods. In this thesis, I use Z [39] and UML [8] to specify web-based data warehouses. The Z specification provides a formal model of the system, while the UML diagrams describe the system visually. Such visual modelling can give intuition to the readers about the system specified.

Many aspects of a data warehouse design depend on the application domain. A data warehouse for an insurance company is different from a data warehouse for a retail company, however, the underlying design principles are similar. To limit the discussion in this thesis, I focus on the general/fundamental concepts of web-based data warehouses. The following three main topics are discussed:

1. Requirements for web-based data warehouses.

A requirements specification describes properties that are necessary in a system. The requirements gathering process is considered one of the most challenging parts in the software development life cycle [34]. This section tries to identify and summarize the general requirements for web-based data warehouses.

2. Formal specification of web-based data warehouses.

   This section presents a formal specification of web-based data warehouses. The focus is to abstract and identify the fundamental aspects of the system. Specifically, I specify the following concepts: data warehouse schema, partitioning, aggregation, data marting, metadata, and basic functionalities of web-based data warehouses.

3. Validation and error checking of the specification.

   To ensure that the specification is error free, I perform type and domain checking using Z-EVES [29]. Type checking guarantees the correctness of syntax and types used in the specification. Domain checking ensures that the specification is meaningful [37]. In addition to these mechanical checks, I provide rigorous mathematical proofs of some critical properties of the system.

This thesis is not intended to provide a complete review of data warehouses, however, I try to identify and present the main ideas of each data warehouse component discussed.

1.3 Preview of Contributions

This thesis contributes to several aspects of research in data warehouses. It formalises the concepts of data warehouses and models basic functionalities of web-based data warehouses. The fundamental principles in data warehousing, database systems, and software engineering are identified and integrated in the specification. I expect that the combination of formal notations and visual modeling will result in a rigorous, yet understandable, specification for data warehouses.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 provides background information and a literature review. Chapter 3 presents the basic requirements for web-based data warehouses and provides a formal model of the system. Chapter 4 describes the error checking procedures used and provides rigorous proofs to validate the specification. Finally, Chapter 5 presents the conclusions and suggests some directions for future work.

Chapter 2

Background and Related Work

2.1 An Overview of Data Warehouses

A data warehouse is a repository of integrated data from various heterogeneous data sources [40]. The generic, high level architecture for a data warehouse is shown in Figure 2.1. Operational data, which is the source for a data warehouse, is extracted, cleaned, integrated, and loaded into the data warehouse. Due to the complexity of a data warehouse, information about the data and the warehousing process must be captured as metadata. The integrated data is then used by end-users to support decision-making processes.

The main purpose of a data warehouse is to provide better resources for decision support systems. Top managers usually require an overall view of a company. An overall view involves data from different departments in the company. Without a warehousing process,

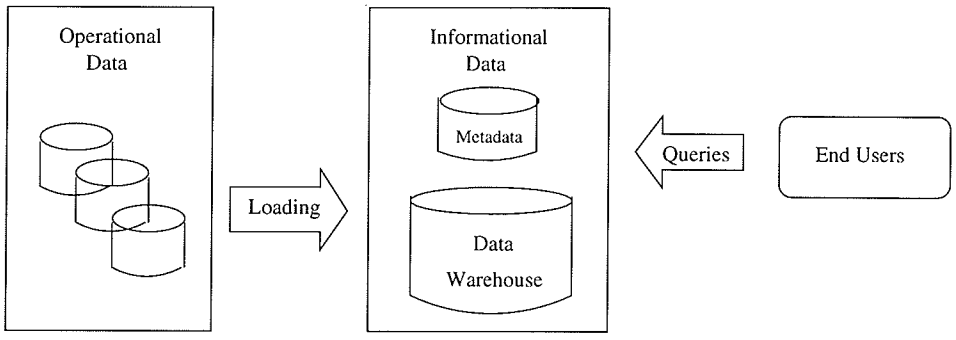

Figure 2.1: The generic architecture of a data warehouse

accessing data directly from the departments may result in inconsistent information. For example, a business term used in a department may have different meaning in another department. This kind of discrepancy must be resolved during the warehousing process.

2.2 Fundamental Concepts of Data Warehouses

2.2.1 Data Warehouse Schema

The core of a data warehouse is the data warehouse schema. A data warehouse schema is a structure that describes the organisation of data for decision support analyses.

Designing a data warehouse schema is a non-trivial task. Due to the large volume of a data warehouse, restructuring the schema after the data warehouse is implemented is very costly. It may be as expensive as building a new data warehouse. Thus, a data warehouse schema must be designed properly from the beginning.

One of the main requirements for a data warehouse schema is the flexibility to support On-Line Analytical Processing (OLAP). This requirement is critical due to the constantly evolving business and user requirements. Since business and user requirements are only partially known at the design stage, it is impossible to predict all queries that may be submitted by users in advance.

To cope with unpredictable user requirements, a data warehouse should not be designed based on specific query requirements [4]. Instead, a data warehouse should be designed to support arbitrary decision-making processes.

Decision-makers usually need to analyse a significant fact from multiple dimensions. Common OLAP operations in decision support systems are to "slice", "dice", "drill-down", and "roll-up". To slice is to select a subset of data in a data warehouse (e.g., selecting sales data in certain cities). To dice is to analyse data from different dimensions. For example, a marketing manager may be interested in analysing a sales report over a certain period of time and store locations. The manager may also want to know what the favourite products in different stores are. Thus, the manager needs to create another sales report over store locations and product types. To drill-down is to produce a more-detailed report. For

example, after creating an annual sales report, a manager can examine the report further by drilling down the annual sales into quarter sales. Conversely, to roll-up is to produce a less-detailed report.

The data warehouse research community has identified and proposed database schemas that are suitable for supporting OLAP applications. The "star" model and "snowflake" model are two basic models for a data warehouse schema [5]. These schemas are based on the relational data model, and have characteristics that are suitable for answering multidimensional queries.

In a star schema, a fact table is placed in the centre, surrounded by its dimensions (Figure 2.2). Fact tables contain numerical data (measures) that record past data transactions; hence the data tend to be consistent. For example, how many product items that a customer bought, how much the products were, how much the total transaction was, etc. Dimension tables contain information that describes data in fact tables. For example, what kind of products that a customer bought, when and where the customer bought the products, etc. This structure allows business analysts to examine and construct views from various dimensions. The reference information is more dynamic. However, because of the star schema structure, the reference information can evolve and be updated without restructuring the whole data warehouse schema.

The snowflake schema has a similar structure to the star schema. It has a fact table in the centre, and a set of dimension and sub-dimension tables (Figure 2.3). Sub-dimension tables are produced by normalizing dimension tables.

Data modeling is an active research area. Much research has been done which proposes data models that are suitable for data warehouse/OLAP applications. Agrawal, *et al.* [2] propose a *hypercube-based* data model and a set of minimal algebraic operations that are suitable for multidimensional queries. The model treats measures and dimensions symmetrically. Thus, operators on dimensions can be applied on measures as well, for example, using a measure to categorize data. Furthermore, the operators are kept close to relational algebra to make them translatable to SQL.

Li and Wang [28] present a formal model of multidimensional data and develop a query

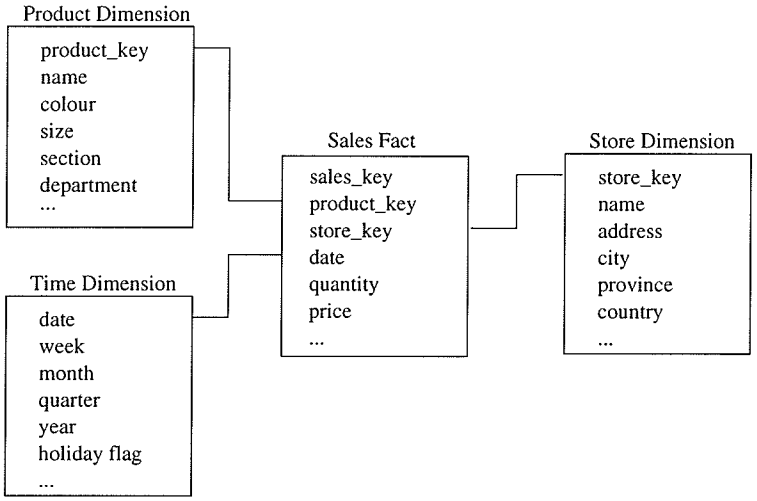

Figure 2.2: An example of a star schema

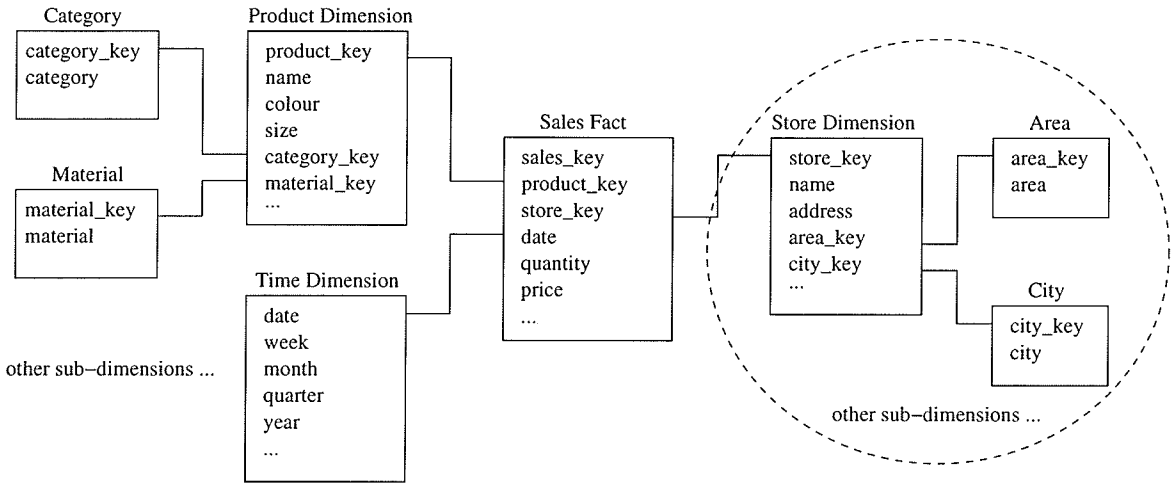

Figure 2.3: An example of a snowflake schema

language called *grouping algebra* that is flexible in expressing OLAP queries. They define a multidimensional cube as the basic component of their data model. The cube consists of a set of dimension tables where each combination of tuples from the dimensions has an associated data values.

To simplify data manipulation in multidimensional databases, Gyssens and Lakshmanan [20] separate structural aspects from the contents of their multidimensional database model. They show that data cube operators can be expressed in a simple way as the result of the structural and content separation. They also develop an algebra and its equivalent calculus for multidimensional databases.

These multidimensional models have recently emerged because the traditional relational model is generally inadequate to support OLAP queries efficiently [20, 28, 33]. The multidimensional models store highly summarized data in arrays [25]. The use of arrays avoids complex joins that decrease query performance in the relational model. However, the multidimensional models are less flexible than the relational model. Whenever the dimensional structure changes, the multidimensional database must be reorganized completely [25].

Both relational and multidimensional models have their own positions in data warehouses. The relational model is more appropriate for enterprise data warehouses where we anticipate evolving dimensional structures. The multidimensional models are suitable for data marts that have specific purposes.

In addition to the theoretical aspects of data modeling, Kimbal [26] presents various sample cases of multidimensional modeling based on real-life scenarios. An in-depth discussion and an estimated size of the data warehouse are given for each model.

2.2.2 Partitioning

To partition is to divide a (usually) large table into several tables. Due to the vast amount of data in a data warehouse, we need to partition the data based on certain criteria. In data warehouse systems, partitioning is inevitable to simplify data management and increase query performance. Small tables are more manageable than very large tables. In addition, small tables usually give better query performance. For example, consider a fact table that

keeps a sales history since 1995. If we do not partition the table, when a query looks for a record in 1998, the query may have to scan from 1995's record. The query will be faster, for example, if the table is partitioned yearly.

Partitioning also enables queries to be processed in parallel. Table partitions can be located in different nodes. This distributed allocation enables a query to be divided into sub-queries that run parallel in these nodes. However, the distributed design must be examined carefully since there are many essential factors to consider, for example, the system architecture, the criteria used to partition the data, the allocation strategy, etc.

To guarantee the integrity of the database, there are rules of partitioning that must be obeyed. These rules are *completeness*, *reconstruction*, and *disjointness* [32]. These principles ensure that all data from the original table can be found in a resulting table partition (completeness); table partitions can be used to construct the original table (reconstruction), hence preserving any constraints or functional dependencies in the original table; and data in table partitions are not overlapped (disjointness). Özsu and Valduriez [32] define a formal model of simple and minterm predicates in partitioning.

Ezeife [17] proposes an algorithm to select a data warehouse view, and to partition the view horizontally based on certain benefit criteria applied to the predicates.

Noaman and Barker [31] propose an algorithm to partition a huge fact relation horizontally. Instead of using predicates defined on only one dimension relation, they use predicates defined on all the dimension relations to derive the horizontal fragments. Their approach produces a set of table partitions that reflects the multidimensional user queries against the data warehouse.

One dimension that is usually used to partition a data warehouse is time [4, 24]. The partition will be optimal if the users' query patterns are known. For example, if most queries access monthly data, it will be faster if the data warehouse is partitioned into monthly segments. Unfortunately, at design time, these patterns are not known completely. Thus, the initial degree of partitioning should normally be at a moderate level.

2.2.3 Aggregation

To aggregate is to summarize data based on certain criteria, e.g. aggregate sales of a product in a particular city. This operation is important in data warehouses because business analysts are more interested in analysing summary data, instead of individual transaction data. Summary data provides more condensed information about an enterprise. Enterprise-wide information is essential to shape the company's strategies; however, detailed data is still needed to anticipate user requirements. For example, there may be cases when users want to examine detailed transaction records to find the cause of a business phenomenon.

Data warehouses, as part of decision support systems, should be optimized to support aggregate queries. It takes a considerable amount of time to process aggregate queries on the fly. If the majority of queries will access summary data, we can increase query performance by aggregating the commonly accessed values at an appropriate level [4, 27]. For example, suppose analysts tend to access the total monthly sales at each store or city frequently. Rather than calculating these values when such queries occur, we can pre-aggregate the monthly sales at each store and keep the summary tables in the data warehouse server. When aggregate queries occur, the data warehouse server does not need to calculate these aggregate values from the base tables, but it can use the pre-aggregated data in the summary tables. To achieve improved query performance, the pre-aggregated tables must be adjusted/restructured when the query profiles change.

Some research has been done to try to speed up aggregation operations. Agarwal, *et al.* [1] propose two optimized algorithms based on the sort-based and hash-based grouping methods for computing the CUBE operator. Gupta, *et al.* [19] developed an algorithm that exploits the use of pre-existing materialised views (summary tables) to answer aggregate queries. Generally, better optimization of aggregate queries will have a significant impact on the overall performance of a data warehouse [19].

There are various factors to consider in aggregating data. The factors vary from determining which data to aggregate to deciding the level of aggregation. These factors are application dependent, because they depend on user query profiles. Thus, I will not discuss them in the specification. Instead, I model aggregate functions based on standard SQL as

described in [11, 12, 16].

2.2.4 Data Marting

Organizations usually have different sub-systems/departments based on particular functionality. For example, a marketing department is responsible for planning and promoting products; a production department is responsible for producing products.

An enterprise data warehouse integrates various data from many different departments in the company. However, applications in each sub-system will mostly access data related to the sub-system. We can take advantage of this characteristic by dividing a data warehouse into "data marts" that contain only specific data needed by each sub-system. Since data marts are much smaller than an enterprise data warehouse, it is expected that data marting can result in better average-case query performance for the system.

The main problem in data marting is to decide what criteria to use to divide the enterprise data warehouse. The category must be chosen properly to minimize the risk of having to rebuild the data marts. The risk comes from the possibility of structural changes in the organization. For example, a branch office might have been responsible for sales in cities A, B, and C, but when the company changes its organizational structure, the branch office becomes responsible for sales in cities A and B only. If the enterprise data was divided based on the city allocation to branch offices, the data marts would then have to be reorganized.

Data marting must ensure consistency of the data. Data in data marts must be consistent with data in the enterprise data warehouse; otherwise, the quality of information in the data warehouse will decline. Due to maintenance issues, building a data mart is not mandatory [4, 27]. The decision to mart an enterprise data warehouse should be supported by a good, relatively stable organizational structure of the enterprise.

2.2.5 Metadata

Metadata has significant roles in data warehouses. It contains data warehouse schemas, and information about the warehousing process that is essential for warehouse use and maintenance. DBMSs use metadata to apply various constraints defined in the database

schemas, and to locate data when processing queries.

From an end-user perspective, metadata is analogous to a library catalogue [18]. While a library catalogue helps people find particular books and their locations in the library, metadata provides important information that helps users use the data warehouse effectively.

We can categorize metadata into two groups: technical and business metadata [23]. Both have significant roles in data warehouse systems. Technical metadata facilitates warehouse administrators' maintenance of the data warehouse. Business metadata assists end-users to understand the meaning of informational objects in the warehouse.

Technical metadata refers to all information related to data warehouse design and implementation. This includes information about data transformation and loading, data management, and query generation [4]. Information about data sources and the way they were integrated is needed, especially if the gap between the data sources and the data warehouse is big. The "gap" between the sources and the data warehouse is big if their structures are so different, lots of inconsistent data exist in the sources, and the transformation functions that are applied to the data sources are complex. Data management involves complex database structures, data marting, and partitioning and aggregation schemas. Query information is captured to recognize users' query profiles. By knowing such profiles, some adjustments can be made to increase system performance.

It is not uncommon for a data warehouse to employ various tools from different vendors. Due to the lack of a metadata standard, different tools may use different formats and terms to describe the same object. This makes the metadata integration process difficult. To resolve this integration problem, several industry vendors have attempted to define a standard for metadata management [23].

Stöhr, *et al.* [30] introduced a metadata model that integrates both technical and semantic (or business) aspects. They use UML to represent their model.

2.2.6 Requirements Specification

The requirements gathering process is a crucial stage in the software development life cycle. At this stage, designers try to identify all requirements for the product. This process will result in a requirements specification.

A requirements specification is a document that describes the intended functionality and qualities of a product. In this document, designers should focus on *what* the product will do instead of *how* to implement the needed functionality. In other words, the specification should be abstract and technology independent to enable requirements reuse in the future [36]. A requirements specification is the base used by developers to build a product; hence it is very important and must be prepared properly.

A requirements specification can be categorized into two sections. The first section, functional requirements, defines a list of functionality that a product should possess. The second section, non-functional requirements, specifies the quality of the functionality. For example, how fast the system will process a query, or how secure the system will be.

2.3 Architecture of Web-Based Data Warehouses

This section presents an architecture for web-based data warehouses. The architecture is based on the three-tier client/server architecture. The three-tier architecture was proposed to resolve the limitations of classical two-tier client/server architectures. To give more background information, an overview of file-based systems and two-tier architectures are discussed.

In file-based systems, each application maintains its own data. If an application is run on multiple machines, it is hard to guarantee the consistency of the data. The two-tier architecture was proposed to solve inconsistency problems that arise in file-based systems. In the two-tier architecture, data and application logic are separated in different layers. Application logic resides on client computers, while data is centralized in a database server.

While the two-tier architecture partially resolves the inconsistency problems in file-based systems, the system requires clients to run an application program that accesses data in a

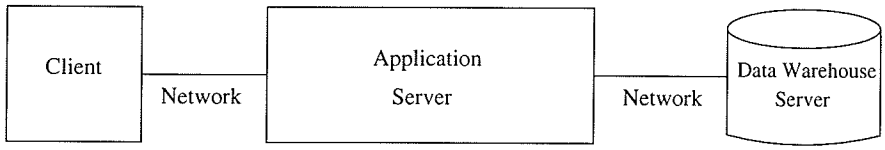

Figure 2.4: The three-tier client/server architecture

database server. This application program must be installed on all clients. If the application program changes, then all clients have to upgrade their programs. This requirement makes the client machines difficult to administer.

To resolve the problems in the two-tier architecture, the three-tier architecture was introduced. The three-tier architecture separates application logic from the client and adds it as another layer (Figure 2.4). This architecture is suitable for web-based data warehouses.

- **Client**: In the three-tier client/server architecture, application logic does not reside on the clients' computers. The application logic is located in a separate layer. Therefore, a client only needs a web browser that provides the user interface to the system. Basically, a client uses a web browser to send requests and to display results from the application server.

- **Application server**: Instead of putting the application programs on the clients, the three-tier architecture adds a new layer called an application server. The server provides application programs that can communicate with web browsers on the client sites and the data warehouse server on the back-end. Thus, an application server has to transform inputs from clients to a language that can be understood by the data warehouse server. As well, after receiving query results from the data warehouse server, the application server must generate HTML documents to be sent back to the clients. The implementation of an application server can actually consist of several layers (e.g. see Figure 2.5).

- **Data warehouse server**: A data warehouse server stores and maintains data. By separating the application server from the data warehouse server, the data warehouse server can optimize its resources for processing queries; hence it is able to deliver good

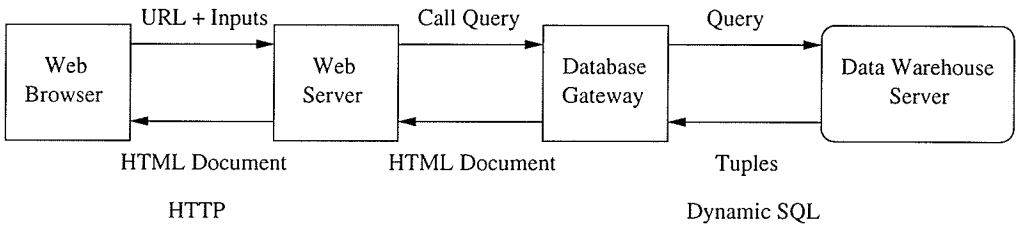

Figure 2.5: Architecture for web-based data warehouses (adapted from [32])

performance that is critical in web-based data warehouse systems. An application server and a data warehouse server can also reside on the same computer. This avoids data transmission overhead. However, the server must have enough computer resources to serve as an application and data warehouse server efficiently.

Figure 2.5 shows a basic architecture for web-based data warehouses. A web browser sends a URL and an input query to a web server using the HTTP protocol. The web server passes the input query to a database gateway that translates the input into SQL, and sends the SQL query to a data warehouse server. After processing the query, the data warehouse server sends the resulting tuples to the database gateway that transforms the tuples into an HTML document. The web server receives the query results in the form of an HTML document and sends the document back to the web browser.

Interaction between clients and servers using the HTTP communication protocol is stateless (i.e. a web server does not maintain a client's state between requests.). This also means, the web server does not recognise when multiple requests come from the same authorized client. This stateless interaction poses a problem in web-based data warehouse systems. A data warehouse server must authenticate clients for security reasons. With the stateless interaction, the system must have a mechanism to track a client between requests. As long as the requests come from an authorized user, the server must allow the user to access the data warehouse without repeating the authentication procedure.

Examples of user tracking mechanisms are the use of cookies and session objects. Cookies are small pieces of information that are originally sent by a web server to a web browser, and returned unchanged to the web server every time the browser visits the same web

site or domain [21]. A cookie contains specific information about a client, for example, a user name and a password. This information can be used by a web server to track the client between requests. However, some users choose to disable cookies because of privacy concerns. Thus, alternative mechanisms must be used. Session objects are one such alternative. The *HttpSession* API in servlets provides a high-level interface to a user-tracking mechanism [21]. A servlet developer only needs to determine what information is to be stored in a session object. The API, which is built on top of cookies, takes care of the details of how to implement the tracking mechanism.

2.4 Formal Methods

Formal methods are techniques for specifying a system using formal notations. The notations used in formal methods are usually based on logic and mathematical notations. Logic and mathematical concepts are universal and have been developed for centuries and thus they have well defined structures and semantics. In addition, mathematical notations are provable. Properties of a system that are specified formally can be validated using various proof techniques in mathematics.

A formal specification is an abstraction of a system. A formal model hides implementation details by capturing only the essential properties of a system. A formal model specifies *what* a system should be rather than *how* it should be built. Thus, it is appropriate to use formal methods in the early phases of the software development life cycle. Formal models should always be accompanied with explanatory text/diagrams and be developed using well-defined software development methods [6]. Formal methods are complements to, rather than replacements for, existing software development techniques.

2.4.1 The Benefits

Formal methods are mostly used in safety-critical system developments, because they can reduce the verification and validation costs that are crucial in such systems [38]. Formal methods have characteristics that make them suitable for critical systems development:

- Rigorous specification of systems.

  To formally specify a system, developers must have a good understanding of the system. This means that the developers have to spend more time in requirements analysis and design [6]. A good understanding leads to a good design. A good design is the foundation for a good final product.

- Errors can be caught early.

  Because formal methods use mathematical notations, desired properties of the system specified can be validated formally or rigorously. The process of proving a system's properties helps to reveal errors in the specification. Therefore, errors in a model/design can be caught early at the specification stage, instead of later during the development stage.

- Reduce verification, validation, and maintenance costs.

  A formal design process, if done properly, can detect and eliminate errors early. Thus, during the verification and validation stages, fewer errors will be found. Furthermore, products with fewer errors will incur less maintenance costs in the long run.

- Increase the quality of products.

  A formal specification is aimed at avoiding any ambiguities that may arise by the use of a natural language alone. By reading an unambiguous specification, developers are able to interpret the specification consistently. This certainty facilitates building the system properly. In addition, a formal model can be used to validate the design to the users. A list of examples of successful software projects using formal methods can be found in [38]. In those projects, fewer errors are reported as a result of using formal methods.

Even though the popularity of formal methods is currently limited to life-critical system development, we can essentially adopt the techniques for use in other software project development. To adopt a formal method does not necessarily mean to use the method in the whole development process. Formal methods can be used partially at certain stages

in the software development life cycle. For non life-critical systems, especially those that have vague problem definitions, it may be beneficial to employ a formal method during the specification stage. This is to prevent the developers from implementing a system before having a clear concept of the system.

2.4.2 The Limitations

Although formal methods offer many benefits, the software industry is still reluctant to adopt formal methods in the software development life cycle. Using a formal method does take more time and cost more, so industries consider employing formal methods to not be cost-effective.

There are a number of additional reasons that inhibit industries from accepting and using formal methods widely:

- Successful software development methods [38].

  Software development methods[1] have evolved and successfully improved software quality. Examples include the waterfall model through to the object-oriented development methods. Peters and Pedrycz [34] provide an extensive literature review of various software development life cycle models.

- Increased costs at the specification stage.

  To specify a formal model requires more time and resources. Designers must have proper training in the use of formal methods. These requirements make formal methods harder to justify in time-to-market software industries.

- Limited applicability to modeling user interfaces [38].

  User interfaces are very important components in most software systems. The lack of ability to model user interfaces using formal methods makes industry hesitant to use formal methods.

[1] These methods refer to those that do not emphasize the use of mathematical notations in specifying a system.

2.4.3 Z: An Example of a Formal Specification Language

Z is a formal specification language that has well-defined syntax and semantics. It is based on mature mathematical concepts and is in the process of international standardization. Hence, Z specifications enjoy the advantage of "universal" syntax and semantics. Using Z, we can combine formal notations and natural language to specify a system. This combination makes the specification rigorous yet understandable.

Using Z, a schema[2] is constructed to specify the structure of a system and its properties. The schema notation is one of the distinguishing features of Z [3]. It enables one to deal with the complexities of a system by abstracting and decomposing the system into small manageable modules.

There are extensive resources on Z (see [41]) that facilitate learning the process of using Z. Hints describing the process of writing a Z specification using a case study are also available [9]. In addition, there exist various tools to check Z specifications ([41]).

Besides the available support tools and the elegant features of Z, the rationale of choosing Z as the specification language in this thesis are: (i) the similar foundation of Z and the relational database model (i.e. set theory), and (ii) the experience gained by the author when doing preliminary work in data warehouses [15].

In the specifications developed, I use the following conventions:

- ?: an input variable
- !: an output variable
- $'$: a post-operation state
- Ξ: no state change
- Δ: exists state change

I illustrate the conventions used with the following examples. Please refer to Appendix A for the description of Z notations used.

[2]There is a possible ambiguity in the use of term "schema" that can refer to a Z schema or a database schema. Therefore, I will use the term "Z schema" every time I refer to a Z schema; unless there is no possible vagueness.

STRING is defined as a basic type. *MESSAGE* is a type used to report the result of an operation.

$$[STRING]$$
$$MESSAGE ::= Success \mid Found \mid Fail$$

Student is a schema that consists of a student number and a name.

$$Student \mathrel{\widehat=} [studentID : STRING;\ name : STRING]$$

MAX is defined as a constant value, to denote the maximum number of students in a class. Constants and functions in Z are represented by the same notation, i.e. the axiomatic definition. Once a constant or a function is defined, we can use the definition throughout the specification.

$$MAX : \mathbb{N}_1$$
$$\hline$$
$$MAX = 50$$

Class consists of a finite set of students, where the number of students is less than or equal to the predefined value *MAX*.

Class
$$allStudents : \mathbb{F}_1\ Student$$
$$\hline$$
$$\#allStudents \leq MAX$$

AddStudent defines an operation to add a new student to the class. The $\Delta Class$ indicates that the *AddStudent* operation changes the *Class*'s state. The *newStudent*? indicates an input variable. The *result*! indicates an output variable. The pre-conditions of this operation are that *newStudent*? is not an existing student, and the number of students in the class is less than *MAX*. If these pre-conditions are satisfied, then the operation adds the new student to the class. The predicate $allStudents' = allStudents \cup \{newStudent?\}$ states that the *allStudents* after the operation ($'$) consists of the union of *allStudents* before the operation and the *newStudent*?.

AddStudent

$$\Delta Class$$
$$newStudent? : Student$$
$$result! : MESSAGE$$

$$newStudent? \notin allStudents$$
$$\#allStudents < MAX$$
$$allStudents' = allStudents \cup \{newStudent?\}$$
$$result! = Success$$

SearchStudent is an operation to search for a student based on an input name.

SearchStudent

$$\Xi Class$$
$$name? : STRING$$
$$result! : MESSAGE$$

$$\exists s : Student \mid s \in allStudents \bullet s.name = name?$$
$$result! = Found$$

The $\Xi Class$ indicates that the *SearchStudent* operation does not change the *Class*'s state.

Chapter 3

Specification of Data Warehouse Components

This chapter presents a formal specification of the data warehouse components discussed in Chapter 2. I discuss the general requirements followed by a formal model of each component. Several UML diagrams are provided as supplements for better understanding of the formal model.

3.1 Data Warehouse Schema

3.1.1 Requirements

The following characteristics are desirable of a data warehouse schema:

- Flexible to cope with the changing business and user requirements.

  Business and user requirements are dynamic over time and therefore are not fully known during the design stage. To avoid costs of restructuring a data warehouse schema, the schema must be adaptable and flexible enough to answer ad hoc queries. Specifically, the schema must be effective in answering the multidimensional queries that are typical in decision support systems.

- Support classical OLAP operations, such as roll-up, drill-down, slice, and dice.

Business analysts need to view a subset of data from various levels of aggregation. Highly summarized information is needed to observe the overall performance of a company. More detailed data may be required for further analyses of the aggregated information. Thus, a data warehouse schema must enable users to perform OLAP operations effectively.

- Ensure data consistency in the data warehouse.

  A data warehouse is populated from heterogeneous data sources. The data sources may have different forms and structures. These inconsistencies must be resolved before the data is loaded into the data warehouse. Thus, the selection of attributes (e.g. primary key) in the data warehouse schema must take into account the structure of data sources.

3.1.2 Specification

This section provides an abstract description of the star and snowflake schema. The specifications are based on the relational data model [10]. I start the specifications by introducing the following user defined types.

$[ATTRIBUTE, DOMAIN]$

$ATTRIBUTE$ is used to represent an attribute of an entity. Every attribute is defined on a certain domain. $DOMAIN$ specifies the data type of an attribute.

$VALUE$ is used to denote a value of an attribute. Semantically, a value could have a type of integer, real, string, etc. However, I define $VALUE$ as numeric just for the specification's type checking purpose.

$$VALUE == \mathbb{Z}$$

A relation schema (intension) defines the structure of a relation (extension). $RELSCHEMA$ is modelled as a partial function from $ATTRIBUTE$ to $DOMAIN$.

$$RELSCHEMA == ATTRIBUTE \nrightarrow DOMAIN$$

For example, a relation schema *Employee* consists of {*(EmpNo, INTEGER), (FirstName, STRING), (LastName, STRING), (Job, STRING)*}. The pair *(EmpNo, INTEGER)* de-

clares that the attribute *EmpNo* is defined on the domain *INTEGER*.

TUPLE represents an instance of a relation schema. *TUPLE* is modelled as a partial function from *ATTRIBUTE* to *VALUE*.

$$TUPLE == ATTRIBUTE \nrightarrow VALUE$$

For example, a tuple t, which is an instance of the relation schema *Employee*, may consist of *{(EmpNo, 6457081), (FirstName, "John"), (LastName, "Stone"), (Job, "Programmer")}*.

In the literature, sometimes the domains in a relation schema and the attributes in a tuple are not written. However, in this thesis I choose to specify them explicitly.

I provide the following function signature to capture properties of a relation. *typeof* is a function that takes a value and returns the value's type (the domain). For example, if we call *typeof("Programmer")*, then the function returns STRING.

$$typeof : VALUE \rightarrow DOMAIN$$

Using the definitions above, a relation (extension) can be abstracted as follows.

Relation

$$columns : RELSCHEMA$$
$$rows : \mathbb{F}\ TUPLE$$

$$columns \in \mathbb{F}_1(ATTRIBUTE \times DOMAIN)$$

$$\forall a : ATTRIBUTE;\ tup : rows \mid a \in \mathrm{dom}\ columns \bullet$$
$$\mathrm{dom}\ tup = \mathrm{dom}\ columns \land typeof(tup(a)) = columns(a)$$

A relation consists of a relation schema and a set of tuples. If we represent a relation as a table, then the relation schema is the heading that forms the columns, while the tuples are the rows of the table. The relation schema determines the structure of the tuples. The value for each attribute in every tuple must conform to the domain of the attribute that is defined on the relation schema. The predicate $columns \in \mathbb{F}_1(ATTRIBUTE \times DOMAIN)$ is used to emphasize that the relation schema consists of a non-empty *finite* set of *(ATTRIBUTE, DOMAIN)* pairs. Figure 3.1 shows an example of a relation with its structure explicitly

columns:
{*(EmpNo, INTEGER), (FirstName, STRING), (LastName, STRING), (Job, STRING)*}
rows:
{ {*(EmpNo, 1), (FirstName, "Mary"), (LastName, "Stone"), (Job, "Programmer")*},
{*(EmpNo, 2), (FirstName, "Peter"), (LastName, "Pan"), (Job, "Analyst")*},
{*(EmpNo, 3), (FirstName, "John"), (LastName, "Hill"), (Job, "Programmer")*},
{*(EmpNo, 4), (FirstName, "Susan"), (LastName, "Grey"), (Job, "Administrator")*} }

Figure 3.1: A relation *Employee*

stated[1]. Table 3.1 illustrates an equivalent relation *Employee* in a table format.

Table 3.1: Employee

EmpNo	FirstName	LastName	Job
1	Mary	Stone	Programmer
2	Peter	Pan	Analyst
3	John	Hill	Programmer
4	Susan	Grey	Administrator

There are various constraints that can be specified in a relation schema. These constraints restrict the data that can be stored in the relation defined on the schema. The following Z schema captures basic integrity constraints (i.e. domain constraints, key constraints, and entity integrity [16]) in a database schema. The basic integrity constraints are defined by specifying the attributes and the domains, the primary key, and the set of attributes that cannot contain *null*. Recall, a primary key must not contain *null*.

RelationSchema

$$attributes : RELSCHEMA$$
$$priKey : \mathbb{F}_1\ ATTRIBUTE$$
$$notNull : \mathbb{F}_1\ ATTRIBUTE$$

$$priKey \subseteq \text{dom}\ attributes$$
$$notNull \subseteq \text{dom}\ attributes$$
$$priKey \subseteq notNull$$

Before the star relation and snowflake relation are specified, the concepts of primary key and foreign key are introduced. I adopt, with slight modifications, the primary and foreign

[1]In the following examples, I will illustrate a relation as a table without explicitly showing the structure.

key specifications presented in [7].

primaryKey is a relation between a finite set of *ATTRIBUTE* and *Relation*, where the set of attributes is a unique identifier of the tuples in the relation. Suppose a finite set of attributes *key* is the primary key of a relation r. For all tuples $tup1$ and $tup2$ in r, $tup1 = tup2$ if, and only if, each value of the key attribute(s) in the tuples is identical.

$$
\begin{array}{|l}
primaryKey : \mathbb{F}_1\, ATTRIBUTE \leftrightarrow Relation \\
\hline
\forall\, key : \mathbb{F}_1\, ATTRIBUTE;\ r : Relation \bullet \\
\quad (key, r) \in primaryKey \Rightarrow \\
\qquad key \subseteq \mathrm{dom}\, r.columns \land \\
\qquad (\forall\, tup1, tup2 : r.rows \bullet \\
\qquad\quad tup1 = tup2 \Leftrightarrow (\forall\, a : key \bullet tup1(a) = tup2(a)))
\end{array}
$$

Null represents an unknown value. In this specification, *null* is introduced as an integer with value 0. However, different domains may have different default values for *null*. Once again, the reason for this choice is to let the specification pass the type checking.

$$
\begin{array}{|l}
null : \mathbb{Z} \\
\hline
null = 0
\end{array}
$$

foreignKey is a relationship between *(ATTRIBUTE, Relation)* pairs. Suppose *fk* and *pk* are attributes in relation $r1$ and $r2$, respectively. If $(fk, r1) \mapsto (pk, r2)$ is a member of *foreignKey*, then *fk* references the primary key *pk*. Thus, *fk* and *pk* must have the same domain, and for every tuple in $r1$, the value of *fk* is either *null* or there exists a corresponding tuple in $r2$ such that the value of *pk* in the corresponding tuple is identical with the value of *fk*[2].

$$
\begin{array}{|l}
foreignKey : (ATTRIBUTE \times Relation) \leftrightarrow (ATTRIBUTE \times Relation) \\
\hline
\forall\, fk, pk : ATTRIBUTE;\ r1, r2 : Relation \bullet \\
\quad (fk, r1) \mapsto (pk, r2) \in foreignKey \Rightarrow \\
\qquad fk \in \mathrm{dom}\, r1.columns \land \\
\qquad r1.columns(fk) = r2.columns(pk) \land \\
\qquad (\{pk\}, r2) \in primaryKey \land \\
\qquad (\forall\, tup1 : r1.rows \bullet tup1(fk) = null \lor \\
\qquad\qquad (\exists\, tup2 : r2.rows \bullet tup2(pk) = tup1(fk)))
\end{array}
$$

Recall that a star relation consists of a fact relation and a set of dimension relations

[2]These constraints are known as *referential integrity* constraints.

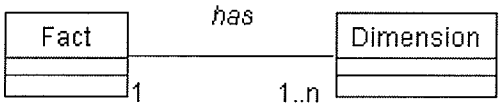

Figure 3.2: Star relation

(Figure 3.2). Every dimension relation has a primary key (pk) that is referenced by some foreign key (fk) in the fact relation. The multiplicity in the UML diagram in Figure 3.2 states that a fact object can be associated with one or more dimension objects.

StarRelation

$$fact : Relation$$
$$dims : \mathbb{F}_1\ Relation$$

$$fact \notin dims$$

$$\forall d : dims \bullet \exists pk, fk : ATTRIBUTE \bullet$$
$$(\{pk\}, d) \in primaryKey \land (fk, fact) \mapsto (pk, d) \in foreignKey$$

The description of a star relation is stored in a star schema. I abstract a star schema as a collection of relation schemas (the fact and dimension schemas) and basic integrity constraints (the constraints that are specified in the *RelationSchema* plus additional referential integrity constraints). In fact, a star schema may contain more information, such as enterprise or application constraints.

StarSchema

$$factSchema : RelationSchema$$
$$dimSchemas : \mathbb{F}_1\ RelationSchema$$
$$forKeys : (ATTRIBUTE \times RelationSchema) \leftrightarrow (ATTRIBUTE \times RelationSchema)$$

$$factSchema \notin dimSchemas$$

$$\forall ds : dimSchemas \bullet \exists pk, fk : ATTRIBUTE \bullet$$
$$\{pk\} = ds.priKey \land fk \in \mathrm{dom}\, factSchema.attributes \land$$
$$(fk, factSchema) \mapsto (pk, ds) \in forKeys$$

Before being materialized with data, all relations in a star relation are empty. The following specification captures the initial condition of a star relation.

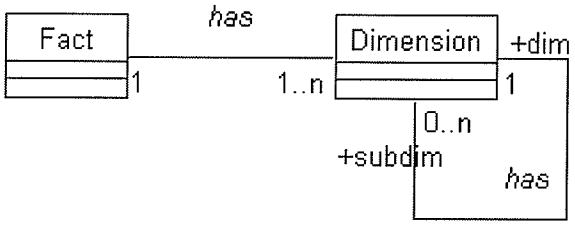

Figure 3.3: Snowflake relation

InitStarRelation

$$StarRelation'$$

$$\forall r : Relation \mid r = fact' \lor r \in dims' \bullet r.rows = \emptyset$$

A snowflake relation has a similar structure to the star relation (Figure 3.3). A fact relation references dimension relations. Furthermore, a dimension relation may reference sub-dimension relations[3]. Normalization in dimension relations generally can reduce the data storage requirements. However, this reduction is typically not significant since a large portion of data storage is occupied by the fact relation [5].

SnowflakeRelation

$$StarRelation$$
$$subdims : \mathbb{F}_1\ Relation$$

$$fact \notin subdims \land dims \cap subdims = \emptyset$$

$$\forall sd : subdims \bullet$$
$$\quad \exists pksd, fkd : ATTRIBUTE;\ d : dims \bullet$$
$$\quad\quad (\{pksd\}, sd) \in primaryKey \land$$
$$\quad\quad (fkd, d) \mapsto (pksd, sd) \in foreignKey$$

The structure of a snowflake relation is kept in a snowflake schema. Similar to the star schema above, a snowflake schema is defined as a set of relation schemas and integrity constraints.

[3]The notations +dim and +subdim in Figure 3.3 denote different roles of dimension instances.

SnowflakeSchema

$$
\begin{array}{l}
StarSchema \\
subdimSchemas : \mathbb{F}_1\ RelationSchema \\
\hline
factSchema \notin subdimSchemas \\
dimSchemas \cap subdimSchemas = \emptyset \\
\\
\forall\, sds : subdimSchemas \bullet \\
\quad \exists\, pksd, fkd : ATTRIBUTE;\ ds : dimSchemas \bullet \\
\qquad \{pksd\} = sds.priKey \land fkd \in \mathrm{dom}\, ds.attributes \land \\
\qquad (fkd, ds) \mapsto (pksd, sds) \in forKeys
\end{array}
$$

The initial state of a snowflake relation is the same as the initial state of a star relation. It starts with empty relations.

InitSnowflakeRelation

$$
\begin{array}{l}
SnowflakeRelation' \\
\hline
\forall\, r : Relation \mid r = fact' \lor r \in dims' \lor r \in subdims' \bullet \\
\quad r.rows = \emptyset
\end{array}
$$

A data warehouse schema now can be defined as either the star schema or snowflake schema. This schema defines the data warehouse states, which I call *DWRelation*.

$$DWSchema ::= starSch\langle\langle StarSchema\rangle\rangle \mid snowSch\langle\langle SnowflakeSchema\rangle\rangle$$

$$DWRelation ::= starRel\langle\langle StarRelation\rangle\rangle \mid snowRel\langle\langle SnowflakeRelation\rangle\rangle$$

3.2 Partitioning

There are several criteria for proper partitioning. This section discusses the correctness rules for partitioning and integrates the rules into the partitioning specification.

3.2.1 Correctness Rules of Partitioning

The following properties are required for correct partitioning operations [32]:

1. **Completeness**: There is no loss of information in the resulting table partitions. Each data item from the original table must be stored in one or more of the resulting table

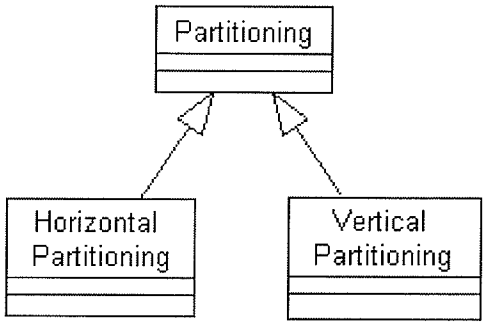

Figure 3.4: Two basic partitioning schemas

partitions. A data item refers to a tuple or an attribute depending on the type of partitioning.

2. **Reconstruction**: It must be possible to reconstruct the original table from resulting table partitions.

3. **Disjointness**: There is no duplication of a data item in table partitions. A tuple that exists in a table partition must not exist in any other table partitions. However, in the case of vertical partitioning, the disjointness property only applies to non-primary key attributes, since the primary key is typically repeated in table partitions to enable the reconstruction of the original table.

Another important requirement for partitioning is that table partitions must be transparent to users. Users should only view a data warehouse as one big table. DBMSs are responsible for maintaining this transparency.

3.2.2 Horizontal and Vertical Partitioning

This section specifies two basic partitioning schemas: horizontal and vertical partitioning (Figure 3.4). The following additional basic types are introduced for the specification.

$[OPERATOR]$
$ATTRorVAL ::= attr\langle\langle ATTRIBUTE\rangle\rangle \mid val\langle\langle VALUE\rangle\rangle$

$OPERATOR$ is a set of comparison operators. It consists of $\{<, \leq, =, \neq, \geq, >\}$. A predicate describes a constraint(s)/criteria used to select a subset of tuples from a relation.

A predicate has the following form [16]:

$$< attribute >< comparison\ operator >< attribute \mid constant\ value >$$

An attribute is compared to another attribute or a constant value. The attribute's type on the left-hand side must be comparable with the other attribute's type/the constant value on the right. For example, a predicate p is specified as:

$$p\text{: Job} = \text{“Programmer”}$$

Formally, a predicate is modelled as follows:

Predicate

$$left : ATTRIBUTE$$
$$cop : OPERATOR$$
$$right : ATTRorVAL$$

SetOfAttributes is defined as a finite set of attributes. This type is used to denote a subset of attributes in a relation for vertical partitioning.

$$SetOfAttributes == \mathbb{F}_1\ ATTRIBUTE$$

To formalize horizontal partitioning, the selection operator σ is defined. The selection operator σ takes a predicate and a relation, and produces a new relation where all tuples in the new relation satisfy the condition(s) defined in the predicate.

$$\sigma : Predicate \times Relation \nrightarrow Relation$$

Following the previous examples, given the *Employee* relation (Table 3.1) and predicate p above, the resulting relation of $\sigma(p, Employee)$ is shown in Table 3.2.

Table 3.2: Job = “Programmer”

EmpNo	FirstName	LastName	Job
1	Mary	Stone	Programmer
3	John	Hill	Programmer

Figure 3.5 shows an example of horizontal partitioning. In this scenario, *Employee*

Job = "Programmer"

EmpNo	FirstName	LastName	Job
1	Mary	Stone	Programmer
3	John	Hill	Programmer

Job = "Analyst"

EmpNo	FirstName	LastName	Job
2	Peter	Pan	Analyst

Job = "Administrator"

EmpNo	FirstName	LastName	Job
4	Susan	Grey	Administrator

Figure 3.5: An example of horizontal partitioning

(Table 3.1) is partitioned horizontally based on job categories, i.e. programmer, analyst, and administrator. The predicates used to partition *Employee* are:

p1: Job = "Programmer"
p2: Job = "Analyst"
p3: Job = "Administrator"

The *union* function takes a finite set of relations and returns a finite set of tuples. The input relations must have the same structure. The set of tuples that is returned by the function is the union of all tuples in the input relations.

$$union : \mathbb{F}\, Relation \nrightarrow \mathbb{F}\, TUPLE$$

$$\begin{aligned}
&\mathrm{dom}\, union = \{rs : \mathbb{F}\, Relation \mid \forall\, r1, r2 : rs \bullet r1.columns = r2.columns\} \\
&\forall\, rs, rs' : \mathbb{F}\, Relation \mid rs \in \mathrm{dom}\, union \bullet \\
&\quad rs = \emptyset \Rightarrow union(rs) = \emptyset \wedge \\
&\quad rs \neq \emptyset \Rightarrow (\exists\, r : rs \bullet \\
&\qquad\qquad rs' = rs \setminus \{r\} \wedge \\
&\qquad\qquad union(rs) = r.rows \cup union(rs'))
\end{aligned}$$

The following Z schema specifies a formal definition of horizontal partitioning (as discussed below).

HorizontalPartitioning

$$
\begin{array}{l}
original? : Relation \\
predicates? : \mathbb{F}_1\ Predicate \\
partitions! : \mathbb{F}_1\ Relation \\
\hline
\#partitions! = \#predicates? \\
\\
\forall p : predicates? \bullet (p, original?) \in \mathrm{dom}\,\sigma \land \\
\qquad \sigma(p, original?) \in partitions! \\
\\
\forall tup1 : original?.rows \bullet \\
\quad \exists_1 tup2 : TUPLE;\ part : partitions! \bullet \\
\qquad tup2 \in part.rows \land tup1 = tup2 \\
\\
partitions! \in \mathrm{dom}\ union \land original?.rows = union(partitions!)
\end{array}
$$

The horizontal partitioning operation takes a relation (*original?*) and a set of predicates (*predicates?*) as inputs. The operation produces a set of relation partitions (*partitions!*). The number of resulting partitions must be the same as the number of input predicates. This implies that all input predicates must be valid. For every input predicate p, the result of the selection operator $\sigma(p, original?)$ must be a member of the output partitions. For each tuple in the original relation, there must exist a unique partition that stores the tuple (complete and disjoint). Since the selection operator does not change the structure of the original relation nor the resulting relations, the union of all tuples in the resulting partitions results in the original relation (reconstruction).

To formalise vertical partitioning, the following join ($\bowtie$) and projection (π) functions are defined. The join function ($\bowtie$) takes a finite set of attributes and two relations, and joins the relations based on certain criteria/predicates[4]. The input set of attributes must be a subset of attributes in the input relations. Attributes in the resulting relation are the union of attributes from input relations. Two tuples from different input relations are joined if they have common values for the input attributes. Formally, $\bowtie$ is defined as follows:

[4]Connolly and Begg [11] discuss various forms of join operations in relational algebra. In this thesis, I provide, however, only a specification of the natural join operation.

{EmpNo, FirstName, LastName}

EmpNo	FirstName	LastName
1	Mary	Stone
2	Peter	Pan
3	John	Hill
4	Susan	Grey

{EmpNo, Job}

EmpNo	Job
1	Programmer
2	Analyst
3	Programmer
4	Administrator

Figure 3.6: An example of vertical partitioning

$$
\begin{array}{|l}
\bowtie : \mathbb{F}_1\, ATTRIBUTE \times \mathbb{F}_1\, Relation \nrightarrow Relation \\
\hline
\mathrm{dom} \bowtie = \{as : \mathbb{F}_1\, ATTRIBUTE;\ rs : \mathbb{F}_1\, Relation \mid \\
\qquad (\forall r : rs \bullet as \subseteq \mathrm{dom}\, r.columns) \land \#rs = 2\} \\
\\
\forall as : \mathbb{F}_1\, ATTRIBUTE;\ rs : \mathbb{F}_1\, Relation;\ result : Relation \mid \\
\quad (as, rs) \in \mathrm{dom} \bowtie \bullet \bowtie (as, rs) = result \Rightarrow \\
\qquad (\exists r1, r2 : rs \mid r1 \neq r2 \bullet \\
\qquad\quad result.columns = r1.columns \cup r2.columns \land \\
\qquad\quad (\forall a : as;\ t : result.rows \bullet \\
\qquad\qquad (\exists t1, t2 : TUPLE \mid t1 \in r1.rows \land t2 \in r2.rows \bullet \\
\qquad\qquad\quad t1(a) = t2(a) \land t = t1 \cup t2)))
\end{array}
$$

The projection operator π takes a finite set of attributes and a relation, and produces a relation where the domain of the resulting relation's columns and rows are restricted by the input attributes. The input attributes must be a subset of the attributes in the input relation.

$$
\begin{array}{|l}
\pi : \mathbb{F}_1\, ATTRIBUTE \times Relation \nrightarrow Relation \\
\hline
\mathrm{dom}\, \pi = \{a : \mathbb{F}_1\, ATTRIBUTE;\ r : Relation \mid a \subseteq \mathrm{dom}\, r.columns\} \\
\\
\forall a : \mathbb{F}_1\, ATTRIBUTE;\ r1, r2 : Relation \mid (a, r1) \in \mathrm{dom}\, \pi \bullet \\
\quad \pi(a, r1) = r2 \Rightarrow \\
\qquad\quad r2.columns = a \lhd r1.columns \land \\
\qquad\quad (\forall tup : r1.rows \bullet (a \lhd tup) \in r2.rows) \land \\
\qquad\quad \#r2.rows = \#r1.rows
\end{array}
$$

Figure 3.6 shows an example of vertical partitioning. *Employee* (Table 3.1) is partitioned vertically into two relations, i.e. one selects the attributes *EmpNo*, *FirstName*, and *LastName*; the other one selects the attributes *EmpNo* and *Job*.

Using the functions above, vertical partitioning can be specified as follows (described below).

VerticalPartitioning

$$
\begin{array}{l}
original? : Relation \\
subAttributes? : \mathbb{F}_1\ SetOfAttributes \\
partitions! : \mathbb{F}_1\ Relation \\
\hline
\#partitions! = \#subAttributes? \\
\{\text{dom}\ original?.columns\} = \bigcup\{subAttributes?\} \\
\\
\forall\, sa : subAttributes? \bullet (sa, original?) \in \text{dom}\,\pi \land \\
\quad \pi(sa, original?) \in partitions! \\
\\
\forall\, r1, r2 : partitions! \bullet \exists\, a : \mathbb{F}_1\ ATTRIBUTE \bullet \\
\quad a = \text{dom}(r1.columns \cap r2.columns) \land \\
\quad (a, r1) \in primaryKey \land (a, r2) \in primaryKey \\
\\
\forall\, part : partitions! \bullet \exists\, pk : \mathbb{F}_1\ ATTRIBUTE \bullet \\
\quad (pk, part) \in primaryKey \land \\
\quad (pk, partitions!) \in \text{dom} \bowtie \land \\
\quad original? = \bowtie (pk, partitions!)
\end{array}
$$

The vertical partitioning operation takes a relation (*original?*) and a set of attribute sets (*subAttributes?*) as inputs, and produces a set of relation partitions (*partitions!*). The number of resulting partitions must be the same as the number of *subAttributes?*. The output partitions are produced by applying the projection operator π to the inputs. Intersections between two output partitions' attributes must be the primary key of the relations (disjoint). The union of all input sub-attributes must be the same as the set of attributes in the original relation (complete). Finally, the join operator must be able to reconstruct the original relation from the output partitions (reconstruction).

3.3 Aggregation

3.3.1 Requirements

The following describes the requirements for aggregation operations:

- Aggregated values must be consistent with the detailed/original data.

Aggregation is performed to increase query performance. However, in decision support systems, the quality of information produced by the systems must not be sacrificed to increase query performance. Thus, aggregation operations must guarantee the consistency and correctness of data.

- Aggregation must be transparent to users.

  DBMSs are responsible for directing users' queries to the appropriate aggregated tables. From the users' point of view, aggregation must not increase the complexity of the system. Hence, aggregation must be hidden from end-users.

- Detailed data must still be accessible in the system.

  Although most queries will access summary data, data warehouse systems must preserve the availability of more detailed data. Detailed data are needed to anticipate new user requirements. Analysts may want to examine detailed data after inspecting general views.

3.3.2 Specification

This section specifies aggregate functions based on [11, 12, 16]. *MINVAL* and *MAXVAL* represent the minimum and maximum value in a domain respectively. *EXCEPTION* represents an exception that may occur in an operation. These constants are defined as numeric to conform to the return type of the aggregate functions specified below. However, in this context, the semantics of these constants are more important than their type definition.

$$MINVAL, MAXVAL, EXCEPTION : VALUE$$

There are five aggregate functions in the SQL2 standard [11]: *COUNT*, *SUM*, *AVG*, *MIN*, and *MAX*. The *SUM* and *AVG* functions only apply to numeric values, while the other functions work properly with non-numeric values as well. However, Z only has a predefined basic type, i.e. integer, where mathematical operators apply. Therefore, I abstract the *MIN* and *MAX* functions to work with integers only.

The *countAll* function takes a relation and simply returns the number of rows in the relation. This is equivalent to the SQL $COUNT(*)$ operator.

$$countAll : Relation \rightarrow \mathbb{N}$$

$$\forall r : Relation \bullet countAll(r) = \#r.rows$$

The *counting* function takes an attribute and a relation, and returns the number of non-*null* values in the specified attribute. The input attribute must be one of the attributes in the input relation. This function is used in the *sum*, *avg*, *minimum*, and *maximum* functions to eliminate any *nulls* in the input attribute. All aggregate functions, except for $COUNT(*)$, operate only on non-*null* values.

$$counting : ATTRIBUTE \times Relation \nrightarrow \mathbb{N}$$

$$\begin{aligned}
&\operatorname{dom} counting = \{a : ATTRIBUTE;\ r : Relation \mid a \in \operatorname{dom} r.columns\} \\
&\forall a : ATTRIBUTE;\ r, r' : Relation \mid (a, r) \in \operatorname{dom} counting \bullet \\
&\quad r'.rows = r.rows \setminus \{tup : r.rows \mid tup(a) = null\} \land \\
&\quad counting(a, r) = \#r'.rows
\end{aligned}$$

The *sum* function takes an attribute and a relation, and returns the total value of all the attribute instances in the relation. Recall that the attribute must have a numeric data type. The specification sums each value of the attribute in each row recursively using the *counting* specification above.

$$sum : ATTRIBUTE \times Relation \nrightarrow \mathbb{Z}$$

$$\begin{aligned}
&\forall a : ATTRIBUTE;\ r, r' : Relation \mid (a, r) \in \operatorname{dom} sum \land (a, r) \in \operatorname{dom} counting \bullet \\
&\quad counting(a, r) = 0 \Rightarrow sum(a, r) = 0 \land \\
&\quad counting(a, r) \neq 0 \Rightarrow (\exists\, tup : r.rows \bullet \\
&\qquad\qquad r'.rows = r.rows \setminus \{tup\} \land \\
&\qquad\qquad sum(a, r) = tup(a) + sum(a, r'))
\end{aligned}$$

The *avg* function takes an attribute and a relation, and calculates the average value of the attribute instances in the relation. If the number of non-*null* values in the input attribute happens to be 0 (or an empty set), then an exception is raised [12]. Otherwise, the function returns the average value[5].

[5]This *avg* function uses the integer division operator *div* because *div* is the only division operator available in Z. In real applications, we must use the appropriate division operator.

$$avg : ATTRIBUTE \times Relation \pfun \mathbb{Z}$$

$$\begin{aligned}
&\forall\, a : ATTRIBUTE;\ r : Relation \mid \\
&\quad (a, r) \in \mathrm{dom}\, avg \land (a, r) \in \mathrm{dom}\, sum \land (a, r) \in \mathrm{dom}\, counting \bullet \\
&\qquad counting(a, r) = 0 \Rightarrow avg(a, r) = EXCEPTION \land \\
&\qquad counting(a, r) \neq 0 \Rightarrow \\
&\qquad\quad avg(a, r) = sum(a, r)\ \mathsf{div}\ counting(a, r)
\end{aligned}$$

The following functions, *minimum* and *maximum*, are equivalent to the SQL *MIN* and *MAX* functions, respectively, except that the SQL functions also work with non-numeric values. These functions take an attribute and a relation, and return the minimum or maximum value of the attribute in the relation, respectively. If the attribute's value in the relation is an empty set, then the *minimum* and *maximum* functions return the minimum or maximum value in the attribute's domain, correspondingly.

$$minimum : ATTRIBUTE \times Relation \pfun \mathbb{Z}$$

$$\mathrm{dom}\, minimum = \{a : ATTRIBUTE;\ r : Relation \mid a \in \mathrm{dom}\, r.columns\}$$

$$\begin{aligned}
&\forall\, a : ATTRIBUTE;\ r : Relation \mid \\
&\quad (a, r) \in \mathrm{dom}\, minimum \land (a, r) \in \mathrm{dom}\, counting \bullet \\
&\qquad counting(a, r) = 0 \Rightarrow minimum(a, r) = MINVAL \land \\
&\qquad counting(a, r) \neq 0 \Rightarrow (\exists\, tup1 : r.rows \bullet \\
&\qquad\quad minimum(a, r) = tup1(a) \Leftrightarrow \\
&\qquad\qquad (\forall\, tup2 : r.rows \bullet tup1(a) \leq tup2(a)))
\end{aligned}$$

$$maximum : ATTRIBUTE \times Relation \pfun \mathbb{Z}$$

$$\mathrm{dom}\, maximum = \{a : ATTRIBUTE;\ r : Relation \mid a \in \mathrm{dom}\, r.columns\}$$

$$\begin{aligned}
&\forall\, a : ATTRIBUTE;\ r : Relation \mid \\
&\quad (a, r) \in \mathrm{dom}\, maximum \land (a, r) \in \mathrm{dom}\, counting \bullet \\
&\qquad counting(a, r) = 0 \Rightarrow maximum(a, r) = MAXVAL \land \\
&\qquad counting(a, r) \neq 0 \Rightarrow (\exists\, tup1 : r.rows \bullet \\
&\qquad\quad maximum(a, r) = tup1(a) \Leftrightarrow \\
&\qquad\qquad (\forall\, tup2 : r.rows \bullet tup1(a) \geq tup2(a)))
\end{aligned}$$

The SQL aggregate functions actually return a relation instead of a value [16]. The aggregate functions are *COUNT*, *COUNTALL*[6], *SUM*, *AVG*, *MIN*, and *MAX*. In this model, *AGGRFUNCTION* is used to pass the input arguments to the actual functions that do the computation.

[6]This is equivalent to $COUNT(*)$.

$$AGGRFUNCTION ::= COUNT \mid COUNTALL \mid SUM \mid AVG \mid MIN \mid MAX$$

The following Z schema abstracts an arbitrary, basic aggregation operation. Such an aggregation operation takes a base relation, an attribute, and an aggregate function as inputs, and produces a summary relation. Each aggregate function returns a relation with a single attribute and a tuple that stores the result of the aggregation. If the input aggregate function is either *COUNT* or *COUNTALL*, then the output relation has an attribute with integer domain, since the output relation contains an integer value. For the other input aggregate functions, the domain is derived from the input relation. For example, if the attribute summed has real number as the domain, then this domain is used in the resulting relation. The resulting value (the row of the summary relation) comes from the aggregate function called.

Aggregation

$$\begin{array}{l}
baseRel?, summary! : Relation \\
attr? : ATTRIBUTE \\
aggrFn? : AGGRFUNCTION \\
\hline
(aggrFn? = COUNT \lor aggrFn? = COUNTALL) \Rightarrow \\
\qquad (\exists\, count : ATTRIBUTE;\ integer : DOMAIN \bullet \\
\qquad\qquad summary!.columns = \{count \mapsto integer\}) \\
(aggrFn? = SUM \lor aggrFn? = AVG \lor aggrFn? = MIN \lor aggrFn? = MAX) \Rightarrow \\
\qquad summary!.columns = \{attr?\} \lhd baseRel?.columns \\
\\
aggrFn? = COUNT \Leftrightarrow (attr?, baseRel?) \in \mathrm{dom}\, counting \land \\
\quad (\textbf{let}\ rs == counting(attr?, baseRel?) \bullet summary!.rows = \{\{attr? \mapsto rs\}\}) \\
aggrFn? = COUNTALL \Leftrightarrow (\textbf{let}\ rs == countAll(baseRel?) \bullet \\
\qquad\qquad summary!.rows = \{\{attr? \mapsto rs\}\}) \\
aggrFn? = SUM \Leftrightarrow (attr?, baseRel?) \in \mathrm{dom}\, sum \land \\
\quad (\textbf{let}\ rs == sum(attr?, baseRel?) \bullet summary!.rows = \{\{attr? \mapsto rs\}\}) \\
aggrFn? = AVG \Leftrightarrow (attr?, baseRel?) \in \mathrm{dom}\, avg \land \\
\quad (\textbf{let}\ rs == avg(attr?, baseRel?) \bullet summary!.rows = \{\{attr? \mapsto rs\}\}) \\
aggrFn? = MIN \Leftrightarrow (attr?, baseRel?) \in \mathrm{dom}\, minimum \land \\
\quad (\textbf{let}\ rs == minimum(attr?, baseRel?) \bullet summary!.rows = \{\{attr? \mapsto rs\}\}) \\
aggrFn? = MAX \Leftrightarrow (attr?, baseRel?) \in \mathrm{dom}\, maximum \land \\
\quad (\textbf{let}\ rs == maximum(attr?, baseRel?) \bullet summary!.rows = \{\{attr? \mapsto rs\}\})
\end{array}$$

In real applications, a summary relation can be derived from several base relations. The aggregate queries are correspondingly more complex. A solid understanding of basic

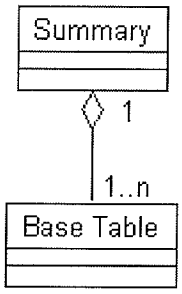

Figure 3.7: Relationship between a summary and base table(s)

aggregate functions is essential to construct such complex queries. Figure 3.7 illustrates the relationship between a summary and base table(s).

3.4 Data Marting

3.4.1 Requirements

In some cases, building a data mart is appropriate. However, there are several principles that must be followed:

- Data marts must be consistent with the enterprise data warehouse.

  Both data marts and the data warehouse must give the same answer to the same query. Any updates in the data warehouse must be propagated to the data marts; as well updates in data marts must be reflected in the data warehouse. This is to keep the data consistent.

- Data marts, naturally, must be able to answer most queries.

  Due to the increased maintenance cost, the decision to build data marts must be justified. The structure of the organization must support data marting. Thus, a data mart in a department must be capable of providing answers to most queries submitted by users in the department. Otherwise, data marting may not be appropriate in that situation.

3.4.2 Specification

A data mart has the same structure as a data warehouse. It consists of a fact relation, a set of dimension relations, and a schema that describes the structure of the data mart. The difference between a data mart and a data warehouse is that a data mart contains only specific data related to a department. A data mart can be obtained by partitioning the data warehouse.

DataMart

$dmSchema : DWSchema$
$dmRelation : DWRelation$

A data mart schema is stored in its metadata just as a data warehouse schema is stored in its metadata. Information in this schema is used to control instances in the data mart.

3.5 Metadata

3.5.1 Requirements

Metadata is commonly known as information about data. Metadata management in a data warehouse is complex, especially due to the lack of a metadata standard. This section tries to identify information that is needed to support use and maintenance of a web-based data warehouse. The aim of this section is to give a high level model of metadata.

I consider metadata in data warehouses to consist of the following main components:

- **Data dictionary**: A data dictionary keeps descriptions of a data warehouse's structures. It includes the data warehouse schemas, data sources, mapping/transformation procedures, aggregation and partitioning schemas, and other information needed to run the data warehouse. SQL2 introduced the concept of an INFORMATION_SCHEMA. INFORMATION_SCHEMA is a special schema that defines the structure of information in a data dictionary [16]. Metadata is also necessary for the DBMS to apply various constraints specified in the data warehouse schema to the data warehouse instances. Furthermore, using information in a data dictionary, a data warehouse

server can perform routine activities such as loading and archiving automatically. A data dictionary is also required by users to understand and use the data warehouse effectively.

- **Business dictionary**: A business dictionary contains all business terms and their descriptions related to the data warehouse applications. For example, if a data warehouse contains revenue information for a company, the formula used to calculate the revenue must be described in the dictionary. This documentation is necessary to avoid ambiguity or misinterpretation.

- **Usage documents**: A web-based data warehouse must provide information on how to use the data warehouse. This information includes the functions provided by the system, steps to perform multidimensional analyses, organisation of the data, and demos/tutorials. In a complex web-based system, providing a web site map that describes the structure of the web site will be helpful as well.

3.5.2 Specification

To specify metadata, I define certain additional basic types (shown below). The use of these basic types will be explained throughout the metadata specification. The model of the data dictionary is adopted from [4].

$$[FUNCTION, INFOSCHEMA, MAPSITE, USERGUIDE, TERM, DESC, STRING]$$

Description stores some information about an attribute. It consists of a unique identifier, an attribute and its type, and a relation where the attribute belongs. This description is used to record information about data sources and their destinations in the data warehouse.

Description

$id : STRING$
$attr : ATTRIBUTE$
$type : DOMAIN$
$rel : Relation$

$attr \mapsto type \in rel.columns$

During the process of loading a data warehouse from the data sources, various transformation functions may be applied to the data. These transformations are intended to resolve any inconsistencies between the data sources and the data warehouse. If the transformation functions are complex, then information about the functions should be kept in the metadata. This information is needed by administrators who maintain the data warehouse.

Transformation contains information about a transformation function. Each transformation schema has a unique identifier to differentiate it from other transformation functions that may have the same name. *FUNCTION* represents information about a procedure/syntax that is applied to one or more data sources.

Transformation

$id : STRING$
$fn : FUNCTION$

To enable a DBMS to direct user queries to the appropriate tables, aggregation schemas and partition schemas of the database must be stored in the metadata. *AggregationSchema* basically consists of a unique id, a base relation, an attribute that is aggregated, and information about the aggregate functions used.

AggregationSchema

$id : STRING$
$baseRel : Relation$
$attr : ATTRIBUTE$
$aggrFn : STRING$

$attr \in \mathrm{dom}\, baseRel.columns$

PartitionSchema stores the necessary information related to partitioning. It consists of a unique id, an original relation, a partition key, and data range in the partition.

PartitionSchema

$id : STRING$
$original : Relation$
$partKey : \mathbb{F}_1\ ATTRIBUTE$
$dataRange : STRING$

$partKey \subseteq \text{dom}\ original.columns$

DataDictionary contains a set of data warehouse schemas, data mappings from sources to destinations - including any transformation functions that are applied to the data sources, aggregation schemas, partition schemas, and INFORMATION_SCHEMA.

$SOURCE == Description$
$DESTINATION == Description$

DataDictionary

$dwschema : \mathbb{F}_1\ DWSchema$
$srcDest : (SOURCE \times DESTINATION) \leftrightarrow Transformation$
$aggrSchema : \mathbb{F}\ AggregationSchema$
$partSchema : \mathbb{F}\ PartitionSchema$
$infoSchema : INFOSCHEMA$

To help end-users analyse information in the data warehouse properly, a business dictionary that contains a list of business terms ($TERM$) and their descriptions ($DESC$) is necessary. A business dictionary ($dict$) is modelled as a partial function from $TERM$ to $DESC$.

BusinessDictionary

$dict : TERM \pfun DESC$

Document contains miscellaneous documents that are needed by end-users. This includes a user's guide and a website map.

Document

$ug : USERGUIDE$
$map : MAPSITE$

Based on the definitions above, metadata in a data warehouse is modelled as consisting of a data dictionary, a business dictionary, and miscellaneous necessary documents (Figure 3.8).

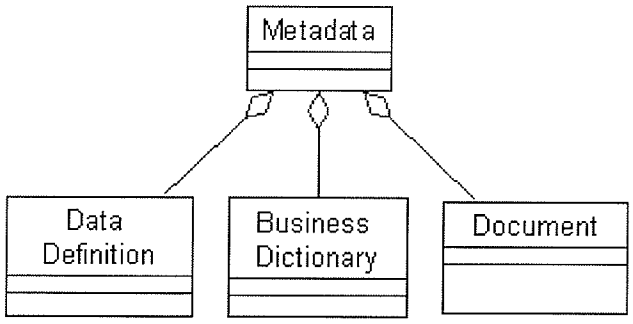

Figure 3.8: Metadata

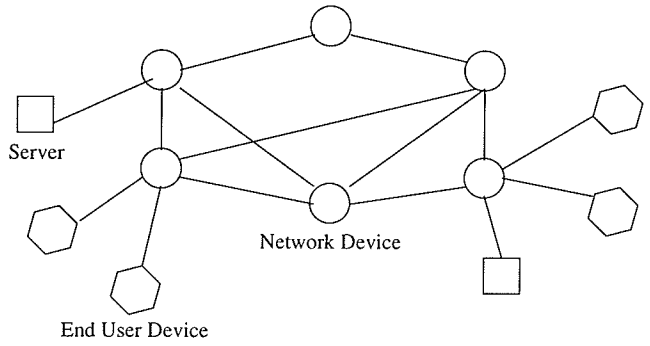

Figure 3.9: A graph representing a network

$$Metadata \mathrel{\widehat{=}} [DataDictionary;\ BusinessDictionary;\ Document]$$

3.6 Functions Supported

3.6.1 The Foundation

This section presents an abstract model of a network, which is the lowest level of infrastructure supporting web-based data warehouses. A network can be represented by a graph (Figure 3.9), where a node in the graph represents an end-user device, a server, or a network device. An edge between two nodes represents the Internet connection. Thus, a network is a collection of nodes that are interconnected with each other using a network protocol running over some physical media.

Each node has a unique Internet Protocol (IP) address in the form of X.X.X.X[7], where X is a byte [22]. Accordingly, there are a limited number of addressable nodes. Let *IPADDRESS* be the type of an IP address, and let *NODETYPE* represent the type of a node.

$[IPADDRESS]$

$NODETYPE ::= EndUser \mid Server \mid NetworkDevice$

A node can be defined as having an IP address and a type. The type of a node can be end-user device, server, or network device. End-user devices refer to all equipments that are connected to the network and are used by end-users (e.g. printers, personal computers). Servers are computers that provide services for the users. Network devices are various devices that support network infrastructure (e.g. routers).

Node

$ip : IPADDRESS$
$type : NODETYPE$

The Internet is abstracted as a collection of nodes and connections. A connection is defined as a relationship between two nodes. This model specifies that there exist at least two nodes in the Internet. Each node has a unique IP address. The Internet connection is symmetric, reflexive and transitive. It is symmetric because a node $n1$ is connected to a node $n2$ if, and only if, $n2$ is connected to $n1$ as well. In addition, every node is connected to itself (reflexive). Furthermore, if a node $n1$ is connected to a node $n2$ and $n2$ is connected to another node $n3$, then $n1$ and $n3$ are also connected (transitive). Therefore, every node in the Internet is connected with one another either directly or indirectly.

[7]There is a new proposed standard for the Internet Protocol called IPv6 or IP Next Generation (IPng) [13] for which this format does not apply. IPv4 is assumed in this thesis.

Internet

$$nodes : \mathbb{F}_1\ Node$$
$$connection : Node \leftrightarrow Node$$

$$\#nodes \geq 2$$
$$\text{dom}\ connection \subseteq nodes \land \text{ran}\ connection \subseteq nodes$$

$$\forall n1, n2 : nodes \bullet (n1.ip = n2.ip \Leftrightarrow n1 = n2) \land$$
$$(n1 \mapsto n2 \in connection \Leftrightarrow n2 \mapsto n1 \in connection) \land$$
$$n1 \mapsto n2 \in connection^*$$

The Internet structure is always changing. Some nodes are added; some are removed. New Internet connections are added, while other ones may be deleted. The following Z schemas specify the dynamic structure of the Internet.

To add a new node (*newNode*?) implies that connections between the new node and an existing node in the network (*linkedTo*?) must also be added. Initially, *newNode*? must not exist in the Internet, while *linkedTo*? must be a part of the Internet. If these pre-conditions are satisfied, then the *AddNode* operation adds *newNode*? and symmetric connections between *newNode*? and *linkedTo*? to the Internet.

AddNode

$$\Delta Internet$$
$$newNode?, linkedTo? : Node$$

$$newNode? \notin nodes$$
$$linkedTo? \in nodes$$

$$nodes' = nodes \cup \{newNode?\}$$
$$connection' = connection \cup \{newNode? \mapsto linkedTo?, linkedTo? \mapsto newNode?\}$$

To remove a node implies that we have to remove all connections that exist between the deleted node and other nodes. The pre-condition of the *RemoveNode* operation is that the deleted node must exist in the Internet. This operation removes all connections that involve the deleted node and the node itself from the network.

RemoveNode

$$\Delta Internet$$
$$rmNode? : Node$$

$$rmNode? \in nodes$$

$$nodes' = nodes \setminus \{rmNode?\}$$
$$connection' = (\{rmNode?\} \lhd connection) \cap (connection \rhd \{rmNode?\})$$

To add a new connection, we have to specify which two nodes we want to connect to each other. The pre-condition of the *AddConnection* operation is that the two input nodes must exist in the network. If the pre-condition is fulfilled, then the operation adds the new connection. Recall that a connection is symmetric, therefore *AddConnection* always adds a pair of symmetric connections.

AddConnection

$$\Delta Internet$$
$$node1?, node2? : Node$$

$$node1? \in nodes \land node2? \in nodes$$

$$connection' = connection \cup \{node1? \mapsto node2?, node2? \mapsto node1?\}$$

To remove a connection that exists between two nodes may exclude one of the nodes from the network. Such exclusion occurs if the connection removed is the only connection that connects that node to the network. If this is the case, then the node must be removed from the network as well. Recall that all nodes in the network must be connected with one another.

RemoveConnection

$$\Delta Internet$$
$$node1?, node2? : Node$$

$$node1? \mapsto node2? \in connection$$

$$connection' = connection \setminus \{node1? \mapsto node2?, node2? \mapsto node1?\}$$
$$node1? \notin \mathrm{dom}\, connection' \Rightarrow nodes' = nodes \setminus \{node1?\}$$
$$node2? \notin \mathrm{dom}\, connection' \Rightarrow nodes' = nodes \setminus \{node2?\}$$

Having specified the Internet as the infrastructure of web-based data warehouses, the following section defines the basic functions provided by web-based data warehouses. These

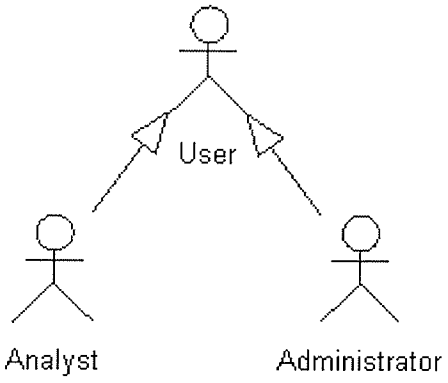

Figure 3.10: Two groups of users

functionalities are built on top of the infrastructure. Thus, if an operation requires the Internet connection, then we assume that the connection exists.

3.6.2 Functional Specification

The following basic types are defined to model the functions of web-based data warehouses. *LoginName* and *Password* are defined as strings. *ACCESSTYPE* consists of different access types in queries. *GROUP* is used to identify a user's group. *BOOLEAN* is a type that consists of two possible values: *True* or *False*.

LoginName == *STRING*
Password == *STRING*
ACCESSTYPE ::= *Select* | *Insert* | *Delete* | *Update*
GROUP ::= *Analyst* | *Administrator*
BOOLEAN ::= *True* | *False*

Generally, users can be classified into two groups, i.e. business analyst or administrator (Figure 3.10). The business analyst group includes executives, managers, and other personnel who are involved in decision-making processes. The administrator group consists of people who are responsible for developing and maintaining the data warehouse system.

Administrators mostly work at the backend. They implement and manage users' accounts, web and data warehouse servers. Most of their tasks rely on the DBMS's functions.

Business analysts are end-users who interact with the user interfaces of the data warehouse system. They submit queries, analyse data, and create various reports. Developers

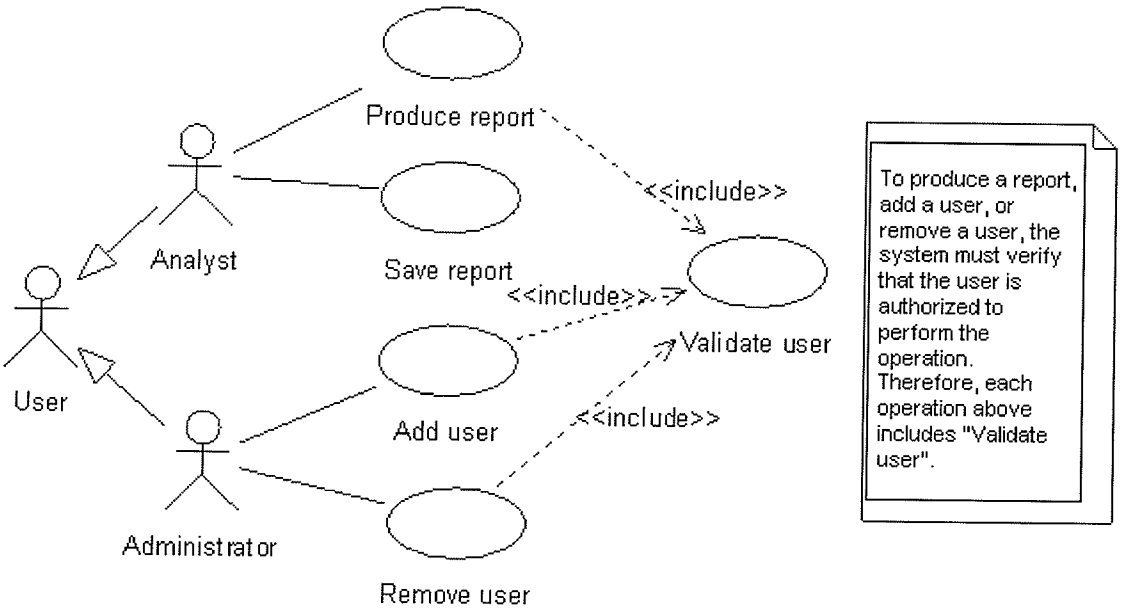

Figure 3.11: Use cases illustrating the uses of a web-based data warehouse

must either implement these requirements or use third party tools to do so.

The main purpose of a web-based data warehouse is to support decision-making processes. Principally, a data warehouse must enable analysts to access and analyse the enterprise data easily. Hence, the system generates various reports based on user queries. Figure 3.11 illustrates various use cases in a data warehouse system.

Producing reports is one of the key functions of a data warehouse system. This process involves several activities (Figure 3.12). Based on the use cases (Figure 3.11) and the activity diagram (Figure 3.12), the following requirements for a web-based data warehouse are specified:

- The system shall support concurrent access by multiple users.

  A web-based data warehouse naturally supports access by multiple users. Authorized users must be allowed to access the system regardless of their locations or time of day.

- The system shall authenticate and identify users.

  Because a data warehouse contains valuable information, a data warehouse system

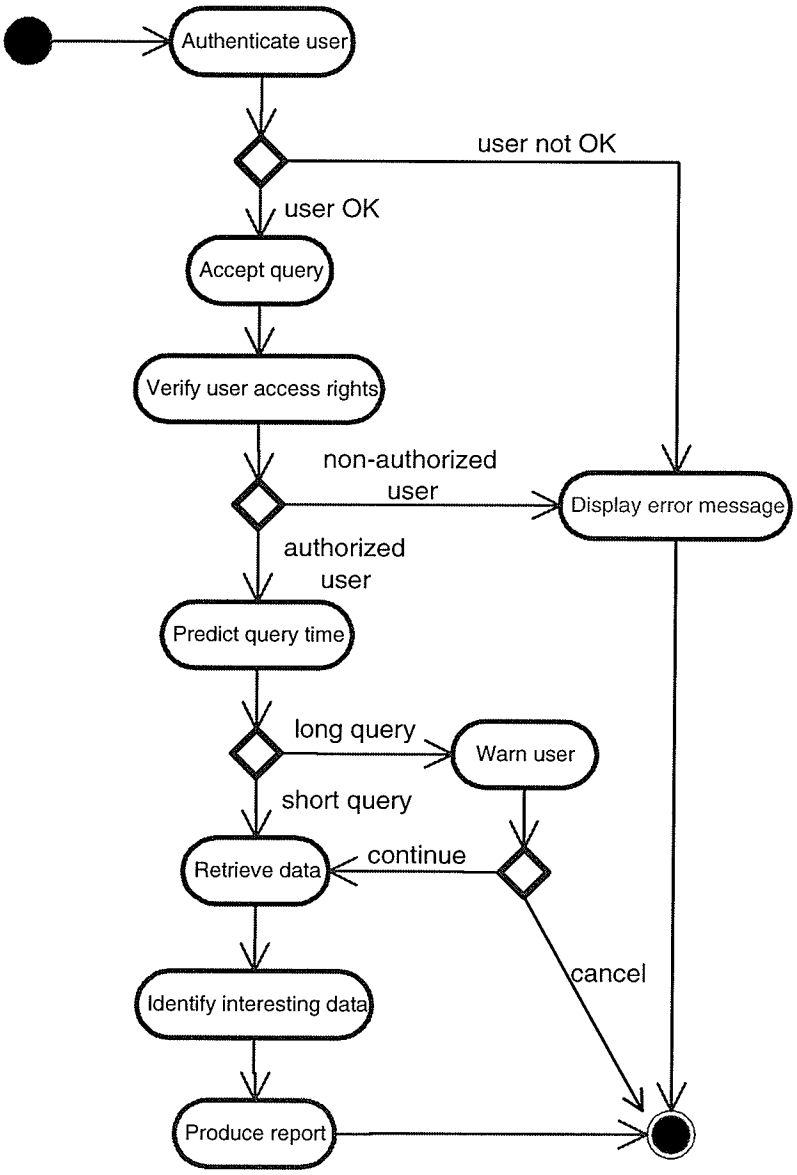

Figure 3.12: Activity diagram for the "Produce report" use case

must ensure that only authorized users can access the system.

- The system shall accept user queries.

  To generate reports, the system has to accept queries from users. The input queries may not necessarily be in an SQL-like format. Instead, the system should provide user-friendly interfaces to help less-technical users to use the data warehouse effectively. Behind the scene, the system must translate the input query into a language that is understandable by the data warehouse server.

- The system shall verify that a user is authorized to access the requested data.

  After identifying the user and accepting the query, the system must verify that the user has the necessary privileges to access the data.

- The system shall warn the user if query processing takes a very long time.

  Some user queries may take a long time to process. If this is the case, the system should warn the user about the long processing time.

- The system shall retrieve data from the database.

  After accepting a user query, the system retrieves the requested data from the database.

- The system shall identify potentially interesting data.

  The system must try to identify any potentially interesting data from the retrieved data. For example, the maximum or minimum value of sales may interest users.

- The system shall present data in an easy-to-read format.

  The system must facilitate user data analysis. To do this, data must be presented in a good format. Commonly used formats are to present data in a table or as a graph.

- The system shall make users aware of any potentially interesting data.

  Any interesting data that have been previously identified must be highlighted to help users notice them. For example, having identified the most popular product in a

certain year, the system should present data on that product in a way that attracts the user's attention.

- The system shall allow users to perform basic multidimensional analyses, such as drill-down, roll-up, slice, and dice.
- The system shall allow users to capture/save reports.

  Saving reports to a disk avoids regenerating the reports during analyses. This can reduce network traffic and processing overhead.
- The system shall provide metadata to enable users to understand the meaning of the data presented.
- The system shall provide various ways to view data.

  This is to allow users to compare different data categories easily.

To perform user authentication, every user must have an account to access the data warehouse system. An account usually consists of a login name and a password. A login name must be unique because it is used to identify a specific user. Every user belongs to a group. Each group has different privileges. For example, the administrator group has an exclusive privilege to manage users' accounts. Session objects are used to track a user's activities thereby compensating the stateless nature of interactions on the web. If a user's session is *True*, then the user is logged onto the system.

Each user is assigned a privilege to access data in the data warehouse. A privilege determines which relations and access types are allowed for the user. All information related to users' access information is stored in the *UserDB*.

UserDB

$$account : LoginName \pfun Password$$
$$group : LoginName \pfun GROUP$$
$$session : LoginName \pfun BOOLEAN$$
$$privilege : LoginName \rel (Relation \times ACCESSTYPE)$$

$$\mathrm{dom}\, group = \mathrm{dom}\, session = \mathrm{dom}\, account$$
$$\mathrm{dom}\, privilege \subseteq \mathrm{dom}\, account$$

The *UserDB* initially has no instances (the database is empty).

InitUserDB

$$UserDB'$$

$$\text{dom } account' = \text{dom } group' = \text{dom } session' = \text{dom } privilege' = \emptyset$$

Login names and passwords stored in the *UserDB* are used for authentication. The authentication process requires a user to enter his/her login name and password. If these inputs match with a valid account, then the user's session is set to *True*. This session's value is used to indicate whether or not the user has passed the authorization procedure.

Authenticate

$$\Delta UserDB$$
$$loginName? : LoginName$$
$$password? : Password$$

$$(loginName? \mapsto password?) \in account$$

$$session' = session \oplus \{loginName? \mapsto True\}$$

A web-based data warehouse system must support simultaneous access from multiple users. Two users are simultaneously accessing the data warehouse if both users' session are *True*.

SimultaneousUser

$$UserDB$$
$$simultaneous : LoginName \leftrightarrow LoginName$$

$$\forall u1, u2 : LoginName \mid (u1, u2) \in simultaneous \bullet$$
$$(u1 \mapsto True) \in session \land (u2 \mapsto True) \in session$$

The following Z schema captures an operation to list active/online users. Active users are those with session values set to *True*.

ListActiveUsers

$$\Xi UserDB$$
$$activeUser! : \mathbb{F}\, LoginName$$

$$activeUser! = \{u : \text{dom } session \mid session(u) = True\}$$

Users can change their passwords. To change a password, a user must be logged onto

the system (session = *True*). The user enters his/her old password, a new password, and a confirmed password. The old password must match with the password in the database. The new and confirmed passwords must be identical. If these inputs are valid, then the system replaces the old password with the new one.

ChangePassword

$\Delta UserDB$
$loginName? : LoginName$
$old?, new?, confirm? : Password$

$(loginName? \mapsto True) \in session$
$(loginName? \mapsto old?) \in account$
$new? = confirm?$

$account' = account \oplus \{loginName? \mapsto new?\}$

To exit the system, a user must currently be logged onto the system. If this pre-condition is satisfied, then the logout operation sets the user's session to *False*. If a user's session is *False*, then the user is unable to perform any operations that require an authorization process.

Logout

$\Delta UserDB$
$loginName? : LoginName$

$(loginName? \mapsto True) \in session$

$session' = session \oplus \{loginName? \mapsto False\}$

The following Z schemas describe administrators' tasks related to account management. To add a user, an administrator has to submit a login name, a password, and a group where the new user belongs. The input login name must not exist in the account. To perform this task, an administrator must be logged onto the system. Furthermore, only users who belong to the administrator group can add new accounts. If these pre-conditions are satisfied, then the system adds the new account into the user database.

AddUser

$\Delta UserDB$
$admin?, new? : LoginName$
$password? : Password$
$group? : GROUP$

$new? \notin \text{dom}\ account$
$(admin? \mapsto True) \in session$
$(admin? \mapsto Administrator) \in group$

$account' = account \oplus \{new? \mapsto password?\}$
$group' = group \oplus \{new? \mapsto group?\}$
$session' = session \oplus \{new? \mapsto False\}$

An administrator also has the privilege to remove users from the system. The *RemoveUser* operation accepts an account and checks if the account exists in the database. Similar to *AddUser*, before performing the operation, the *RemoveUser* checks if the user who submits the request is an administrator and is currently logged onto the system. If these pre-conditions are fulfilled, then the system removes the input account from the user database.

RemoveUser

$\Delta UserDB$
$admin?, old? : LoginName$

$old? \in \text{dom}\ account$
$(admin? \mapsto True) \in session$
$(admin? \mapsto Administrator) \in group$

$account' = \{old?\} \lhd account$
$group' = \{old?\} \lhd group$
$session' = \{old?\} \lhd session$
$privilege' = \{old?\} \lhd privilege$

Users' privileges are assigned by administrators. To add a new access privilege to a user, an administrator must specify which relation and what kind of access type are to be given to the user. Again, the pre-conditions in *AddPrivilege* make sure that the request is valid. If the request is valid, then the operation adds the inputs to *privilege*.

AddPrivilege

$\Delta UserDB$
$admin?, usr? : LoginName$
$r? : Relation$
$at? : ACCESSTYPE$

$usr? \in \text{dom}\ account$
$(admin? \mapsto True) \in session$
$(admin? \mapsto Administrator) \in group$

$privilege' = privilege \cup \{usr? \mapsto (r?, at?)\}$

To specify a process for producing reports, the following additional basic types are introduced. *DATE* represents a date. *GRAPH* and *TABLE* represent types of data presentation formats used in a report's body. *MESSAGE* is used to notify users of some events that occur during the process of producing reports. The resulting data of a query can be presented either as a graph or a table. The format details of how a graph and a table will be drawn, however, are left to the implementers.

$[DATE, GRAPH, TABLE]$
$MESSAGE ::= LongQuery \mid Unauthorized$
$REPORTYPE ::= Table \mid Graph$
$REPORTBODY ::= table\langle\langle TABLE\rangle\rangle \mid graph\langle\langle GRAPH\rangle\rangle$

First, a query needs to be defined. A query can be considered to consist of an access type, a list of attributes, relations, with optional predicates and other constraints (e.g. grouping attribute(s), ordering criteria). The attribute list in a query must be taken from the attributes in the relations specified in the query. A set of attributes in the grouping or ordering criteria, if any, must be a subset of the attribute list in the query. For example, a query to retrieve all programmer's names, ordered by the last names, from the *Employee* relation (Figure 3.1) can be specified as:

```
SELECT FirstName, LastName
FROM Employee
WHERE Job = "Programmer"
ORDER BY LastName
```

In the example above, *SELECT* defines the access type of the query. *FirstName* and *LastName* are the attributes from the *Employee* relation. *Job = "Programmer"* is the

predicate that must be satisfied by all tuples in the resulting relation. *LastName* specifies the ordering criterion.

Formally, a query is defined as follows:

Query

$$
\begin{array}{l}
accessType : ACCESSTYPE \\
attributes : \mathbb{F}_1\ ATTRIBUTE \\
relations : \mathbb{F}_1\ Relation \\
predicates : \mathbb{F}\ Predicate \\
groupBy, orderBy : \mathbb{F}\ ATTRIBUTE \\
\hline
groupBy \subseteq attributes \\
orderBy \subseteq attributes \\
\\
\forall\, a : attributes \bullet \exists\, r : relations \bullet a \in \mathrm{dom}\, r.columns
\end{array}
$$

A report itself can be modelled as having a date, a title, a producer, and a body. The date records when the report is created, the title provides information to the user about what the report concerns, the producer is the user who creates the report, and the body contains data that is represented either as a table or a graph.

Report

$$
\begin{array}{l}
date : DATE \\
title : STRING \\
producer : LoginName \\
body : REPORTBODY
\end{array}
$$

To produce a report, the system must first accept a query from a user. To submit a query, the user must be logged onto the system.

AcceptQuery

$$
\begin{array}{l}
\Xi UserDB \\
usr? : LoginName \\
q? : Query \\
\hline
(usr? \mapsto True) \in session
\end{array}
$$

Based on the input query, the system determines if the user is authorized to access the data requested in the query. An authorized user must have the privilege to access all relations specified in the query. If there exists a relation in the query where the user does not have the privilege to access the relation, then the system must not continue to process

the query and must report the error to the user.

AuthorizedUser

$$\Xi UserDB$$
$$usr? : LoginName$$
$$q? : Query$$
$$\forall r : q?.relations \bullet (usr? \mapsto (r, q?.accessType)) \in privilege$$

UnauthorizedUser

$$\Xi UserDB$$
$$usr? : LoginName$$
$$q? : Query$$
$$\exists r : q?.relations \bullet (usr? \mapsto (r, q?.accessType)) \notin privilege$$

ErrorMessage

$$m! : MESSAGE$$
$$m! = Unauthorized$$

The function *predict* takes a query and returns the predicted time needed to process the query. I provide its signature as follows:

$$predict : Query \rightarrow \mathbb{N}_1$$

LONGQUERY is a predefined value to determine if the data warehouse system needs to warn a user regarding a long query processing time.

$$LONGQUERY : \mathbb{N}_1$$

If the predicted query processing time is greater than or equal to *LONGQUERY*, then the system warns the user that the query will take a long time to process. Otherwise, the query is considered to be a short query and the system simply continues to process the query.

WarnUser

$$q? : Query$$
$$m! : MESSAGE$$
$$predict(q?) \geq LONGQUERY \Rightarrow m! = LongQuery$$

ShortQuery

$$q? : Query$$

$$predict(q?) < LONGQUERY$$

The relation *satisfy* stores ($TUPLE$, $\mathbb{F}\, Predicate$) pairs, where the tuple fulfills the condition(s) specified in the predicate(s).

$$satisfy : TUPLE \leftrightarrow \mathbb{F}\, Predicate$$

$$\forall t : TUPLE;\ ps : \mathbb{F}\, Predicate \bullet$$
$$(t, ps) \in satisfy \Rightarrow$$
$$((\forall p : ps;\ a : ATTRIBUTE \mid a = p.left \vee attr(a) = p.right \bullet$$
$$a \in \operatorname{dom} t) \vee ps = \emptyset)$$

RetrieveData is an operation that takes a query, and produces a relation. The resulting relation has the attributes specified in the query, and for all tuples in the output relation, the tuples must satisfy the predicate(s) in the input query.

RetrieveData

$$q? : Query$$
$$r! : Relation$$

$$\operatorname{dom} r!.columns = q?.attributes$$

$$\forall tup : r!.rows \bullet (tup, q?.predicates) \in satisfy$$

To help users be aware of potentially interesting data, the system should try to identify such data. "Interesting" data can vary depending on the user's interests. In the following Z schema, I assume that users may be interested in the maximum and minimum value of attributes in a relation. Therefore, any tuples that contain such values are labelled as interesting data. These interesting data should be highlighted in the resulting report.

IdentifyInterestingData

$$r? : Relation$$
$$ts! : \mathbb{F}\, TUPLE$$

$$ts! \subseteq r?.rows$$

$$\forall tup : ts! \bullet \exists a : \operatorname{dom} r?.columns \mid$$
$$(a, r?) \in \operatorname{dom} maximum \wedge (a, r?) \in \operatorname{dom} minimum \bullet$$
$$tup(a) = maximum(a, r?) \vee tup(a) = minimum(a, r?)$$

The functions *generateTable* and *generateGraph* take a relation and a finite set of tuples, and generate a report body based on the inputs. The input set of tuples is the set of interesting data that must be highlighted to attract the user's attention. The implementation details of these functions are up to the developers. Therefore, I only provide the function signatures.

$$
\begin{array}{|l}
generateTable : Relation \times \mathbb{F}\, TUPLE \pfun TABLE \\
generateGraph : Relation \times \mathbb{F}\, TUPLE \pfun GRAPH
\end{array}
$$

The following Z schema describes the process of producing a report in a data warehouse system.

GenerateReport

$$
\begin{array}{|l}
date? : DATE \\
title? : STRING \\
usr? : LoginName \\
type? : REPORTTYPE \\
r? : Relation \\
ts? : \mathbb{F}\, TUPLE \\
report! : Report \\
\hline
(r?, ts?) \in \mathrm{dom}\, generateTable \\
(r?, ts?) \in \mathrm{dom}\, generateGraph \\
\\
report!.date = date? \\
report!.title = title? \\
report!.producer = usr? \\
\\
type? = Table \Rightarrow report!.body = table(generateTable(r?, ts?)) \\
type? = Graph \Rightarrow report!.body = graph(generateGraph(r?, ts?))
\end{array}
$$

The system must get necessary data, such as a date, a title, a producer, a report type, a relation, and a set of interesting tuples, to generate a report. The date, the title, and the producer of the resulting report are set based on the inputs. Depending on the report type, the appropriate function to generate the report body is called. The input relation and the set of interesting tuples are the data to be presented in the report.

The whole process of producing a report is represented in *ProduceReport*.

$$
\begin{aligned}
ProduceReport \mathrel{\widehat=} ((&AcceptQuery \land AuthorizedUser \land \\
&(WarnUser \lor ShortQuery) \land RetrieveData) \fatsemi \\
&IdentifyInterestingData[r!/r?] \fatsemi GenerateReport[r!/r?, ts!/ts?]) \\
&\lor \\
&(AcceptQuery \land UnauthorizedUser \land ErrorMessage)
\end{aligned}
$$

There are two possible scenarios. One is *AcceptQuery* and *AuthorizedUser* and (*WarnUser* or *ShortQuery*) and *RetrieveData*, followed by *IdentifyInterestingData* where the input relation is the output relation from *RetrieveData*, then the outputs of *IdentifyInterestingData* are passed to *GenerateReport* that completes the process. The other scenario happens when the user who submits the query is an unauthorized user. If such unauthorized access occurs, then the system displays an error message.

The *ProduceReport* schema also represents basic OLAP operations such as drill-down, roll-up, slice, and dice, because these OLAP operations are basically different queries that are submitted by the user to produce different reports.

Before an operation to save reports is specified, we need to model a file system where the reports can be saved. A Z specification of a file system is available in Barden, *et. al.* [6]. This model is adopted for the specification.

A file system is modelled as a *tree* that consists of a non-empty set of nodes. Each node in the tree, except for the *root*, has a "parent" node. The set of nodes that have no descendents are called *leaf* nodes. The relation between a "child" node and its "parent" is modelled as a partial function called *parent*. The inverse function of *parent* is a relation called *child*. The *child* contains pairs of a "parent" node and its "child".

This model separates the structure and contents of the file system. The structure of the file system is encapsulated in *Tree*; the contents are described by functions in *FileSystem*0 (see below). Each node is either a file or a directory, and must be named. A node's name in a directory is unique. The *name* function takes a node and returns the node's name. The *data* function takes a node and returns the data in the node[8]. An additional function called *namedNode* is defined in *FileSystem*1. The *namedNode* function takes a sequence of names (path) and returns the node that is referred to by the path. For completeness, I include the

[8]A node that contains data is a file; hence it must be a leaf node.

file system specification by Barden, *et. al.* [6][9].

$$digraph[X] == \{v : \mathbb{P}\, X;\ e : X \leftrightarrow X \mid (\operatorname{dom} e \cup \operatorname{ran} e) \subseteq v\}$$
$$dag[X] == \{v : \mathbb{P}\, X;\ e : X \leftrightarrow X \mid (v, e) \in digraph[X] \land \operatorname{disjoint} \langle e^+, \operatorname{id} X\rangle\}$$
$$condag[X] == \{v : \mathbb{P}\, X;\ e : X \leftrightarrow X \mid (v, e) \in dag[X] \land (e \cup e^\sim)^* = v \times v\}$$
$$tree[X] == \{v : \mathbb{P}\, X;\ e : X \leftrightarrow X \mid (v, e) \in condag[X] \land e \in X \nrightarrow X\}$$

$$[NAME, NODE]$$
$$DATA ::= rep\langle\langle Report\rangle\rangle$$

Tree[*N*]

$$t : tree[N]$$
$$parent : N \nrightarrow N$$
$$node, leaf : \mathbb{P}_1\, N$$
$$child : N \leftrightarrow N$$
$$root : N$$

$$(node, parent) = t$$
$$leaf = node \setminus \operatorname{ran} parent$$
$$\{root\} = node \setminus \operatorname{dom} parent$$
$$child = parent^\sim$$

*FileSystem*0

$$Tree[NODE]$$
$$name : NODE \nrightarrow NAME$$
$$data : NODE \nrightarrow DATA$$

$$\operatorname{dom} name = node$$
$$root \notin \operatorname{dom} data \subseteq leaf$$
$$\forall n : node \bullet (\forall c, d : child(\!|\ \{n\}\ |\!) \mid c \neq d \bullet name(c) \neq name(d))$$

*FileSystem*1

$$FileSystem0$$
$$namedNode : \operatorname{seq}_1 NAME \nrightarrow NODE$$

$$namedNode =$$
$$\{path : \operatorname{seq}_1 NODE \mid head\ path = root$$
$$\land (\forall i, j : \operatorname{dom} path \mid j = i + 1$$
$$\bullet\ path\ i \mapsto path\ j \in child)$$
$$\bullet\ path \fatsemi name \mapsto last\ path\}$$

[9] In the original specification, *DATA* is introduced as a given set. I change this to fit the data type in my specification.

$$path == \{p : \mathrm{seq}\, NAME \mid \#p \geq 2\}$$

Let *write* be a function that takes a report and writes the report to a node (thus, the resulting node contains the input report).

$$write : Report \rightarrow NODE$$

Using the specification above, I abstract a "save-report" operation as follows. *SaveReport* accepts a report, a file name (including the complete path from the root), and produces a node that contains the report.

SaveReport

$\Delta FileSystem1$
$report? : Report$
$name? : path$
$file! : NODE$

$front\ name? \in \mathrm{dom}\ namedNode$
$namedNode(front\ name?) \notin \mathrm{dom}\ data$
$name? \notin \mathrm{dom}\ namedNode$

$file! = write(report?)$
$root' = root$
$name' = name \oplus \{file! \mapsto last\ name?\}$
$data' = data \oplus \{file! \mapsto rep(report?)\}$
$namedNode' = namedNode \oplus \{name? \mapsto file!\}$
$parent' = parent \oplus \{file! \mapsto namedNode(front\ name?)\}$

The pre-conditions of the operation state that:

- the input path must be valid, i.e. the *front* of the sequence of names *name*? must exist and refer to a directory.
- the file name does not exist in the directory.

If these pre-conditions are satisfied, then the operation saves the report to a file and performs the necessary updates to the structure and contents of the file system.

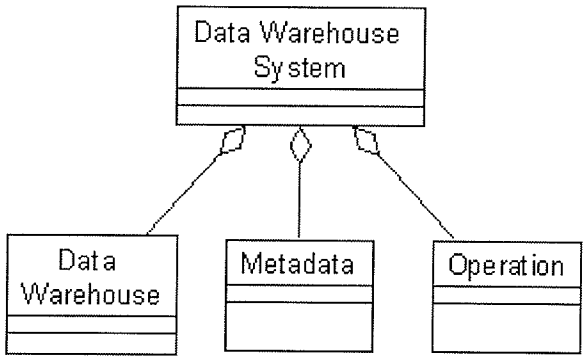

Figure 3.13: Data warehouse system

3.7 Putting It All Together

A web-based data warehouse system consists of a set of data warehouse relations, metadata, and a set of operations. Figure 3.13 illustrates this abstraction.

The warehouse functions presented in this thesis can be categorized into different classes, i.e., *SystemFunction*, *UserFunction*, *AnalystFunction*, and *AdminFunction*. These classes are then integrated into the *Operation* schema. This set of operations, together with the data warehouse relations, and metadata constitute a data warehouse system.

$$SystemFunction \mathrel{\widehat{=}} [InitStarRelation;\ InitSnowflakeRelation;\ HorizontalPartitioning;\ VerticalPartitioning;\ Aggregation;\ InitUserDB]$$

$$UserFunction \mathrel{\widehat{=}} [Authenticate;\ ListActiveUsers;\ ChangePassword;\ Logout]$$

$$AnalystFunction \mathrel{\widehat{=}} [ProduceReport;\ SaveReport]$$

$$AdminFunction \mathrel{\widehat{=}} [AddUser;\ RemoveUser;\ AddPrivilege]$$

$$Operation \mathrel{\widehat{=}} [SystemFunction;\ UserFunction;\ AnalystFunction;\ AdminFunction]$$

$$DataWarehouseSystem \mathrel{\widehat{=}} [dw : DWRelation;\ Metadata;\ Operation]$$

Besides the basic functions specified in this chapter, a web-based data warehouse cer-

tainly requires additional functions based on the business domain or purpose of the data warehouse. These functions vary from one organization to another organization; hence they are out of scope of this thesis. However, the specification in this thesis can be used as a basis for building a complete web-based data warehouse system for a specific business entity.

Chapter 4

Validation

This chapter describes the validation process for the specifications presented in Chapter 3. I present the procedure for syntax, type, and domain checking, and then pose several desired properties of the system as theorems, and prove that the specifications have captured the requirements correctly.

4.1 Syntax and Type Checking

In the sense of formalism, the only mandatory checking for Z specifications is syntax and type checking [37]. Syntax checking ensures that a specification has no syntax errors while type checking ensures that types are used properly in the specification. Examples of type errors are assigning an integer variable to a string, or dividing a number by a string.

Z/EVES [29] has the ability to perform syntax and type checking mechanically. We pass a file containing a Z specification to Z/EVES by invoking the Z/EVES "read" command. The "read" command causes Z/EVES to perform syntax and type checking on the input file, to print the declarations of any types, schemas or functions in the specification, and to produce error messages, if necessary. A specification is type and syntactically correct if the output of the "read" command has no error messages. Figure 4.1 shows partial results of syntax and type checking of the specifications in this thesis.

One can write a specification that is type and syntactically correct, but meaningless. For

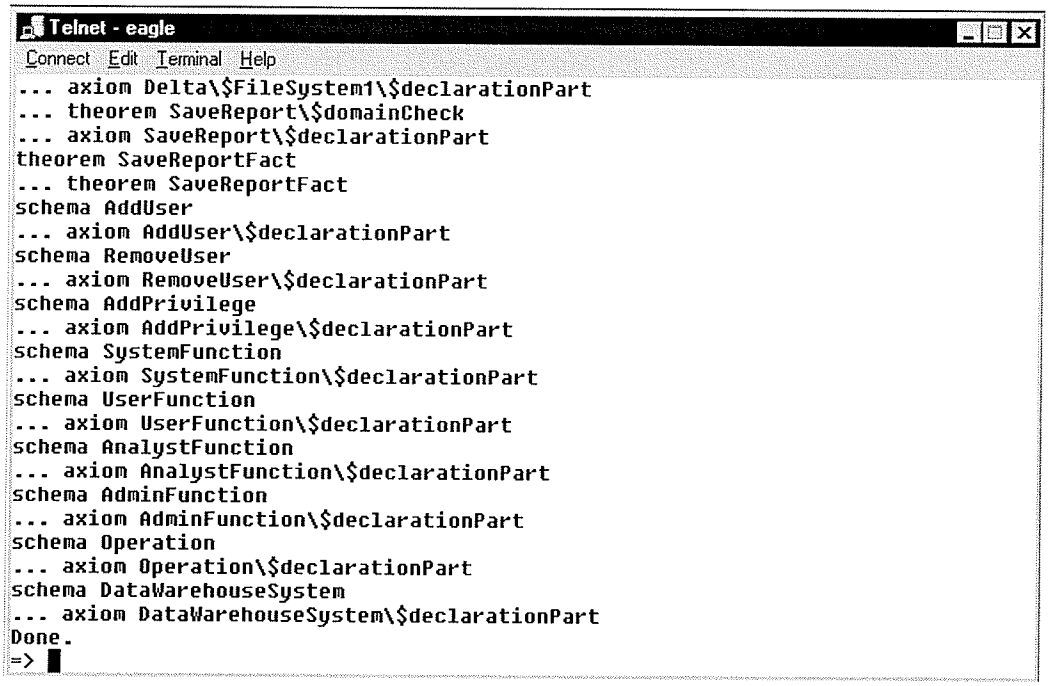

Figure 4.1: A screen shot displaying partial results of syntax and type checking

example, a zero division is a meaningless expression because the result is outside its domain [37]. Therefore, domain checking is necessary to ensure that a specification is meaningful.

4.2 Domain Checking

While checking a specification, Z/EVES automatically generates a list of non-trivial domain checks. The Z/EVES "print status" command prints out a list of the domain checks. Based on the list, we can write a proof script to check if the domain checks are true. Figure 4.2 shows the list of non-trivial domain checks generated by Z/EVES from the specifications in Chapter 3.

To increase confidence in the specification, I write and submit a proof script for several domain checks to Z/EVES. Figure 4.3 shows the result of domain checks after Z/EVES execute the proof script.

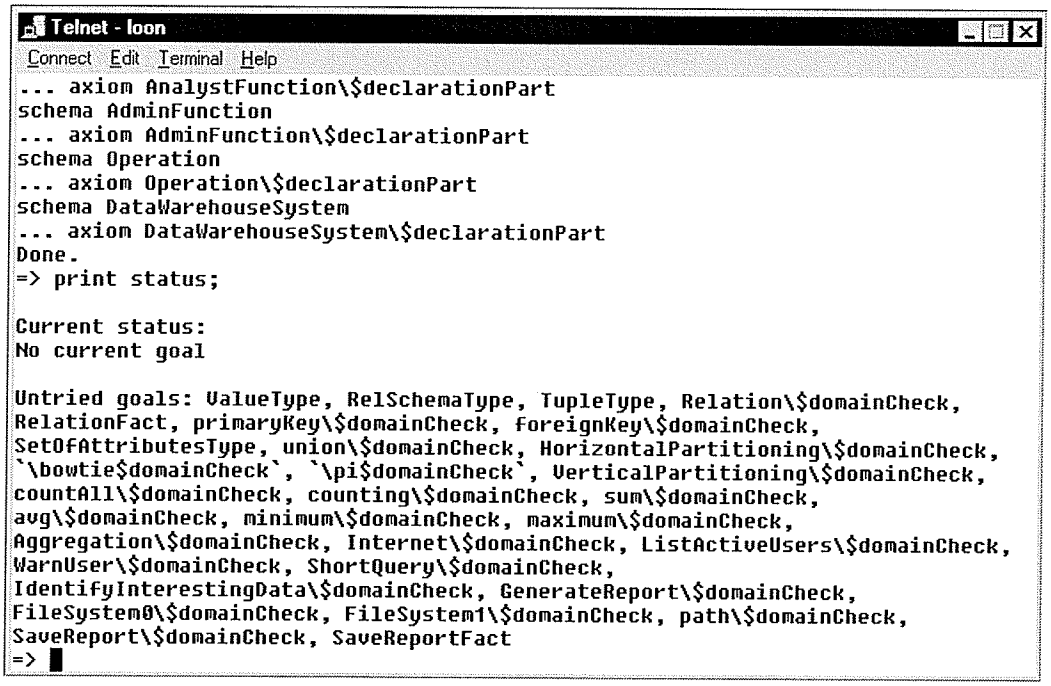

Figure 4.2: A list of non-trivial domain checks

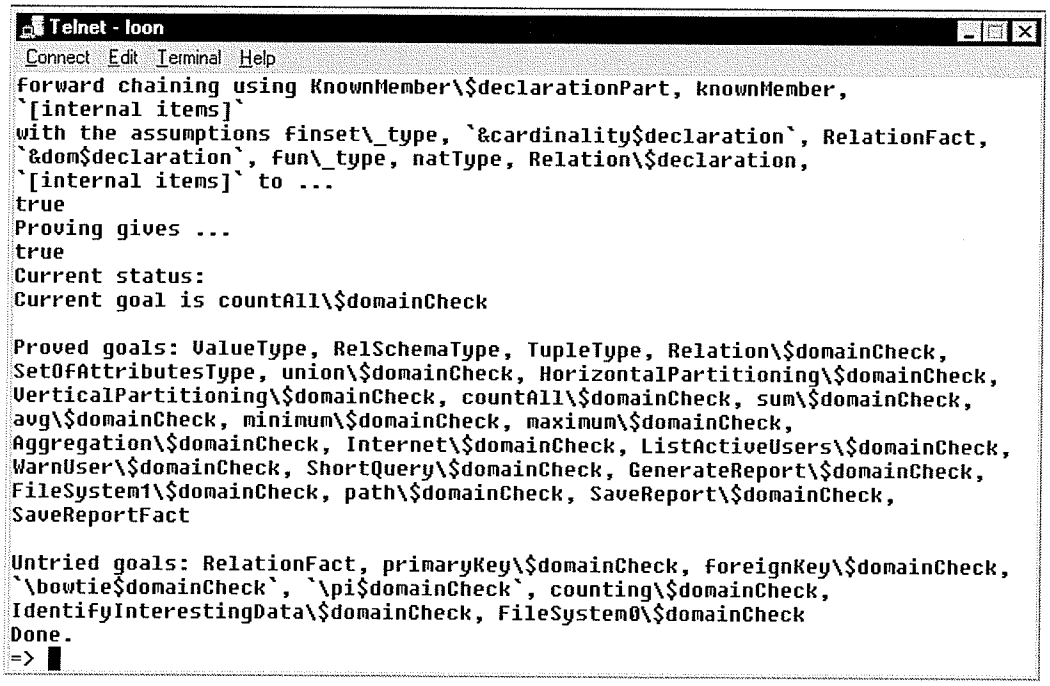

Figure 4.3: The result of domain checks

4.3 Requirements Validation

This section provides various theorems and proofs of the theorems. The theorems describe desirable properties of a web-based data warehouse. The proofs show that the properties have been captured properly in the specification.

Theorem 1 *A relation schema (intension) determines the structure of a relation (extension) that is defined on the relation schema.*

Proof: A relation schema defines a set of attributes and their domains. A domain of an attribute determines the set of permissible values in the attribute. A relation schema forms the columns of a relation. To prove that a relation schema defines the structure of a relation, we need to show that:

i. All tuples in the relation have the same attributes that are defined on the relation schema.

ii. All values of the attributes in the tuples conform to the data types defined on the relation schema.

Let rs and R be an arbitrarily chosen relation schema and a relation, where

$$R.columns = rs$$

From the invariant state of *Relation*

$$\forall\, a : ATTRIBUTE;\ tup : rows \mid a \in \mathrm{dom}\, columns \bullet \\ \mathrm{dom}\, tup = \mathrm{dom}\, columns \wedge typeof(tup(a)) = columns(a)$$

we can derive that the domain of all tuples in R is the same as the domain of relation schema rs $(R.columns)$.

$$\mathrm{dom}\, tup = \mathrm{dom}\, R.columns, \text{ where } tup \in R.rows$$

The domain of $R.rows$ and $R.columns$ is the set of attributes in relation R. $R.rows$ contains all tuples in R. Thus, all tuples in R have the same attributes that are defined in the relation schema rs $(R.columns)$.

From the second conjunct of the invariant state *Relation* above, we can derive that for each attribute a and tuple tup in R, the type of the attribute's value in the tuple $tup(a)$ is the same as the type of attribute a defined in rs ($R.columns$).

$$typeof(tup(a)) = R.columns(a), \text{ where } tup \in R.rows \text{ and } a \in \operatorname{dom} R.columns$$

This completes the proof. □

Theorem 2 *A primary key of a relation is a unique identifier of the tuples in the relation.*

Proof: This theorem can be restated as: two tuples are identical if, and only if, the values of the primary key in the tuples are the same.

Let pk and R be an arbitrarily chosen primary key and a relation, where

$$(pk, R) \in primaryKey$$

From the invariant state of *primaryKey*

$$\begin{array}{l}(key, r) \in primaryKey \Rightarrow \\ \quad key \subseteq \operatorname{dom} r.columns \wedge \\ \quad (\forall\, tup1, tup2 : r.rows \bullet \\ \qquad tup1 = tup2 \Leftrightarrow (\forall\, a : key \bullet tup1(a) = tup2(a)))\end{array}$$

we can derive that for all tuples in R, the tuples are identical if, and only if, the primary key of the tuples refers to the same value.

$$\begin{array}{l}tup1 = tup2 \Leftrightarrow (\forall\, a : pk \bullet tup1(a) = tup2(a)), \text{ where} \\ \quad tup1 \in R.rows \text{ and } tup2 \in R.rows\end{array}$$

Thus, pk uniquely identifies all tuples in R. □

Theorem 3 *An attribute that references another attribute obeys the referential integrity constraint.*

Proof: The referential integrity constraints are captured in *foreignKey*.

Suppose *forKey*, *priKey*, $R1$, and $R2$ are arbitrarily chosen foreign key, *forKey*, in relation, $R1$, that references primary key, *priKey*, in relation, $R2$. Thus,

$$(forKey, R1) \mapsto (priKey, R2) \in foreignKey$$

To prove this theorem, we need to show that:

i. *forKey* and *priKey* have the same domain.

ii. for each tuple in $R1$, the value of attribute *forKey* is either null, or there exists a corresponding tuple in $R2$ that has the same value as *forKey*.

From the invariant state of *foreignKey*

$$\begin{aligned}&(fk, r1) \mapsto (pk, r2) \in foreignKey \Rightarrow \\ &\quad fk \in \operatorname{dom} r1.columns \wedge \\ &\quad r1.columns(fk) = r2.columns(pk) \wedge \\ &\quad (\{pk\}, r2) \in primaryKey \wedge \\ &\quad (\forall\, tup1 : r1.rows \bullet tup1(fk) = null \vee \\ &\qquad (\exists\, tup2 : r2.rows \bullet tup2(pk) = tup1(fk)))\end{aligned}$$

we can derive that *forKey* and *priKey* have the same domain.

$$R1.columns(forKey) = R2.columns(priKey)$$

Furthermore, for all tuples in $R1$, the value of *forKey* is either null or there exists a corresponding tuple in $R2$.

$$\begin{aligned}&(\forall\, tup1 : R1.rows \bullet tup1(forKey) = null \vee \\ &\qquad (\exists\, tup2 : R2.rows \bullet tup2(priKey) = tup1(forKey)))\end{aligned}$$

□

Theorem 4 *Partitioning operations obey the three correctness rules for partitioning, i.e. completeness, reconstruction, and disjointness.*

Proof: To prove the theorem, we need to show that the specification of horizontal and vertical partitioning has properly followed the correctness rules of partitioning.

Case 1: Horizontal Partitioning

Let *org*, *pre*, and *par* be arbitrarily chosen original relation, *org*, a set of predicates, *pre*, and a set of resulting partitions, *par*, respectively.

From the invariant state of *HorizontalPartitioning*

$$\forall\, p : predicates? \bullet (p, original?) \in \operatorname{dom} \sigma \wedge \sigma(p, original?) \in partitions!$$

we can derive that *par* contains all resulting partitions, given the original table *org* and a set of predicates *pre*.

$$\forall p : pre \bullet (p, org) \in \mathrm{dom}\,\sigma \land \sigma(p, org) \in par$$

Let t be an arbitrarily chosen tuple in the original table org.

$$t \in org.rows$$

From the third conjunct of predicates in *HorizontalPartitioning*

$$\begin{array}{l} \forall tup1 : original?.rows \bullet \\ \quad \exists_1 tup2 : TUPLE;\ part : partitions! \bullet \\ \qquad tup2 \in part.rows \land tup1 = tup2 \end{array}$$

we can derive that there exists a unique tuple $tup2$ in a partition $part$ where $tup2 = t$ (completeness).

$$\exists_1 tup2 : TUPLE;\ part : par \bullet tup2 \in part.rows \land tup2 = t$$

In other words, t exists only in $part$ and there is no other partition that contains t as one of its tuples (disjointness).

Finally, from the last predicate in *HorizontalPartitioning*

$$original?.rows = union(partitions!)$$

we can derive that the original relation org can be reconstructed by taking the union of all tuples in par.

$$org.rows = union(par)$$

Case 2: Vertical Partitioning

Let org, att, and par be arbitrarily chosen original relation, org, a set of attribute sets, att, and a set of resulting partitions, par, respectively.

From the invariant state of *VerticalPartitioning*

$$\begin{array}{l} \{\mathrm{dom}\ original?.columns\} = \bigcup\{subAttributes?\} \\ \forall sa : subAttributes? \bullet (sa, original?) \in \mathrm{dom}\,\pi \land \\ \quad \pi(sa, original?) \in partitions! \end{array}$$

we can derive that par contains all resulting partitions, given the original relation and a set of attribute sets att. Furthermore, the union of attributes in all partitions is equal to the complete set of attributes in the original relation (completeness).

$$\begin{array}{l} \{\mathrm{dom}\ org.columns\} = \bigcup\{att\} \\ \forall sa : att \bullet (sa, org) \in \mathrm{dom}\,\pi \land \pi(sa, org) \in par \end{array}$$

Let $R1$ and $R2$ be any table partitions in *par*. From the following invariant state of *VerticalPartitioning*

$$\forall r1, r2 : partitions! \bullet \exists a : \mathbb{F}_1\, ATTRIBUTE \bullet \\ a = \mathrm{dom}(r1.columns \cap r2.columns) \land \\ (a, r1) \in primaryKey \land (a, r2) \in primaryKey$$

we can derive that the intersection between $R1$ and $R2$ is the primary key of both relations (disjointness).

$$\exists a : \mathbb{F}_1\, ATTRIBUTE \bullet \\ a = \mathrm{dom}(R1.columns \cap R2.columns) \land \\ (a, R1) \in primaryKey \land (a, R2) \in primaryKey$$

Let *priKey* be the primary key in the original table *org*. From the last predicate in *VerticalPartitioning*

$$\forall part : partitions! \bullet \exists pk : \mathbb{F}_1\, ATTRIBUTE \bullet \\ (pk, part) \in primaryKey \land \\ (pk, partitions!) \in \mathrm{dom} \bowtie \land \\ original? = \bowtie (pk, partitions!)$$

we can derive that *priKey* can be used to reconstruct the original table *org* by joining all partitions *par* based on the primary key *priKey* (reconstruction).

$$org = \bowtie (priKey, par)$$

Thus, both horizontal and vertical partitioning specifications follow the correctness rules of partitioning. □

Theorem 5 *An IP address is unique in the Internet.*

Proof: Let *node*1 and *node*2 be arbitrarily chosen nodes in the Internet. Every node is assigned an IP address. From the following invariant state in *Internet*

$$\forall n1, n2 : nodes \bullet (n1.ip = n2.ip \Leftrightarrow n1 = n2) \land \\ (n1 \mapsto n2 \in connection \Leftrightarrow n2 \mapsto n1 \in connection) \land \\ n1 \mapsto n2 \in connection^*$$

we can derive that the IP address of *node*1 equals the IP address of *node*2 if, and only if, *node*1 and *node*2 refer to the same node.

$$node1.ip = node2.ip \Leftrightarrow node1 = node2$$

The invariant state above guarantees the uniqueness of an IP address in the Internet. □

Theorem 6 *Only authorized users can access the data warehouse system.*

Proof: To perform any operation (e.g. produce a report), a user must be logged onto the data warehouse system. The specification represents a user being logged onto the system by setting the user's *session* to *True*. The only operation that changes a user's session to *True* is the *Authenticate* operation. Other operations will check the user's session value before fulfilling the user's requests. Thus, we need to show that the *Authenticate* operation assigns *True* only to authorized users' sessions.

Let u be an arbitrarily chosen user in the data warehouse system. To log onto the system, the user u must submit his/her login name and password. The pre-condition of the *Authenticate* operation is:

$$(loginName? \mapsto password?) \in account$$

The operation checks if $u.loginName$ and $u.password$ match an existing account in the system (i.e. u is an authorized user). Thus,

$$(u.loginName? \mapsto u.password?) \in account$$

If this condition is satisfied, then the user's *session* u is assign *True*.

$$session' = session \oplus \{u.loginName? \mapsto True\}$$ □

Theorem 7 *A user must be logged on to perform operations in the data warehouse system.*

Proof: In the specification, there are six operations that are available to active users: change a password, produce a report, add a user, remove a user, add a privilege, and logout. A user is logged onto the system if, and only if, the user's *session* is *True*. To prove that the operations above can only be performed by active users, we need to show that the user who is invoking the operations is currently logged on.

Let u be an arbitrarily chosen user who invokes one of the operations above. One of the pre-conditions for those operations is that the user's session must be *True*. That is,

$$(usr? \mapsto True) \in session$$

Thus, to be able to invoke any of the operations above, $u.session$ must be $True$ as well. This means u is currently logged onto the system.

$$(u \mapsto True) \in session$$ □

Theorem 8 *An account maps a login name to a password uniquely.*

Proof: Proof by contradiction. Accounts are modelled as a partial function from a login name to a password.

$$account : LoginName \pfun Password$$

Suppose $account$ does not map a login name to a password uniquely. Thus, there exists a login name ln that maps to different passwords, say $p1$ and $p2$.

$$ln \mapsto p1 \in account \land ln \mapsto p2 \in account$$

From the definition of a partial function [35]:

$$X \pfun Y == \{R : X \rel Y \mid (\forall x : X;\ y, z : Y \bullet x \mapsto y \in R \land x \mapsto z \in R \Rightarrow y = z\}$$

we can derive that if $ln \mapsto p1 \in account$ and $ln \mapsto p2 \in account$, then $p1$ and $p2$ must be identical. This conclusion contradicts the supposition; hence an account maps a login name to a password uniquely.

Another operation that changes and may violate the property of $account$ as a partial function is $AddUser$. Therefore, we need to show that the $AddUser$ operation also guarantees the uniqueness of mapping between login names and passwords.

Let u be a new login name to be added to $account$. The pre-condition of the $AddUser$ operation is

$$new? \notin \mathrm{dom}\ account$$

$AddUser$ is successfully performed only if the pre-condition above is satisfied. Hence, there was no existing account that has a login name equal to u. In addition, $AddUser$ uses functional overriding, $\oplus$, that guarantees that the result of the operator is a function as well.

$$account' = account \oplus \{new? \mapsto password?\}$$

This operator guarantees the uniqueness of an account. □

Theorem 9 *Only administrators have the privilege to perform account management.*

Proof: Proof by contradiction. To prove this theorem, we need to show that *AddUser*, *RemoveUser*, and *AddPrivilege* are executed only if the user that requests the operations is an administrator.

Suppose u is a user that is able to invoke one of the operations above, and u is not an administrator (i.e. u is an analyst).

$$(u \mapsto Analyst) \in group$$

One of the pre-conditions of the operations above is

$$(admin? \mapsto Administrator) \in group$$

This pre-condition prevents u from adding or removing a user, and from assigning an access privilege to a user. The fact that u belongs to the analyst group violates the pre-condition. Thus, the system does not fulfill u's request. This fact contradicts the supposition. Hence, we can conclude that only administrators are allowed to perform account management. □

Theorem 10 *The data warehouse system guarantees that users can only produce reports based on their access privileges.*

Proof: Let u and q be an arbitrarily chosen user and query, respectively. The user u submits the query q to the data warehouse system. The access privileges of each user are stored as a relation between a login name and a cross product of a relation and an access type.

$$privilege : LoginName \leftrightarrow (Relation \times ACCESSTYPE)$$

An authorized user is one that submits a query, where the query does not violate the user's privileges.

$$\forall r : q?.relations \bullet (usr? \mapsto (r, q?.accessType)) \in privilege$$

This invariant state is a necessary and sufficient condition for the data warehouse system to continue to process the user query. Therefore, to produce a report, the user u must have all necessary privileges.

$$\forall r : q.relations \bullet (u \mapsto (r, q.accessType)) \in privilege$$

The predicate above guarantees that any reports produced by the data warehouse system do not violate the access privileges of the users. □

4.4 Concluding Remarks

This chapter has presented the error checking procedures used to validate the correctness of the specification, and provided rigorous proofs for several important properties of the system. The specification has successfully passed the mandatory syntax and type checking. However, a few domain checks are left unproven. The difficulty of proving these domain checks mechanically is due to the complexity of using Z/EVES as a proof support tool. The available automatic commands for proving theorems in Z/EVES "will only succeed in proving easy theorems, and then only when the way has been prepared." [37]. Thus, one has to guide Z/EVES to complete the proof process because of the complexity involved. The ability to provide this guidance requires extensive knowledge about the Z/EVES's inference mechanism.

It is a much more expensive process to prove a specification formally than to write the specification [6]. Thus, the appropriate level of proof for different systems varies depending on the risks of the systems' failures. For life-critical systems, it may be appropriate to employ a formal expensive proof method. For other less-critical systems, an informal proof method is likely sufficient.

The specification in this thesis was validated using rigorous proof methods because:

- rigorous proofs can be produced within a reasonable amount of time, and are adequate to show that the specification has captured the desired properties.
- the use of more expensive methods is likely not suitable for proving web-based data warehouses, considering the fast pace of the business world.

Chapter 5

Conclusions

This section presents conclusions, a summary of contributions, and suggestions for future work.

5.1 Conclusions

This thesis provided a formal specification of several important aspects of web-based data warehouses. It discusses the requirements, the specification, and the validation of web-based data warehouses. Chapter 3 presented the requirements and the formal specification of such systems. The specification focused on the generic fundamental principles in data warehouses, instead of using a specific case study. Therefore, the specification can be used as a basis to build data warehouses.

Chapter 4 presented the validation of the specification. The Z specification in this thesis has been type checked using Z/EVES. In addition to this mechanical checking, I provided rigorous proofs of several important properties of the systems. The desired properties were stated as theorems, and then, based on the specification I show that the properties have been captured in the models.

5.2 Summary of Contributions

This thesis contributes in the following ways:

1. It formalises the main concepts of data warehouses and provides a basic model of the functions of web-based data warehouses.

2. It identifies and integrates fundamental principles in database systems into data warehouse systems.

3. It provides a case study of modelling web-based data warehouses in Z.

5.3 Future Work

The following aspects are potential for future work of this thesis:

1. Using the framework presented in this thesis to develop domain specific data warehouse applications.

2. Animating the specifications, i.e. exploring the dynamic aspects of the specifications, to reveal any errors, by translating them into an executable language (e.g., prolog) and running it [6].

3. Providing additional proofs of the Z specifications, such as, initial state calculation, pre-condition calculation, and inconsistency check. A guide for performing these checks using Z/EVES is available in [37].

4. Refining the present specifications towards their operational equivalence. The specifications in this thesis are abstract. Therefore, they need to be refined towards implementation details.

Appendix A

The Z Notation

This section lists the meaning of the Z notation used in this thesis. The definition of the Z notation in the list is as in [42].

Z Paragraphs, Declarations

[X]	given set
$S \mathrel{\widehat=} T$	horizontal schema definition
$X == e$	abbreviation definition
$T ::= A \mid B\langle\langle E \rangle\rangle$	free type definition

Expressions, Schema Expressions

(a, b)	tuple
{a, b}	set display
$X \times Y$	cross product
let $V == E \bullet P$	local definition
ΔS	schema name prefix
ΞS	schema name prefix
$S \fatsemi T$	sequential composition

Numbers and Finiteness

$\mathbb{N}$	natural numbers
$\mathbb{N}_1$	positive integers
$\mathbb{Z}$	integers
$\mathbb{F}$	finite set
$\mathbb{F}_1$	non-empty finite set
$\#$	number of members of a finite set
div	division

Predicates

$=$	equality
$\in$	membership
$\wedge$	conjuction
$\vee$	disjunction
$\Rightarrow$	implication
$\Leftrightarrow$	equivalence
$\forall$	universal quantification
$\exists$	existential quantification
$\exists_1$	unique quantification

Relations, Functions

$\leftrightarrow$	relation
$\mapsto$	maplet (ordered pair)
dom	domain
ran	range
$\lhd$	domain restriction
$\rhd$	range restriction

$\lhd$	domain anti-restriction
$\rhd$	range anti-restriction
$\sim$	relational inversion
$(\!\vert\!)$	relational image
$\oplus$	overriding
*	reflexive transitive closure
$\pfun$	partial function
$\fun$	total function

Sets

$\neq$	inequality
$\notin$	non-membership
$\emptyset$	empty set
$\subseteq$	subset
$\subset$	proper subset
$\cup$	set union
$\cap$	set intersection
$\setminus$	set difference
$\bigcup$	generalized union

Sequences

seq	finite sequence
seq_1	non-empty finite sequence
head	first element
last	last element
front	all but the last element
disjoint	disjointness

References

[1] Agarwal, S., Agrawal, R., Deshpande, P. M., Gupta, A., Naughton, J. F., Ramakrishnan, R., and Sarawagi, S., "On the Computation of Multidimensional Aggregates", *Proceedings of the 22nd VLDB Conference*, Mumbai (Bombay), India, 1996, pp. 506-521.

[2] Agrawal, R., Gupta, A., and Sarawagi, S., "Modeling Multidimensional Databases", *Research Report RJ10014*, IBM Almaden Research Center, 1996.

[3] Alagar, V. S. and Periyasamy, K., *Specification of Software Systems*, Springer-Verlag, 1998.

[4] Anahory, S. and Murray, D., *Data Warehousing in The Real World: A Practical Guide for Building Decision Support Systems*, Addison Wesley, 1997.

[5] Ballard, C., Herreman, D., Schau, D., Bell, R., Kim, E., and Valencic, A., "Data Modeling Techniques for Data Warehousing", *ITSO SG24-2238-00*, IBM, 1998.

[6] Barden, R., Stepney, S. and Cooper, D., *Z in Practice*, Prentice Hall, 1994.

[7] Barros, R. S. M., "On the Formal Specification and Derivation of Relational Database Applications", *Ph.D. Thesis*, University of Glasgow, 1994.

[8] Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[9] Bowen, J. P., "Z: A Formal Specification Notation", Marc Frappier and Henri Habrias (eds.), Chapter 1, *Software Specification Methods: An Overview Using a Case Study*, pp. 3-19, Springer-Verlag, 2001.

[10] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", Michael Stonebraker (ed.), *Readings in Database Systems*, 2nd Edition, pp. 5-15, Morgan Kaufmann, 1994.

[11] Connolly, T. M. and Begg, C. E., *Database Systems: A Practical Approach to Design, Implementation, and Management*, 2nd Edition, Addison Wesley, 1998.

[12] Date, C. J., *An Introduction to Database Systems*, 6th Edition, Addison Wesley, 1995.

[13] Deering, S. and Hinden, R., "Internet Protocol, Version 6 (IPv6) Specification", RFC 1883, December 1995. Available at: http://www.ietf.org/rfc/rfc1883.txt

[14] Devlin, B., *Data Warehouse: From Architecture to Implementation*, Addison Wesley, 1997.

[15] Ehikioya, S. A. and Indratmo, Y., "Formal Design and Verification of Web-based Data Warehouse", *Proceedings of the International Symposium on Database Technology & Software Engineering, WEB and Cooperative Systems*, Baden-Baden, Germany, August 1-4, 2000, pp. 25-34.

[16] Elmasri, R. and Navathe, S. B., *Fundamentals of Database Systems*, 3rd Edition, Addison Wesley, 2000.

[17] Ezeife, C. I., "Selecting and Materializing Horizontally Partitioned Warehouse Views", *Elsevier Journal of Data and Knowledge Engineering*, Vol. 36, No. 2, pp. 185-210, 2001.

[18] Gardner, S. R., "Building the Data Warehouse", *Communications of the ACM*, Vol. 41, No. 9, pp. 52-60, September 1998.

[19] Gupta, A., Harinarayan, V., and Quass, D., "Aggregate-Query Processing in Data Warehousing Environments", *Proceedings of the 21st VLDB Conference*, Zurich, Swizerland, 1995, pp. 358-369.

[20] Gyssens, M. and Lakshmanan, L. V. S., "A Foundation for Multi-Dimensional Databases", *Proceedings of the 22nd VLDB Conference*, Mumbai (Bombay), India, 1996, pp. 106-115.

[21] Hall, M., *Core Servlets and JavaServer Pages*, Prentice Hall, 2000.

[22] Harrington, J. L., *Ethernet Networking Clearly Explained*, Morgan Kaufmann, 1999.

[23] IBM, "Meta Data Management for Business Intelligence Solutions: IBM's Strategy", *Data Management Solutions White Paper*, November 1998.

[24] Inmon, W. H., *Building the Data Warehouse*, John Wiley & Sons, 1996.

[25] Jarke, M., Lenzerini, M., Vassiliou, Y., and Vassiliadis, P., *Fundamentals of Data Warehouses*, Springer-Verlag, 2000.

[26] Kimbal, R., *The Data Warehouse Toolkit*, John Wiley & Sons, 1996.

[27] Kizan Corporation, *Designing and Implementing a Data Warehouse Using Microsoft SQL Server 7.0.*, Microsoft Corporation, 1998.

[28] Li, C. and Wang, X. S., "A Data Model for Supporting On-Line Analytical Processing", *Proceedings of the Conference on Information and Knowledge Management*, November 1996, pp. 81-88.

[29] Meisels, I. and Saaltink, M., *The Z/EVES Reference Manual (for Version 1.5)*, ORA Canada, September 1997.

[30] Müller, R., Stöhr, T., and Rahm, E., "An Integrative and Uniform Model for Metadata Management in Data Warehousing Environments", *Proceedings of the International Workshop on Design and Management of Data Warehouses*, Heidelberg, Germany, June 1999, pp. 12.1-12.16.

[31] Noaman, A. Y. and Barker, K., "A Horizontal Fragmentation Algorithm for the Fact Relation in a Distributed Data Warehouse", *Proceedings of the Eighth International*

Conference on Information and Knowledge Management, Kansas City, Missouri, USA, November 1999.

[32] Özsu, M. T. and Valduriez, P., *Principles of Distributed Database Systems*, 2nd Edition, Prentice Hall, 1999.

[33] Pedersen, T. B. and Jensen, C. S., "Multidimensional Data Modeling for Complex Data", *TimeCenter Technical Report TR-37*, November 1998.

[34] Peters, J. F. and Pedrycz, W., *Software Engineering: An Engineering Approach*, John Wiley & Sons, 2000.

[35] Potter, B., Sinclair, J., and Till, D., *An Introduction to Formal Specification and Z*, Prentice Hall, 1996.

[36] Robertson, S. and Robertson, J., *Mastering the Requirements Process*, Addison Wesley, 1999.

[37] Saaltink, M., *The Z/EVES User's Guide*, ORA Canada, September 1997.

[38] Sommerville, I., *Software Engineering*, 6th Edition, Addison Wesley, 2001.

[39] Spivey, J. M., *The Z Notation: A Reference Manual*, Prentice Hall, 1998.

[40] Wiener, J. L., Gupta, H., Labio, W. J., Zhuge, Y., Garcia-Molina, H., Widom, J., "A System Prototype for Warehouse View Maintenance", *Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7, 1996, pp. 26-33.

[41] The World Wide Web Virtual Library: The Z Notation. Available at: http://www.afm.sbu.ac.uk/z/

[42] Z/EVES Quick Reference Card. Available at: ftp://ftp.ora.on.ca/pub/doc/97-5493-07.ps.Z