Integrating Legacy Systems with Enterprise Data Store

By

Sonia Narang

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science

Department of Computer Science

Winnipeg, Manitoba, Canada, 2000

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

Integrating Legacy Systems with Enterprise Data Store

BY

Sonia Narang

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

Integrating data from multiple, heterogeneous databases and other information sources has been one of the leading issues in database research and industry. There are two approaches towards solving the data integration problem - Multidatabase Systems and Data Warehousing. This thesis contributes towards solving the problem of data integration using the data warehousing approach. This thesis argues that the operational data store (ODS) fails to provide true operational integration and introduces a new data integration architecture by defining an architectural construct – the Enterprise Data Store (EDS). An Enterprise Data Store is a repository of data that represents an integrated view of enterprise operations and is built for corporate-wide operational informational processing and transactional processing of common business operations. This thesis presents an architecture and a comprehensive set of algorithms for synchronizing the EDS with the operational systems. The philosophy behind the EDS synchronization architecture is to exploit the metadata component of the data warehouse system. A very important component of the data warehouse metadata store is the mapping between the operational systems and the data warehouse. This research, based on this component of the data warehouse metadata store, identifies four kinds of mappings - entity to entity, attribute to attribute, key to key, and record to record mappings that can be used to synchronize the EDS with the operational systems. These mappings are modeled in a metadata model which is implemented as the metadata mapper. The mapping data and algorithms stored in the metadata mapper are then used by the synchronization algorithms to synchronize the EDS with the operational systems. The proposed synchronization architecture offers many advantages and is different from early synchronization architectures (e.g., WHIPS) that are based on a materialized view approach.

Acknowledgments

I owe a debt of gratitude to a large number of people for supporting me in the production of this thesis. First, my sincere thanks to my supervisor Dr. Ken Barker for his immense support and encouragement during this research. He gave me the freedom to explore my own ideas and helped me frame them by scrutinizing, and criticizing them from every angle. His encouragement, patience, and understanding have helped me accomplish this scholarly piece of work.

Many thanks to my committee members Dr. Peter Graham and Dr. Bob McLeod for all their insights and comments, and for taking time to read the thesis.

A special thanks to my best friend Deepti Mathur without her love, help, understanding and care this success was impossible. You gave me the confidence and strength through out this struggle. You were always there as a true friend whenever I needed your help and support. I am indeed fortunate to have a friend like you.

A loving thanks to my Fiancée Jason Paul for always brightening my days with his beautiful smiles, cute little notes and beautiful roses. You kept the romance in my logical world. Without your love, understanding, patience, and encouragement the road to success would not have been so easy.

This acknowledgment will be incomplete without paying homage to my family - my parents and my two loving sisters. Their immovable confidence, constant encouragement and selfless love have helped me climb the ladder of success. You were always there for me even though you were miles apart.

Contents

List of Figures

Chapter 1.

Introduction.

Not long ago, managing an enterprise was similar to steering an ocean liner across the pacific - slow reaction time, leisurely pace, two degrees of freedom, and an empty ocean. Today, managing an enterprise is more like flying an airliner over New York City - quick reaction time, jet speeds, many degrees of freedom, and a crowded airspace. In the cockpit, the pilots depend on their instrument panel to tell them just what's happening at any one moment.

: Dr. Richard Hackathorn

Survival has become more and more difficult due to the increased competition and complexity in the marketplace. To survive, you have to be "Big". Indeed, it is - the survival of the fittest. Corporations today are facing more and more deregulations, mergers and acquisitions [1, 3]. Ten years ago, businesses were simple, competition was less. Each business focused on a specific domain. For example, banks focused on their core banking operations, insurance companies sold insurance, mortgage brokers transacted mortgages, financial institutes dealt with investments. Over the last ten years, many of these organizations have changed their business model and now offer a wider product shelf. For example, today banks not only deal with cash management but also offer mortgages, insurance and investments. Similarly, financial institutions are not only selling investments but also insurance and mortgages. To support the new business model new application systems were built over time. These systems support the functionality they were designed for and capture massive amounts of data across divisions and departments. As a result most large organizations have between 15 and 40 legacy systems

that currently store and disseminate massive amounts of data. Instead of building and reinforcing business operations, this data, which is dispersed and often duplicated across the organization, is eroding the very infrastructure for which it was collected [2]. How this huge, redundant, dirty data can be integrated, purged and transformed into information, has been an open research problem for years and will persist into the foreseeable future.

This lack of data integration is recognized by nearly all corporations today. Ideally, organizations can reengineer to better support the current business model. It is a daunting task to architect and engineer systems that not only match the company's new business processes but are also flexible enough to deal with change [3].

1.1 Existing Approaches to Data Integration

Providing integrated access to multiple, distributed, heterogeneous databases and other information sources has become one of the leading issues in database research and industry [4]. The research performed in this area can be broadly classified into two categories - *Multidatabase Systems* and *Data Warehousing*.

A multidatabase system (MDBS) is defined as an interconnected collection of autonomous databases. A multidatabase system typically integrates information from preexisting, heterogeneous, autonomous, local databases in a distributed environment and presents global users with transparent methods to use the total information in the system. The integration is achieved by building a layer of software called a multidatabase management system (MDBMS) that runs on top of independent database management systems (DBMSs) and provides users with the facilities to access various databases. Substantial research has been done in query processing and transaction management for multidatabase systems [5].

The database industry recently approached this problem differently. Inmon, introduced a

new approach to informational processing. He introduced two important architectural constructs - *the data warehouse and the operational data store [6,7].* To effectively do informational processing he proposed that there needs to be a foundation of data, known as the data warehouse (or the "information warehouse"). Inmon defines a data warehouse as *"subject-oriented, integrated, time-variant, non-volatile, contains both summary and detailed data to support management's decision"*. The data warehouse is a collection of integrated, historical data. The data warehouse is built from data formerly residing in the applications of the corporation. Data from each source that may be of interest is extracted in advance, translated and filtered, merged with other relevant information from other sources, and stored in a centralized repository [9].

The operational data store (ODS) is the place where collective, corporate online operational integration occurs. An ODS is built for satisfying the collective, integrated, operational needs of the corporation. Inmon defines an ODS as *"a subject-oriented, integrated, volatile, current or near current collection of data in support of day to day detailed operational decisions"*. An ODS looks very much like a data warehouse when it comes to its first two characteristics, subject orientation and integration. However, the remaining characteristics of an ODS are quite different from a data warehouse [6].

The first difference between an ODS and a data warehouse is perceived in terms of volatility of data. An ODS is volatile whereas a data warehouse is non-volatile. This means that an ODS can be updated as a normal part of processing. A data warehouse contains snapshots (a database dump as of some past moment of time); a new snapshot is created whenever a change needs to be reflected in the data warehouse. The second difference is the timeliness of the data found in the two environments. An ODS contains only current data. A data warehouse contains historical as well as current data. The third difference between an ODS and a data warehouse is that an ODS contains detailed data only, while a data warehouse contains both detailed and summary data. The fourth difference is the type of processing performed in the two environments. An ODS is mainly built for operational processing whereas the data warehouse is built for

informational/DSS (Decision Support System) processing. Operational processing refers to short running queries that access limited amounts of data. It is used for detailed and up-to-the-second decisions. Whereas, informational processing refers to complicated long running queries that access large amounts of data. It is used for long term analysis and trend detection.

In spite of the differences between the two environments, similar steps are taken to build either a data warehouse or an ODS. The steps are -

1) Design a data model - the data model provides the structure and content definition of the informational needs of the corporation. This data model is then implemented into a data warehouse or an ODS.

2) Locate the best data the corporation has to furnish the structure and content of the data model - the best data is determined by evaluating accuracy, completeness, and timeliness of source data. This step is referred to as *identification of the system of record [8]*. The system of record once defined, becomes the source of data for populating a data warehouse or an ODS.

3) Extract, transform, integrate and load data - data is transformed as it passes from application or operational sources into a data warehouse or an ODS. Transformation is needed to map the system of record to a data warehouse or an ODS. The transformation includes such activities as converting data, decoding/encoding data, altering key structures, altering physical structures, reformatting data, internally representing data, recalculating data, and so forth. After the data has undergone transformation, loaders are used to load the data. Transformation and integration of the operational data is achieved by building a layer of software usually referred to as transformation and integration layer.

4) Refresh the data warehouse or the ODS - once the ODS or the data warehouse is

populated, it must be kept synchronized with the operational systems[1]. This is achieved by capturing the changes in the operational systems and propagating them to the data warehouse or the ODS. These changes are reflected in the ODS as updates but a new snapshot is created whenever a new change needs to be reflected in the data warehouse.

1.2 Comparing the Two Approaches

The fundamental difference between multidatabase systems and data warehousing is that the multidatabase system is a logical integration of multiple, heterogeneous, autonomous databases while a data warehouse or an ODS is a physical integration. A data warehouse or an ODS is a repository of data built by cleaning and integrating data from multiple systems. On the other hand, an MDBMS is a layer of software built to provide an integrated view of data residing in multiple, heterogeneous, autonomous database systems.

Query processing in a multidatabase system requires determining the appropriate set of data sources and generating the appropriate subqueries or commands for each data source. Results obtained after the execution of these subqueries are translated, filtered and merged, and sent to the user of the multidatabase system. On the other hand, query processing in a data warehousing environment is much simpler since the user poses a query on a centralized database system. This difference makes query processing in data warehousing more efficient as data has already been cleaned and integrated from heterogeneous data sources by using a common model after resolving the semantic and syntactic differences among various data sources. Furthermore, warehouse data can be accessed without tying up the original data sources (e.g., holding locks, slowing down processing), and is available even when the original data sources are inaccessible. The

[1] The terms operational systems, legacy systems and application systems will be used interchangeably throughout the thesis.

warehousing approach may be considered an "active" or "eager" approach to information integration, as compared to the multidatabase system approach that is considered "passive" or "lazy", where processing and integration starts when a query arrives [9].

Another difference between the two approaches is that the data warehouse and the ODS are built for querying and analysis while multidatabase systems are built for querying and updates. In other words, a user can issue an update transaction to the MDBMS layer that will translate the global update to the respective local updates. These updates will then be sent to the respective local databases for execution. The MDBMS layer is responsible for ensuring the consistency and managing the concurrency of global transactions. Transaction management in multidatabase systems is a complicated and interesting research area that is not applicable to the data warehousing approach.

One potential drawback of the data warehousing approach is that queries are limited to the data contained in the data warehouse. The needs of users are determined in advance and the data relevant to their needs is extracted and maintained in the data warehouse. Hence, the data warehousing approach is only suitable for users with predictable needs. Another drawback is that since the data is physically copied from original data sources to the data warehouse and is typically refreshed every 1 to 24 hours, it may not be as current as the data contained in the original data sources [9]. Thus, systems that require querying and analysis to be performed on the current data will not find the data warehousing approach suitable.

Both approaches are viable solutions to the data integration problem and are appropriate for specific domains. We believe that in spite of certain drawbacks with the data warehousing approach, it is a much simpler and more attainable solution to the data integration problem. Its strengths lie in -:

1) Taking away the load of informational processing from application data sources. This means data can be accessed without tying up the original data sources. User queries

are no longer dependent on availability of the original data sources. Also, delays in query processing caused by busy and slow data sources are also eliminated.

2) It makes query processing more efficient by eliminating the significant processing required for translation, filtering, and merging of data from multiple, heterogeneous and autonomous data sources.

3) It provides the flexibility to modify and store information that is not maintained in the original data sources. The data is extracted from original data sources and stored in a data warehouse. This gives users flexibility in analyzing and storing historical and summarized information which is inappropriate to store in the original data sources.

As mentioned above, the first drawback of the warehousing approach - the currency of the data is due to the cost incurred in refreshing data in the warehouse synchronously with the application data sources. Most data warehouses are refreshed asynchronously between every 1 to 24 hours depending on the need of the organization. I feel that as technology advances and more research is done in this area, this drawback may become insignificant. The second drawback is that data warehousing is not suitable for users with unpredictable needs. I classify this as more of an analysis and design problem than a data warehousing problem. If analysis and design is done thoroughly to predict the needs of the users there should not be any queries outside the domain of the data warehouse. Also, a data warehouse is built in an iterative manner which gives the opportunity to better understand and predict users' requirements.

1.3 The Data Integration Approach Taken for this Research

This thesis contributes towards the problem of data integration using the data warehousing approach. The research to date on the ODS and the data warehouse will be used as a framework. This research focuses on operational integration and hence more closely resembles the architectural construct - the ODS. This thesis argues that the ODS

architecture does not truly integrate the operations of the enterprise and therefore introduces a new architectural construct - the Enterprise Data Store (EDS). The next section illustrates this problem with the ODS architecture and explains how the EDS resolves the problem.

1.3.1 The Problem with the ODS Architecture

Many corporations have established a plethora of older operational applications which has created what is commonly called the "Legacy System" environment or the "Spider Web" environment [6]. Not only is the customer information scattered over disparate operational systems, there are also similar business operations being performed by each application system. For instance, a bank may have a loan system, a mortgage system, an insurance system, an investment system and a cash management system to manage products and services it offers to its clients. All these systems perform some *common business operations* such as collecting client information (Name, age, gender, occupation, dependents, spouse, address, e-mail, phone, fax), client financial information (assets, liabilities, insurance, mortgage), service instructions (electronic fund transfer - EFT, statement preferences), product rates (insurance or mortgage or interest rates), payment collection, etc. There is *operational redundancy* in these application systems. Not only do these operations duplicate customer information across different operational systems, there is also an excessive cost incurred to maintain these operations. Since the customer information was collected over different times it is entirely possible that this information is also different across the operational systems. This possibility brings into question the validity of information provided by these systems. In other words, which is the most accurate and current customer information?

When building an ODS, such questions are answered during the identification of the system of record [8]. Identification of the system of record is a difficult task if multiple candidate sources exist. If there is a single source, selecting the "best" source is trivial but this occurs rarely in an environment where integration is a prime motivator. If multiple

sources exist, the selection process will ultimately be a complex task requiring complicated logic to determine the best source of data. Even though the ODS is populated with the best source of data the other application systems continue to see the incorrect data. Consider an example where the client information is scattered over three application systems (see Figure 1). The client record in the ODS is formed by getting first name, last name and address from Application A, date of birth and gender code from Application B and marital status code and spouse name from Application C. Any changes to these fields in the operational systems must be captured and synchronized with the ODS. In such cases when we have multiple sources forming a system of record, a very complicated synchronization logic is needed as we have more complicated mappings between the ODS data and application sources' data. Further, although we have defined that the best source for getting the address for a client is Application A, Application B continues to see

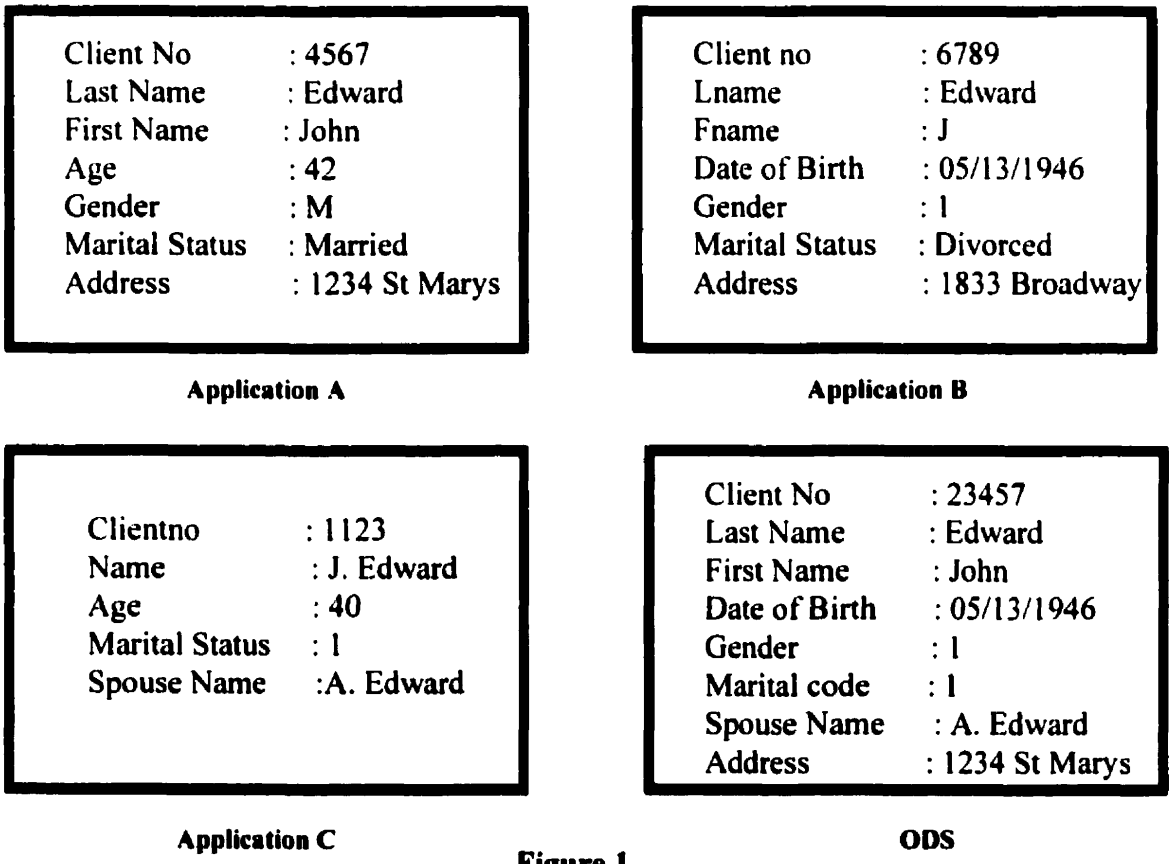

Figure 1

different address information as opposed to Application A and the ODS. Moreover a client can call his insurance agent for an update on his insurance premium, and at the same time report a change of address. If Application B is the insurance system, then it subsequently contains the most current address information that is neither in Application A nor in the ODS. True operational integration cannot be achieved until we eliminate the *operational redundancy* caused by these common business operations that will in turn eliminate duplication of data across legacy systems.

1.3.2 Proposed Architectural Construct - Enterprise Data Store (EDS)

To achieve true operational integration this research proposes a new integration architecture defining a new architectural construct - the *Enterprise Data Store (EDS). An Enterprise Data Store is a repository of data that represents an integrated view of enterprise operations and is built for corporate wide operational informational processing and transactional processing of common business operations.* The EDS integrates and cleans data from disparate operational systems. It also eliminates redundant processing of common business operations by reengineering and moving these operations onto the EDS. The EDS truly integrates an enterprise's data by providing a consistent view of data across the enterprise. In the above example, all the applications and the EDS would then see the same view of the data. The EDS is the system of record for the common business operations. This means any processes to maintain common business operations only exist in the EDS.

Moving and reengineering common business operations onto the EDS complicates synchronization of the operational systems with the EDS. In the ODS environment, any changes to the operational systems are captured and propagated to the ODS. The direction of movement of data is from the operational systems to the ODS. In the EDS architecture, along with the direction of movement of data from the operational systems to the EDS data also moves from the EDS to the operational systems. This additional direction of movement of data is due to reengineering and moving of the common business operations

onto the EDS. By moving these operations to the EDS, the EDS becomes the primary (owns and maintains) source of data for these common business operations and operational systems become the secondary (read only). In other words, the application processes that maintain this data exist in the EDS. Any changes to this data in the EDS must be propagated to the operational systems. Hence to keep the EDS synchronized with the operational systems, we will need a dual propagation mechanism. This thesis proposes an architecture for synchronizing the EDS with the operational systems. It also provides algorithms to keep the EDS and operational systems synchronized. These algorithms use metadata for the synchronization. Though metadata is an essential component of the data warehouse architecture, the development of this component is usually ignored. This thesis also makes a contribution in this area by designing a metadata model to store the mapping between the operational systems and the EDS. This model is then used by the synchronization algorithms to synchronize the EDS with the operational systems.

To summarize, this thesis makes the following contributions -

1. It defines a new data integration architecture by defining an architectural construct, the Enterprise Data Store (EDS).

2. It provides an architecture and algorithms for synchronizing the EDS with the operational systems.

3. It provides a metadata model to store the mapping between the operational systems and the EDS. It also illustrates the use of metadata for synchronizing the operational systems with the EDS.

1.4 Outline Of the Thesis

A review of literature on data warehousing and an overview of research in multidatabase

systems are discussed in Chapter 2. Chapter 3 describes the characteristics of the Enterprise Data Store and compares them with the Operational Data Store. It also discusses the merits and liabilities of the EDS. Chapter 4 presents the architecture for synchronizing the EDS with the operational systems. Chapter 5 describes algorithms for synchronizing the EDS with the operational systems. This chapter also presents a metadata model and illustrates how metadata can be used for synchronization. Finally, Chapter 6 makes some concluding comments and suggests directions for future research.

Chapter 2.

Related Work

This chapter reviews other work related to this thesis by broadly categorizing research on the problem of data integration into multidatabase systems and data warehousing. Data warehousing has been a prominent buzzword in the database industry, but attention from the database research community has been limited. A noteworthy exception is the researchers at Stanford University who have contributed significantly towards the data warehousing approach. Due to the difference in industrial and academic perspective, the related work in data warehousing has been further categorized in this chapter into industrial and academic perspective.

Inmon [6,7] introduced two very important architectural constructs - the data warehouse and the operational data store (ODS). There are substantial differences between the two constructs and each is built to satisfy specific needs of an organization. The proposed research in this thesis more closely resembles the operational data store. This research identifies the problems associated with the ODS and proposes a new architectural construct called the Enterprise Data Store (EDS). Although the ODS is a well-known architectural construct in industry, it has not been explored within the database research community. This thesis focuses on the architectural construct called ODS/EDS and describes a new area of research in data warehousing.

The rest of the chapter is organized as follows. Section 2.1 presents a review of previous work on data warehousing from an industrial perspective and an academic perspective. Section 2.2 presents a review of work on multidatabase systems.

2.1 Data Warehousing

2.1.1 Industrial Perspective on Data Warehousing

Industrial contributions to data warehousing have been made by identifying and defining the two architectural constructs, namely, the data warehouse and the operational data store. This section reviews existing literature on these constructs. Along with these constructs it also reviews previous work related to metadata and defining the "system of record". Further, it reviews literature on corporate data architecture which is also known as the corporate information factory (CIF). The purpose of this section is to give a holistic view of data warehousing that will act as a framework for the thesis.

2.1.1.1 Data Warehouse

Inmon [7, 34] defines a data warehouse as a subject-oriented, integrated, time-variant and non-volatile collection of data in support of management's decisions. As data enters the data warehouse from the operational environment, it is transformed and integrated. Upon entering the data warehouse, data goes into the current detail data. It resides there and is used until one of the following three events occur: it is purged, it is summarized, and/or it is archived. The kinds of data found in the data warehouse are classified as current detail data, older detail data, lightly summarized data, highly summarized data and metadata. The different levels of data within the data warehouse receive different levels of usage. Inmon [34] discusses the two most important issues of the data warehouse design - granularity and partitioning. Inmon also reviews the technology features required for satisfactory data warehouse processing.

2.1.1.2 Operational Data Store

Inmon [6] defines an architectural construct called the operational data store (ODS). The ODS is built for operational integration. There are fundamental and important differences between the ODS and the data warehouse. These differences are also noted by Inmon,

Imhoff and Battas [3]. Inmon [6] defines the ODS as a subject-oriented, integrated, volatile, current or near current collection of data in support of day to day detailed operational decisions. An essential part of the ODS environment is the definition of the system of record. The system of record is the application data that feeds the ODS (as defined by Inmon, Imhoff, Battas [3] and Inmon [8]). Inmon [6] also discusses the pros and cons of moving the system of record to the ODS. The informational processing found in the ODS is for the clerical community making detailed, up-to-the-second decisions. This kind of informational processing is very different from that found in the data warehouse. Inmon also briefly discusses the management of different processing windows to section off parts of the day for various activities like loading and informational processing of the ODS. Creation of these windows helps in managing the processing load on the ODS.

Inmon, Imhoff and Battas [3] give an in depth description of the operational data store. They discuss what an ODS will and will not do for a company. Their work serves as a guide for building and maintaining the ODS. The evolving needs of the organization and the technological advancements have led to the creation of this architectural model. There are three classes of ODSs. These classes are defined based on the frequency with which the data is refreshed in the ODS. In Class I ODS, updates in application systems are propagated to the ODS in a synchronous (immediate) manner. In a Class II ODS, updates in the application systems are stored and forwarded to the ODS on an hourly or even half-hourly basis. In a Class III ODS, the updates in application systems are propagated to the ODS in an asynchronous manner on a twenty-four-hour-or-more basis. The foundation of the design of the operational data store is the corporate data model, where major subjects (entities) are identified. The authors also introduce the term corporate information factory (CIF) and discuss several components constituting the CIF. In depth description of the CIF is also given in Inmon, Imhoff and Sousa [1].

Inmon, Imhoff and Battas [3] mention that many companies are finding that they must reengineer their systems to better support their current business processes. They suggest

from a technology standpoint that building an ODS and a data warehouse can be helpful to the reengineering effort in a number of ways. An exploration of the relationship between an ODS and a reengineered environment show that there are a number of similarities. The similarities lead to the idea of using an operational data store as a basis for reengineering. In this scenario, where the ODS is built for reengineering, applications are migrated from legacy environments to the ODS. Their research proposes migrating the current system of record over time to the operational data store. This will require changes to the transformation and integration layer to deal with synchronization of the old and new system of record as well as changes to deal with exceptions such as collisions. Collisions occur when users simultaneously update the same data elements in the ODS as well as legacy systems. The ODS built for reengineering may have some or all of the system of record built inside the ODS. Moving the system of record to the ODS requires dual propagation to synchronize the ODS with the operational systems.

2.1.1.3 Defining the System of Record

Inmon [8] notes that the definition of the system of record is one of the most important steps in the development of the data warehouse. The system of record entails the identification of the "best" source data in the operational environment. The content and structure of the system of record are determined by the data warehouse data model. The selection of the "best" source data is determined by the following criteria: the most accurate source data, the most complete source data, the most timely source data, the most structurally compatible data, and the data nearest to the operational source. The system of record has many different facets. Some of these facets are: the attributes that make up the system of record, the mapping between the system of record and the data warehouse, summarizations, frequency of transformations, etc. Each facet must be defined by the database designer. The system of record, once defined, then becomes the source of data for populating the data warehouse.

2.1.1.4 Metadata

Inmon [10] discusses the importance of metadata in the data warehouse environment and describes each component of the metadata in detail. Metadata plays an important and active role in the data warehouse environment as compared to the operational environment. This is because the data warehouse and the operational systems serve different user communities. IT (Information Technology) professionals are users of the operational systems whereas, DSS (Decision Support System) analysts are users of the data warehouse. A DSS analyst must know what data is available and where it is in the data warehouse. This information is provided to the DSS analyst by metadata. The data in the warehouse environment spans over a broad spectrum of time. As a result, the same data structure will have multiple forms. To track these changes over a period of time, Inmon suggests versioning the metadata.

One of the most important contents of the data warehouse metadata store is the mapping between operational systems and the data warehouse. The typical contents of the mapping that are stored in the data warehouse metadata store are: identification of source fields, simple attribute to attribute mappings, attribute conversions, physical characteristics conversions, name changes, key changes, defaults, logic to choose from multiple sources, summarization algorithms, etc.

It is shown in the following chapters that metadata is also an important component of the proposed EDS architecture. This research proposes the use of metadata for synchronization of the EDS with the operational systems.

2.1.1.5 Corporate Data Architecture

Inmon, Imhoff and Sousa [1] describe the challenges and problems facing organizations today and suggest the need for an information ecosystem to overcome these challenges and problems. An information ecosystem as a system with different components, each serving a community directly while working in concert with other components to produce

a cohesive, balanced information environment.

Three fundamental business pressures that are fueling the evolution of the information ecosystem are: growing consumer demand, increased competition and complexity, and continued demands for improvements in operating efficiencies. In response to these very real business challenges, companies must be able to support more than just classical business operations. Competitive corporations need capabilities to support business intelligence and business management that can leverage their legacy environment. To achieve these capabilities the authors suggest creating an information ecosystem that will orchestrate the use of various information technologies and constructs like data warehousing, data marts, OLAP, data mining, etc. A corporate information factory (CIF) is the physical embodiment of the notion of an information ecosystem. The corporate information factory is made up of the following components: applications (legacy systems), an integration and transformation layer, data warehouse, data mart, operational data store, metadata, the internet and the intranet. The different components of the CIF create a foundation for information delivery and decision-making activities that can occur anywhere in the CIF. The steps required in building and managing the corporate information factory are detailed. Inmon [35] discusses how legacy systems, the operational data store and the data warehouse together form an information architecture.

2.1.2 Academic Perspective on Data Warehousing

Despite rapid advances in commercial data warehousing tools and products, most of the available systems are relatively inflexible and limited in their features. Most commercial data warehousing systems assume that the sources and the warehouse subscribe to a single data model (normally relational), that propagation of information from the sources to the warehouse is performed as a batch process (perhaps off-line), and that queries from the warehouse to the information sources are never needed. Research at Stanford University addresses these limitations by proposing a new data warehousing architecture called the WHIPS (Warehouse Information Prototype at Stanford) architecture. The

philosophy behind the WHIPS architecture is to consider data in the warehouse as a materialized view (or set of views), where the base data resides at the information sources. The propagation of changes from the base data sources to the warehouse is then essentially a matter of performing materialized view maintenance. This section summarizes data warehousing research at Stanford and provides an overview of research problems in data warehousing, the WHIPS architecture, the warehouse anomaly, materialized views and materialized view maintenance.

Widom [9] motivates the concept of data warehousing in the database research community. The paper outlines a general data warehousing architecture and proposes a number of technical issues arising from the architecture that are suitable topics for exploratory research. Widom compares the data warehousing approach to the existing data integration approaches. She classifies existing data integration approaches as lazy or on-demand and data warehousing as eager or in-advance approach. The differences between the two approaches are also discussed in Hammer, *et al.* [11]. Widom [9] proposes the basic architecture of a data warehousing system in this paper. The philosophy behind the proposed architecture is to consider the data in the warehouse as a materialized view (or set of views), where base data resides at the information sources. The author discusses various reasons why conventional view maintenance techniques are not suitable for the data warehousing approach and new techniques and algorithms must be defined. View maintenance in a warehousing environment is discussed in Zhuge, *et al.* [17]. Widom [9] also outlines various research problems that arise from the warehousing approach like change detection, translation, data scrubbing and warehouse management.

Hammer, *et al.* [11] describe the goal of the data warehousing project at Stanford (the WHIPS project). They give a brief overview of the WHIPS project and describe some of the research problems being addressed in the initial phase of the project. They consider data warehousing as a complement to, not a replacement of, passive query processing schemes. They also illustrate the basic architecture of the WHIPS project and describe its components (the monitor and integrator) in detail. Monitors detect changes to an

information source that are of interest to the warehouse and propagate them in a generic format to the integrator. Depending on the kind of information source changes can be detected by using triggers, examining log files, comparing snapshots and/or modifying application sources to emit and notify relevant changes to the monitors. The WHIPS project is currently focusing on using snapshot differential algorithms for change detection. They have implemented simple differential monitors.

The integrator described by Hammer, *et al* is implemented as a rule-based engine. Each rule is responsible for handling one kind of change notification, and is implemented as an object-oriented method. A rule is triggered whenever a monitor generates a change notification of the appropriate type. The detailed architecture of the WHIPS system is present in Wiener, *et al.* [12]. As with their earlier work, the architecture is based on a data warehouse formed as a materialized view (or views) over the information sources. They show that the decoupling between the base data on the one hand (at the sources), and the view definition and view maintenance machinery on the other (at the integrator) can lead to incorrect views at the warehouse. They refer to this problem as the *warehouse update anomaly*. There are a number of mechanisms for avoiding warehouse update anomalies such as recomputing the view, storing copies of all data involved in views at the warehouse, eager compensating algorithm (Zhuge, *et al.* [17]).

Wiener, *et al.* [12] present a system prototype for warehouse view maintenance. This research extends the basic architecture proposed in Hammer, *et al.* [11]. The authors discuss important goals that must be fulfilled by the WHIPS architecture like plug-and-play modularity, scalability, 24 by 7 operation, data consistency, and support for different source types. Different modules of the WHIPS architecture are: the view specifier, the meta-data store, the integrator, view manager(s), query processor(s), wrappers, sources and monitors, the warehouse and the warehouse wrapper. The prototype is implemented using distributed object technology. Each module is implemented as a CORBA (Common Object Request Broker Architecture) object, using the ILU (Inter-Language Unification) implementation of CORBA. The communication between objects is then performed

within the CORBA distributed object framework, where each object O has a unique identifier used by other objects to identify and communicate with O. Wiener, *et al.* also provide preliminary performance results for the WHIPS prototype.

The materialized view approach is a suitable solution for a data warehouse system. It will be shown in the following chapters that this approach is however, not suitable for the EDS architecture. This is due to the fundamental differences between the EDS and the data warehouse. In this thesis, a new approach and architecture for synchronizing the EDS with the operational systems is proposed. The new approach uses the metadata component of the data warehousing system for synchronization.

Roussopoulos [18] describes the versatility and potential of relational views. He discusses a relational view and summarizes the most important uses, techniques, and benefits pertaining to views and then presents several forms of relational views. Roussopoulos also explains why reusability of views is of great importance in data warehousing. Commercial RDBMSs discard views immediately after they are delivered to the user or to a subsequent execution phase. The cost of generating the views is for one-time use only instead of being amortized over multiple and/or shared access. In a data warehouse where query execution and I/O are magnified in volume, the mandate for reuse cannot be ignored.

Zhuge, *et al.* [17] introduce a new algorithm known as ECA (Eager Compensating Algorithm) for avoiding warehouse anomalies. In a warehousing environment, the sources can inform the warehouse when an update occurs. This information alone, however may not be sufficient to incorporate the update into the warehouse views. Thus, the warehouse may have to issue queries to some of the sources to determine the additional data needed to update the views. Since these queries are evaluated at the sources later than the corresponding update, the sources' states may have changed due to concurrent updates at the sources. As a result, the execution of the queries at the sources may return incorrect data that will lead the warehouse to compute incorrect views. The

ECA takes steps to avoid these warehouse anomalies. The basic idea is to add to the queries sent to the source, compensating queries to offset the effect of concurrent updates. The authors also discuss two improvements to the basic ECA algorithm. First, the ECA-Key Algorithm (ECA^K) that includes a key from every base relation involved in the view so deletions can be handled at the warehouse without issuing a query to the source. Second, the ECA-Local algorithm (ECA^L) that combines the compensating queries of ECA with the local updates of ECA^K to produce a streamlined algorithm that applies to general views. An initial performance study in [17] shows that ECA is more efficient than periodically recomputing the warehouse views from scratch.

The ECA is based on a restrictive warehouse environment that assumes a single source and a single view over several base relations residing on a single source. Zhuge, *et al.* [24] extend this work to the warehouse environment that assumes multiple sources and a view over multiple relations residing on multiple sources. In their research, they propose a new family of algorithms, the strobe family, for multi-source warehouse consistency.

Multidatabase systems are another approach to the data integration problem. There has been substantial research in this field and some of the research is also applicable to data warehousing. The next section reviews the multidatabase system approach to the data integration problem.

2.2 Multidatabase Systems

Database systems serve critical functions and represent significant capital investment. Many organizations have several different computers and database systems. In many cases, this environment must be preserved while also addressing the need to share information on a more global basis. Integrated access is required to semantically similar information at different nodes and with different data representations. Multidatabases typically integrate information from preexisting, heterogeneous local databases in a distributed environment and present global users with transparent methods to use the total

information in the system. A key feature is the autonomy that individual databases retain to serve their existing customer set.

Providing integrated access to multiple, distributed, heterogeneous databases and other information sources has been one of the leading issues in the database research community for over a decade. The research challenges in this area can be broadly classified into: data translation and integration, query processing, and transaction management. Research issues in data translation and integration deal with defining a robust common model, translating and integrating each underlying database (or local database) to obtain a unified schema (schema translation and integration) and defining a mapping methodology between the unified schema and the underlying databases. Ram [36] discusses some of these challenges in detail.

2.2.1 Data Translation and Integration in Multidatabase Systems

Data translation is achieved in multidatabase systems by defining a common model and translating the schemas of various database systems into the common model. Integration is achieved by integrating the translated schemas into a global conceptual schema. Integration is optional in some of the multidatabase architectures and the absence of global conceptual schema is considered to be of significant advantage. Özsu and Valduriez [5] discuss multidatabase system architectures with and without a global conceptual schema and present schema translation and integration techniques using the E-R (Entity Relational) model as the common model.

The common model must be powerful enough to express various relationships and semantic information captured by different database systems. Several "semantic" models have been developed to serve as the common model. For example, the Dataplex [40] multidatabase system is based on a relational model, the Amoco Distributed Database system [41] uses an extended relational model, Multibase [42,43] uses a functional model and Pegasus [38] takes advantage of the object oriented model. Once researchers

construct a common model, they still have the problem of resolving schema and data conflicts among various database systems to obtain a unified schema. Semantic differences such as synonyms, homonyms, naming conflicts, and differences in attribute formats and field length need to be resolved. Kim and Seo [37] provide a comprehensive framework for understanding schematic and data heterogeneity among independently created and administered relational databases. An interesting challenge here is to develop automated tools to help identify and resolve these semantic differences.

2.2.2 Overview of Research in Multidatabase Systems

Ram [36] highlights some of the problems and their solutions associated with heterogeneous distributed database systems (HDDS). HDDS is another name for multidatabase systems. A major challenge of integrating diverse databases is hiding the heterogeneity of the constituent databases from users without sacrificing the autonomy of the constituent databases. This implies that HDDS should neither impose changes on existing databases nor require any reprogramming of the local database management systems (DBMSs). HDDS should appear as a single integrated database. This includes hiding the heterogeneity of file systems, data models, database languages, and data semantics, as well as the hardware and operating systems on which the constituent databases run. Ram [36] also discusses two approaches for developing HDDSs. She classifies the two approaches as "unified schema" and "multi-database". The first approach advocates establishing an integrating model to define a unified schema of the constituent databases also known as the global schema. The second approach argues that complete integration is not necessary to preserve the autonomy of the constituent databases. Each database continues to operate in an independent manner and also forms a part of a federation of users who can share information. Multidatabase system architectures with and without global conceptual schema are also discussed in Özsu and Valduriez [5]. Ram [36] also classifies challenges in a heterogeneous database environment into 1) definition of an integrating model, 2) schema integration, 3) mapping methodology, and 4) data administration functions like transaction management and

recovery.

Özsu and Valduriez [5] discuss MDBS architecture with and without a global conceptual schema. They define database integration as a process by which information from participating databases can be conceptually integrated to form a single cohesive definition of a multidatabase; in other words, it is the process of designing the global conceptual schemas. Their focus is on architectures with global conceptual schemas. They mention that data integration can occur in two steps: schema translation and schema integration. In the first step, the participating local database schemas are translated to a common intermediate canonical representation using translators [19,23]. In the second step, each intermediate schema is integrated into a global conceptual schema. Their discussion on schema integration is based on the work done by Batini, *et al.* [34]. Batini, *et al.* classifies integration methodologies as binary and *n*-ary. According to Batini, *et al.*, schema integration occurs in a sequence of four steps: preintegration, comparison, conformation, and merging & restructuring. During the comparison phase both the naming and structural conflicts are identified. Structural conflicts occur in four possible ways: as type conflicts, dependency conflicts, key conflicts, or behavioral conflicts. Classification of schema and data conflicts among component relational databases organized into a multidatabase system is presented by Kim and Seo [37]. Conformation is the resolution of the conflicts that are determined at the comparison phase. In the merging and restructuring step all intermediate schemas must be merged into a single database schema and then restructured to create the "best" integrated schema. Batini, *et al.* define the three dimensions of merging and restructuring as completeness, minimality, and understandability.

Kim and Seo [37] provide a framework for comprehensive enumeration and classification of schema and data conflicts among component databases organized into a multidatabase system. They define a MDBS as a federation of independently developed component database systems (CDBSs). The MDBS provides a homogenizing layer on top of the CDBSs giving users the illusion of a homogeneous system. Since CDBSs operate

independently they may include structural and representational discrepancies, or conflicts, called schematic and data heterogeneity. These conflicts must be resolved so that MDBS users can access the underlying CDBSs with a single uniform database language rather than a different database language for each CDB. This research presents a comprehensive framework for classifying these conflicts. Kim & Seo believe that such a framework is required to develop an MDBS schema definition, query language and the tools needed by multidatabase designers.

They assume the MDBS common data model is relational; that is each CDB schema is first converted to a semantically equivalent relational schema, and the multidatabase schema is constructed as a view of these relational CDB schemas. They classify conflicts at the highest level as either schema or data conflicts. There are two basic causes of schema conflicts. First is the use of different structures (tables and attributes) for the same information. Second is the use of different specifications for the same structure; these include different names, data types and constraints for semantically equivalent tables and/or attributes. Schema conflicts are broadly classified as 1) table-versus-table conflicts, 2) attribute-versus-attribute conflicts, and 3) table-versus-attribute conflicts. Broadly, there are two types of data conflicts 1) wrong data conflicts that are based on violating integrity constraints, and 2) conflicts based on different representations for the same data.

Ahmed, *et al.* [38] describe in their research Pegasus, a heterogeneous multidatabase system developed by the Database Technology Department at Hewlett-Packard Laboratories. To support the various database systems with different data models, languages, and services, a powerful data model that will resolve mapping and integration problems between diverse data systems is needed. Pegasus takes advantage of object-oriented data modeling and programming capabilities. It uses both type and functional abstractions to deal with the mapping and integration problems. Data abstraction and encapsulation features of Pegasus object model provide an extensible framework for dealing with various kinds of heterogeneities in traditional and non-traditional data

sources. The Pegasus model therefore overcomes many limiting capabilities of mapping and integration inherit in other multidatabase systems. The model is based on the Iris object model.

The unifying data definition and data manipulation language of Pegasus is the heterogeneous object structured query language (HOSQL). Multiple data sources can interoperate via Pegasus without having an integrated global schema. A local data source is represented in Pegasus by an imported schema that looks like a Pegasus schema, but the underlying data is in the local data source. A complete or partial mapping of a local schema can be visible through Pegasus. HOSQL statements may refer directly to the individual imported schemas. Integration in Pegasus is optional and deals with semantic and schematic heterogeneity among different databases, all of which have imported schemes in Pegasus. The authors describe three kinds of semantic and schematic heterogeneity; domain mismatch, schema mismatch and object identification.

Chawathe, *et al.* [19] describe the architecture of TSIMMIS (The Stanford-IBM Manager of Multiple Information Sources) that provides integrated access to diverse and dynamic information residing in heterogeneous information sources. TSIMMIS goal is not to perform fully automated information integration that hides all diversity from the user, but rather to provide a framework and tools to assist humans in their information processing and integration activities. The TSIMMIS project uses a simple, self-describing (tagged) object model called the Object Exchange Model (OEM) as its common model.

The two main components of the TSIMMIS architecture are translators (or wrappers) and mediators. Translators convert queries over information in the common model into requests that the source can execute, and convert the data returned by the source into the common model. A mediator is a system that refines in some way information from one or more sources. A mediator embeds the knowledge that is necessary for processing a specific type of information. To build a mediator it is not required that a mediator understand all of the data it handles, and no person or software component needs to have

a global view of all the information handled by the system. It is important to note that there is no global database schema, and mediators can work independently. The TSIMMIS architecture focuses on generating wrappers and mediators automatically or semi-automatically using a high level description language.

The WHIPS architecture, discussed previously, and the TSIMMIS architecture were both defined by researchers at the Stanford University. Both architectures focus on integrating data from heterogeneous data sources but are based on two different approaches towards data integration. WHIPS is a data warehousing architecture and TSIMMIS is a multidatabase architecture. Data translation and integration is inherent in both approaches to data integration. Therefore, the warehousing approach can take advantage of data translation and integration research in multidatabase systems. The WHIPS architecture uses the research in TSIMMIS by using wrappers/translators in their architecture.

Hammer, *et al.* [23] describe an architecture for template-based wrappers in the TSIMMIS system. They introduce a wrapper implementation toolkit for quickly building wrappers. The goal is to minimize the effort that goes into developing and writing wrappers and to quickly gain access to new information sources. The philosophy behind their "template based" translation methodology is as follows. The wrapper implementer specifies a set of templates (rules) written in a high level declaration language that describe the queries accepted by the wrapper as well as the objects it returns. When an application query matches a template, an implementer-provided action associated with the template is executed to provide the native query for the underlying source. When a source returns the result of the query, the wrapper transforms the answer which is represented in the data model of the source into a representation that is used by the application.

2.2.3 Data Translation and Integration in Multidatabase Systems and Data Warehousing

The data translation and integration problem is inherent in both approaches of data integration - multidatabase systems and data warehousing. In an MDBS, data translation and integration is achieved by defining a global conceptual schema, using a common model, that integrates and resolves data and schema conflicts between the underlying data sources. Unlike a data warehouse, there is no physical implementation of the global conceptual schema. In a data warehouse system, data translation and integration is achieved by 1) defining and implementing a data warehouse model, and 2) building a transformation and integration layer that resolves data and schema conflicts among various data sources before loading data into the data warehouse. In both approaches there is an integrating schema (the global conceptual schema in a MDBS and the data warehouse schema in the data warehouse) defined using a common model and the data from the underlying data sources need to be translated and integrated to map to the defined schema by resolving schema and data conflicts among various data sources. Hence, the research done in data translation and integration in multidatabase systems is applicable to data warehousing as well.

2.2.3.1 Data Translation and Integration in the WHIPS Architecture.

In the WHIPS architecture, data translation is achieved using wrappers or translators and data integration is achieved using integrators. The WHIPS architecture makes use of some of the existing data translation and integration techniques introduced by multidatabase research. Chawathe, *et al.* [19] and Hammer, *et al.* [23] discuss this research.

2.2.3.2 Data Translation and Integration in the EDS Architecture.

Data integration will be achieved in the proposed EDS architecture by eliminating redundant processing of common business operations by reengineering and moving these

operations onto the EDS and also by integrating data belonging to application specific operations. Data translation between the EDS and the operational systems will be achieved by storing the mapping between the EDS and the operational systems in the metadata mapper.

This chapter has broadly organized research on the problem of data integration into - data warehousing and multidatabase systems. The existing literature on architectural constructs the data warehouse and the ODS was reviewed. The term Corporate Information Factory was defined and previous work related to metadata and defining the system of record was reviewed. This chapter also presented a comprehensive review of data warehousing research at Stanford. This review provided an overview of research problems in data warehousing, the WHIPS architecture, the warehouse anomaly, materialized views and materialized view maintenance. Substantial research has been done in Multidatabase systems. The research challenges in multidatabase systems were broadly classified into - data translation and integration, query processing and transaction management. The chapter focused on challenges in the area of data translation and integration as they are common to both multidatabase systems and data warehousing. The data translation and integration techniques chosen by various architectures were reviewed briefly. Also, a brief overview of research in multidatabase systems was presented.

In this chapter existing data integration approaches and architectures were also briefly reviewed. The following chapters discuss the proposed data integration architecture using the new architectural construct the Enterprise Data Store. The Enterprise Data Store was introduced in Chapter 1 and the following chapter discusses its characteristics in detail.

Chapter 3.

Enterprise Data Store

A new architectural construct the Enterprise Data Store (EDS) was defined in Chapter 1. This chapter builds on the previous one by characterizing the EDS. Since in the proposed architecture, the EDS replaces the ODS, the differences between their characteristics are discussed in detail. This chapter also reviews the corporate data architecture with the application systems, the ODS, and the data warehouse. This is followed by the proposed changes to the corporate data architecture with the introduction of the EDS. Section 3.1 presents characteristics of the Enterprise Data Store and compares them with the ODS. Section 3.2 reviews the corporate data architecture with the application systems, the ODS, and the data warehouse. This review is followed by the changes to the corporate data architecture with the new architectural construct - the EDS. Section 3.3 presents the advantages of the Enterprise Data Store. Finally, Section 3.4 discusses the liabilities of the Enterprise Data Store.

3.1 Characteristics of the Enterprise Data Store (EDS)

This section describes the characteristics of the Enterprise Data Store and compares them with the characteristics of the operational data store [3,6]. The EDS has the following characteristics-:

- Subject Oriented
- Distributed System of Record
- Integrated
- Volatile

- Dual Currency of Data
- Current, Detailed, No Summary, No History
- Informational and Transactional Processing

3.1.1 Subject Oriented

Like the ODS, the EDS is designed and organized around the major subjects of the corporation [3]. The major subjects of the corporation are real world objects like CUSTOMER, PRODUCT, PAYMENT, POLICY, CLAIM, SHIPMENT, etc., that collectively provide a complete and integrated operational definition of the enterprise. The Enterprise Data Store is not organized around any specific applications or functions. The subject orientation is necessary to represent a collectively integrated image of data across the corporation.

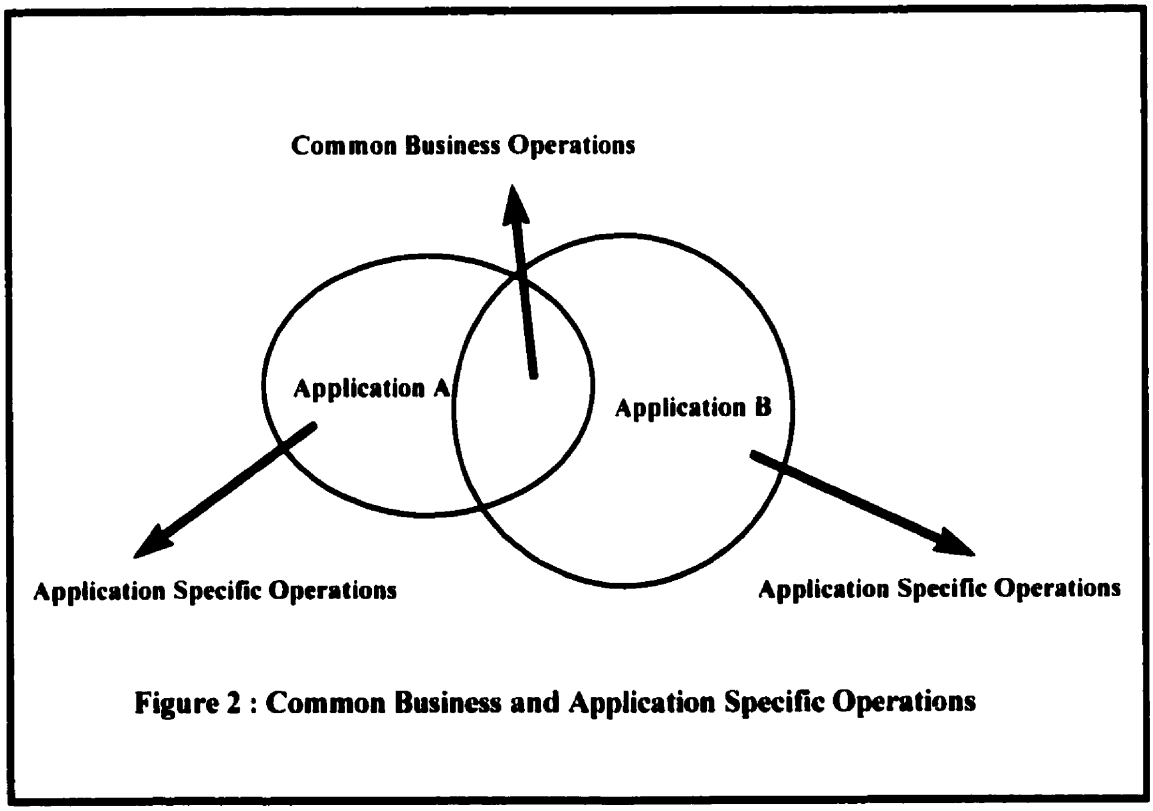

Figure 2 : Common Business and Application Specific Operations

3.1.2 Distributed System of Record

This is a characteristic specific to the EDS and does not belong to the ODS. Each ellipse in Figure 2 represents an application system that consists of data and processes. The figure illustrates that there is a fair amount of intersection of data and processes among these application systems. The intersection of data and processes among these application systems represents the common business operations whose data and processing should be moved to the EDS. In other words, the EDS becomes the primary data source or the

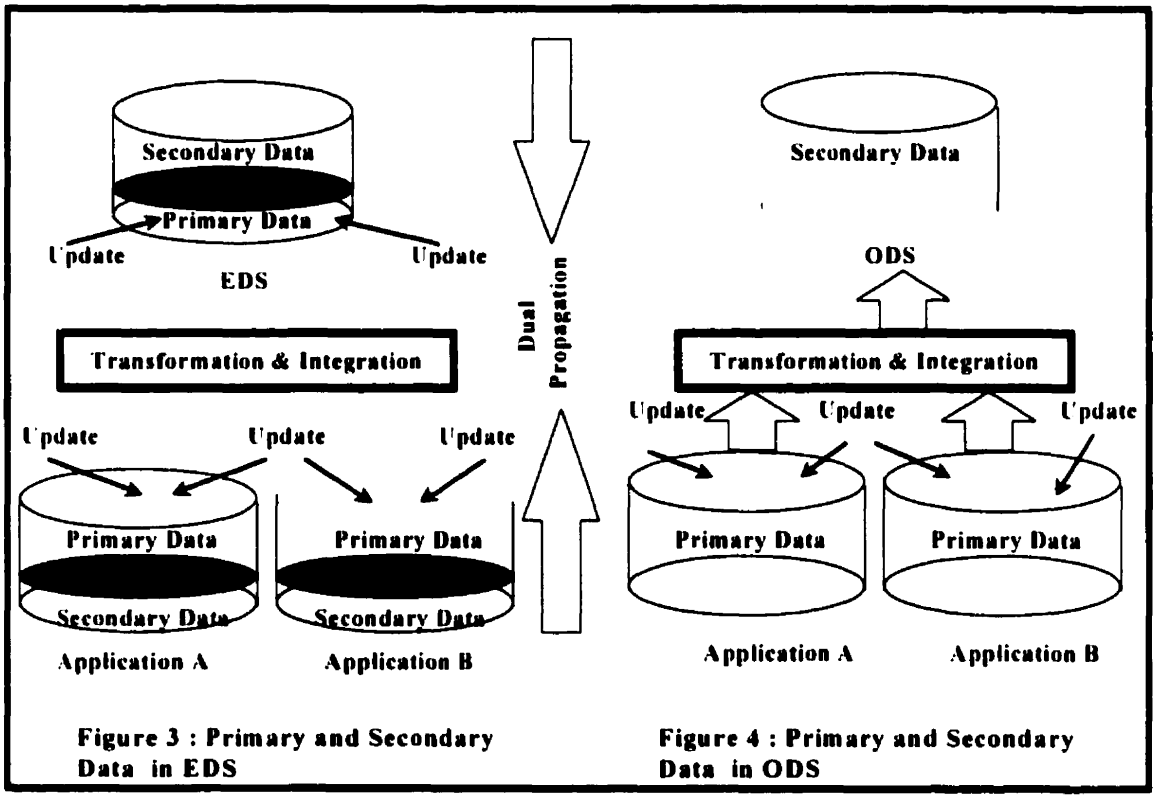

Figure 3 : Primary and Secondary Data in EDS

Figure 4 : Primary and Secondary Data in ODS

system of record for the common business operations and the application systems become the secondary. After the removal of common business operations, the application systems only consist of application specific operations. Data used for the application specific operations is the primary data of the application systems. This data is only maintainable by the application systems. If primary data from the application systems is transformed

and loaded into the EDS, it becomes the secondary data of the EDS. Secondary data of the application systems belongs to the common business operations whose processing has been moved to the EDS. This data is the primary data of the EDS and is only maintainable by the EDS. To summarize, the application systems and the EDS are made up of two kinds of data - *primary and secondary data*. The primary data can only be modified by the system that contains it whereas the secondary data is read only. Figure 3 and Figure 4 compare the EDS and the ODS with respect to this characteristic. In the ODS architecture, the system of record or the primary data only exists in the application systems. In the EDS architecture, some primary data exists in the applications systems as well as in the EDS. A formal classification of types of data in the EDS architecture is presented in Chapter 4.

3.1.3 Integrated

There are two types of integration in the EDS - 1) integration of common business operations, and 2) integration of application specific operations. Integration of common business operations is achieved by identifying the common business operations in the legacy systems and then reengineering and moving these operations onto the EDS. Integration of application specific operations is similar to the ODS architecture [3]. This integration is achieved by selecting, cleaning, transforming and integrating the best application specific data from the legacy systems and then loading it into the EDS.

3.1.4 Volatile

Updates on the EDS can also be classified into "Direct Updates" and "Indirect Updates". The direct updates belong to the primary data of the EDS. The processes that maintain the primary data exist in the EDS and are responsible for triggering these updates. The indirect updates occur on the secondary data of the EDS and are similar to updates on the ODS [3]. These updates are called indirect updates as these updates are generated as a result of changes in the legacy or application systems. Every time the data in the legacy

systems changes, the EDS needs to be synchronized. These changes are captured in the legacy systems and are propagated to the EDS by an integration and transformation layer. These updates can be performed synchronously or asynchronously.

3.1.5 Dual Currency of Data

Dual currency in the EDS is caused by the two types of data found in the EDS. Since the primary data is maintained by the EDS, it is the most current data available. On the other hand, the currency of the secondary data depends on when the data is refreshed in the EDS from the operational systems. This refreshment of data can be performed synchronously or asynchronously. This characteristic is unique to the EDS and is not applicable to the ODS architecture.

3.1.6 Current, Detailed, No History and No Summary

Like the ODS, the EDS serves the operational community and is similar to the ODS in this characteristic [3]. The operational community is concerned with day to day decision-making. These decisions are up-to-the-second decisions and are not used for long term analysis and trend detection. Such decision-making is best supported by detailed data. Therefore, the EDS contains detailed data. Also, the EDS is built for transactional processing of common business operations that certainly implies the transactional or detailed nature of the EDS.

The EDS should not contain summary data for the following reasons. We have two tiers of data in our architecture (see Figure 3). The first tier consists of the operational systems and the second tier consists of the EDS. The primary data in the first tier is the secondary data in the second tier and the primary data in the second tier is the secondary data in the first. Any changes to the primary data in either tier must be propagated to the other tier. In other words, the two tiers must be synchronized with each other. Undoubtedly synchronization will be easier, if the data in the EDS is maintained at an atomic level

rather than at a rolled up level. Another reason for discouraging summary data in the EDS is that summary data is only accurate as of the instant in time it is created. In the very next instant summary data may become inaccurate due to the constantly changing nature of the EDS [3].

There is a very clear demarcation between the EDS and the data warehouse as there is between the ODS and the data warehouse [3,6,7]. The EDS contains current valued and near current valued data whereas the data warehouse contains historical data, as well as near current valued data. There is no place for history in the EDS. If archival data is found, there must be strong operational reasons for its presence. For example, an organization may decide to keep six months of transactions in the EDS for operational analysis.

3.1.7 Informational and Transactional Processing

The EDS is an architectural construct where informational and transactional processing co-exist. The primary data of the EDS is used for operational informational processing and on-line transactional processing (direct updates) of the common business operations. The secondary data of the EDS is used for operational informational processing and off-line transactional processing (indirect updates) of the application specific operations. This is in contrast to the ODS which supports operational informational processing and off-line transactional processing only [3]. There is no on-line transactional processing in the ODS.

3.1.8 Comparing the Reengineered ODS and the EDS

The ODS has also been perceived as being built for reengineering. In this scenario, applications are migrated from legacy environments to the ODS. The ODS (built for reengineering) is discussed in [3] and was also discussed in Chapter 2. The ODS (built for reengineering) and the EDS architecture are similar in that both architectures propose

migrating the system of record to the ODS and the EDS respectively. However, there are significant differences between the two architectures:

1. The EDS provides true operational integration by eliminating redundant processing of common business operations by reengineering and moving these operations onto the EDS. It explicitly identifies what needs to be reengineered to achieve true operational integration. In the ODS architecture (built for reengineering), the idea is to use the ODS as a basis of reengineering so some or all of the system of record is moved to the ODS.

2. In the EDS architecture, data is either owned by the EDS or an application system. This means it is modifiable by one system (EDS or application) and read-only in the others. Therefore, collisions that are caused by users simultaneously updating data elements in the EDS and an application system are eliminated. In the ODS (built for reengineering) architecture, data is modifiable by the ODS as well as application systems and hence collision detection and resolution mechanisms are needed.

Further, this research contributes by proposing a formal architecture and set of algorithms for synchronizing the EDS with the operational systems.

3.2 Corporate Data Architecture

This section discusses how different architectural constructs and legacy systems combine to create a corporate data architecture. This section is divided into two subsections. The first discusses Inmon's corporate data architecture consisting of the application systems, the ODS, and the data warehouse. The second discusses the proposed corporate data architecture with the application systems, the EDS, and the data warehouse.

3.2.1 Corporate Data Architecture with Application Systems, ODS, and Data Warehouse

The operational data store, the data warehouse, and the application systems combine to create a corporate data architecture that Inmon refers to as the "*corporate information*

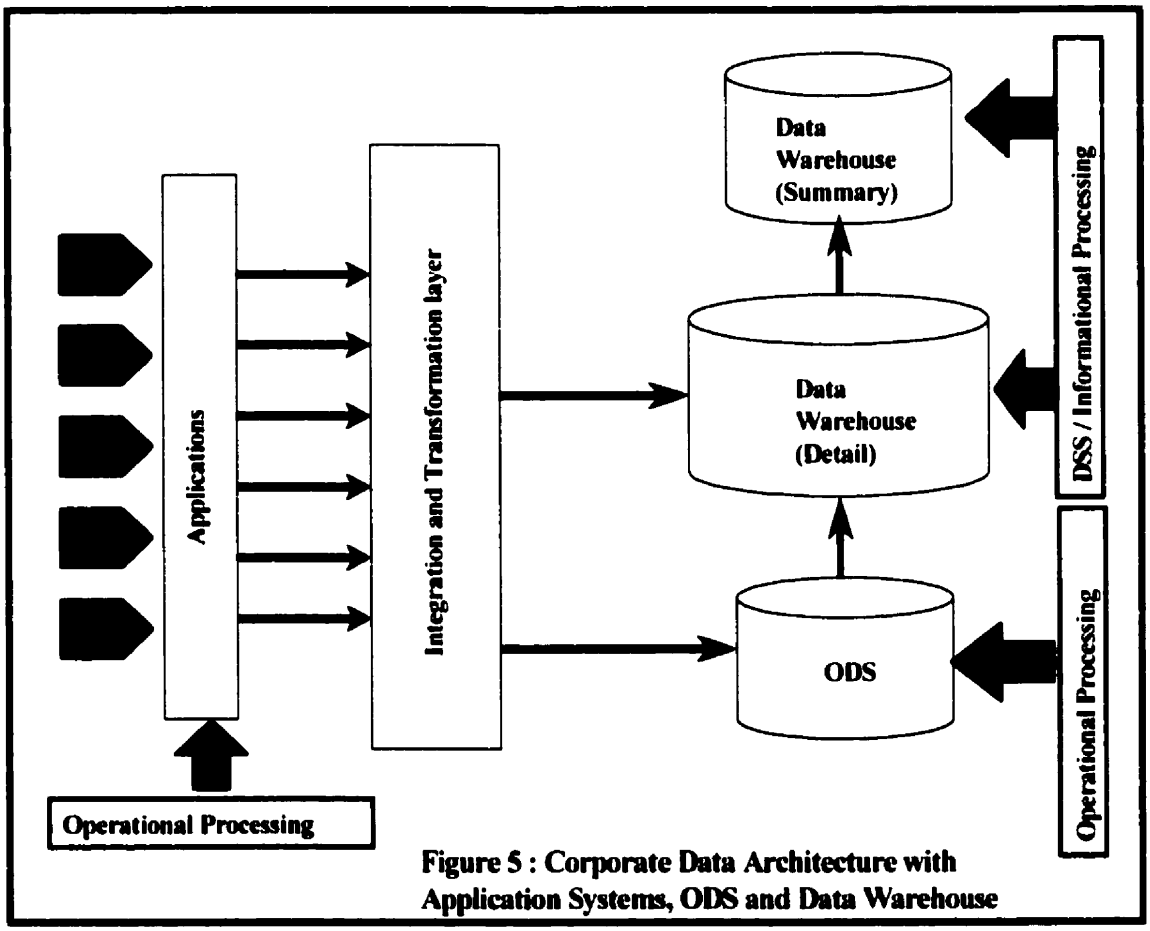

Figure 5 : Corporate Data Architecture with Application Systems, ODS and Data Warehouse

factory" [1,3]. There are many variations to the corporate information factory (CIF) but the focus of this section will be those that include the ODS, the data warehouse, and the application systems. Figure 5 shows that raw, detailed data enters into the corporate data architecture (or corporate information factory) through old application systems. This data is entered by the users of the applications. The users of the applications are clerks, sales/service personnel, and possibly the customers of the corporation themselves. Raw data is refined as it passes through the application systems. The data in the application

systems is best described as detailed, immediate, and application-oriented. The application data is then fed into the integration and transformation layer (I&T layer). The integration and transformation layer consists of sets of programs that integrate and transform functional or application data into corporate data. The functional data is organized around specific operations of the organization and hence has an application flavor to it. Corporate data gives a consolidated view of the corporate operations and is devoid of any application flavor. As data passes out of the integration and transformation layer it is simultaneously fed into the ODS and the data warehouse. In some cases, data from the integration and transformation layer is only fed into the ODS and then the refined data from the ODS is fed into the data warehouse. Note that the data in the ODS and the data warehouse is still at a detail level. The detailed data in the data warehouse is then summarized to produce summary data in the data warehouse.

Different kinds of processing take place at different components in the corporate information factory. The ODS is the architectural construct that enables operational integrated corporate informational processing to occur. It is possible, though rare, for DSS/informational processing to occur in the ODS environment. The users of the ODS are concerned with immediate and very direct decisions, such as:

- How much money is in an account right now?
- Where is a shipment right now?
- What coverage is there for a policy right now?

Classical DSS/informational processing occurs in the data warehouse. DSS users are concerned with decisions that are much broader and long term, such as:

- What type of customer is the most profitable for our corporation?
- Where has sales activity been highest in the spring-time for the past three years?
- Over the years, how has transaction activity changed?

To summarize, the standard flow of data throughout the CIF is from left to right (see Figure 5), that is, from the consumer to the application, from the application to the I & T

layer, from the I & T layer to the ODS and the data warehouse or from the I & T layer to the ODS and then from the ODS to the data warehouse. Also, as discussed above there are fundamental differences between data and processing that occur at each architectural construct of the corporate informational factory.

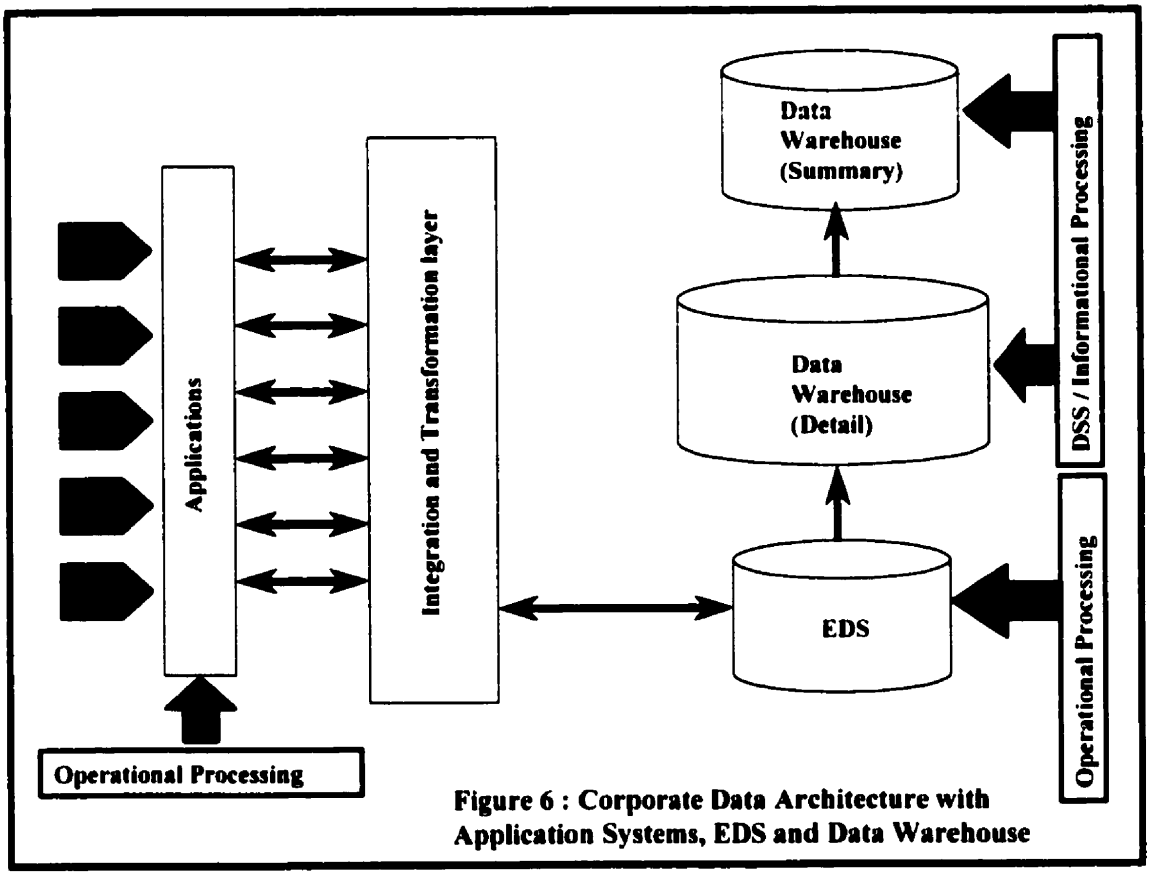

Figure 6 : Corporate Data Architecture with Application Systems, EDS and Data Warehouse

3.2.2 Proposed Corporate Data Architecture with Application Systems, EDS, and Data Warehouse

Figure 6 shows the proposed corporate data architecture with the application systems, the EDS, and the data warehouse. Two fundamental changes are made to the corporate data architecture when the ODS is replaced with the EDS. The changes are:

1. Data not only flows from the application systems to the EDS but also flows back from

the EDS to the application systems through the integration and transformation layer. In the EDS architecture, operational integration is achieved by cleaning and integrating data from disparate application systems and by eliminating redundant common business operations by moving and reengineering these operations onto the EDS. This implies that the EDS contains processes to maintain common business operation data and is the primary source for this data. As previously discussed in Sections 3.1.2 and 3.1.6 any changes made to the primary data in the EDS must be propagated to the application systems.

2) In the corporate data architecture with the EDS, there is no direct flow of data from the integration and transformation layer into the data warehouse. The data flows from the integration and transformation layer into the EDS and then the refined and integrated data flows from the EDS to the data warehouse. As discussed before, applications are the primary source of data for application specific operations and the EDS is the primary source of data for the common business operations. Application specific data is refined and integrated as it passes through the integration and transformation layer. Unless the application data from the integration and transformation layer is loaded and combined with the common business operations data in the EDS, data integration is not achieved. Therefore, it does not make sense to feed the data warehouse directly from the integration and transformation layer.

The EDS contains truly integrated operational data. Building the EDS creates a foundation of data that can be used by new application systems. Since the EDS contains the *best* data that has been cleaned, integrated, and is now residing on the new technology, it is logical to use the EDS to feed the data warehouse or any new application systems of the organization.

3.3 Advantages of the EDS Architecture

The EDS truly integrates operations of the enterprise. It eliminates the possibility of

inconsistent data caused by transactional processing of duplicated common business operations across operational systems. Further, it provides an integrated view of enterprise operations for corporate-wide informational processing.

The EDS is an enterprise repository of data that becomes the ideal feed for the new application systems of the enterprise (i.e., the data warehouse). Data for any new application systems must be extracted from the EDS. As the name suggests, it is the enterprise repository of data, any data needed to support new enterprise operations must be contained in the EDS. If the EDS does not have the data needed for a new application system, it should be modified to contain that data. In other words, existing operational systems should only feed the EDS and the EDS should then feed any new applications needed by the enterprise.

Most organizations have made huge investments in legacy systems. These systems were built with the evolving needs of the organizations in mind and are suited to the functions for which they were designed. The EDS "thins" legacy systems by taking the load of informational processing and common business transactional processing away from the legacy systems. This opens a possibility to reengineer legacy systems as needed. The EDS also secures the organizations' investment in the legacy systems as they continue to co-exist and even perform better due to the lower operational and informational load.

Reengineering is the ideal solution to the challenge posed by the lack of integration of older applications. Unfortunately, the translation of data models and process models into new systems from a base of older, non-integrated operational applications in a rational, affordable fashion is a very difficult transition [3]. The EDS provides an effective solution by reengineering only the problem areas responsible for the lack of operational integration. The cost incurred by reengineering the problem areas is very nominal as compared to the cost incurred by reengineering the legacy systems. Additionally, organizations save costs by eliminating redundant common business operations across the legacy applications. Further, building the EDS is not as mission critical as reengineering

the older applications.

Moving common business operations to the EDS gives organizations an opportunity to reengineer these operations in a logical, phased-in manner. As a result, additional functionality can be added to these operations.

Building the EDS will help organizations take advantage of new technology.

3.4 Liabilities of the EDS Architecture

The EDS is an architectural construct where informational and transactional processing co-exist. An expensive and complicated hardware and software infrastructure is required to support the two very different kinds of processing.

By moving and reengineering common business operations to the EDS there is added complexity in terms of synchronization of operational systems with the EDS. Recall Figure 3, which shows the two tiers of data in an EDS based environment. The first tier consists of operational systems and the second tier consists of the EDS. Since updates can happen in either tier a dual propagation mechanism is needed to keep the two tiers synchronized. Chapters 4 and 5 deal with this issue and present the architecture and algorithms for synchronizing the EDS with the operational systems.

As common business operations are moved and reengineered to the EDS, changes are required to existing operational systems to support the new functionality. These issues will be discussed in further detail in the next chapter.

Chapter 4.

Synchronizing the EDS with the Operational Systems

This chapter proposes an architecture for synchronizing the EDS with the operational systems. Before discussing the proposed architecture, a brief review of existing synchronization approaches and their suitability to the EDS architecture is presented.

4.1 Existing Approaches for Synchronization

The WHIPS architecture for synchronizing the data warehouse with the operational systems is presented in [9,11,12]. As discussed in Chapter 2, the WHIPS approach has been to consider the data residing in the warehouse as a materialized view (or set of views) over the data in the operational systems. Viewing the problem in this way, synchronization of the data warehouse with the operational systems is essentially to perform materialized view maintenance. Indeed, there is a close connection between the view maintenance problem and synchronization in data warehousing [13]. As a result, the work done in traditional view maintenance [14,15,16,17] has been adapted to the problem of view maintenance in data warehousing. This work on data warehouse view maintenance is discussed in [17,18]. A system prototype (the WHIPS prototype) for data warehouse view maintenance is presented in [12]. In spite of the close connection between the data warehousing synchronization problem and the conventional view maintenance problem, there are a number of reasons why conventional view maintenance techniques cannot be directly applied to data warehousing [9]. These reasons are: 1) warehouse views are a function of the history of the underlying base data rather than a function of the underlying base data itself, 2) warehouse views are more complicated as they tend to contain highly aggregated and summarized information, 3) view definition

and the base data are decoupled as compared to the traditional view maintenance problem where the base data updates are closely tied to the view maintenance machinery, 4) warehouse views may not need to be refreshed after every modification or set of modifications in the base data, and 5) base data may need to be transformed before it can be integrated in the data warehouse environment. More work on data warehouse view maintenance is discussed in [29,30,31,32,33].

4.2 Complexity Involved with the Existing Approaches

Materialized view maintenance in the data warehouse requires a complicated architecture. The complication is due to the decoupling of the base data (at the sources) from the view maintenance machinery in the warehouse [9]. As mentioned before, sources can inform the warehouse when an update occurs, e.g., a new employee has been hired, or a patient has paid her bill. However, they cannot determine what additional data may or may not be necessary for incorporating the update into the warehouse views. When update information is received by the data warehouse, the warehouse may issue queries to the sources to get the additional information needed for maintaining the view. The queries are evaluated at the sources later than the corresponding updates, so the source states may have changed. This can lead the warehouse to compute incorrect views. This problem is referred to as the warehouse anomaly [11]. There are a number of mechanisms to avoid a warehouse anomaly. For example, recompute the view, store copies of all relations involved in views at the warehouse, eager compensating algorithm [17], the strobe algorithm [24], *etc.*

The drawback associated with the materialized view approach is querying operational or base data sources to get the additional information needed to maintain the view. This is required because all the information needed to maintain the view is not stored in the data warehouse. As a result, the architecture for synchronizing the data warehouse with the operational systems has to deal with issues like global query decomposition, global query optimization, distributed query processing, schema translation, merging data, etc. Not

only are these issues complicated but there is also an overhead attached to each. This makes the architecture complex and requires dealing with multiple wrappers, mediators, query processors, view managers, etc. Also, interleaving of these queries with the updates arriving from the base data sources may cause inconsistent views in a data warehouse. To avoid warehouse view inconsistency, rigid solutions like running each update and all actions needed to incrementally integrate it into the warehouse as a distributed transaction (spanning the warehouse and all the sources involved) must be adapted. This approach requires dealing with global concurrency control. Another less rigid approach would be to define new correctness and consistency algorithms [24].

4.3 Suitability of the Existing Approaches to the EDS Architecture

Complex query results are materialized as views to speed up applications that depend on these views. Algorithms for materialized view maintenance must balance the cost of view update against the query response time.

The philosophy that the data residing in the warehouse can be seen as a materialized view (or set of views) over the base data in the operational systems is not applicable to the EDS and the ODS architectures. A data warehouse is constructed for DSS informational processing and hence contains complicated, long running queries that access large amounts of data. Materializing queries for faster access is a suitable solution for data warehousing. On the other hand, an EDS is built for operational processing and contains short running queries that access limited amounts of data. Therefore, materializing queries may not be efficient for the EDS. Further, due to the volatile nature of the EDS, it may be difficult to balance the cost of view update against the query response time.

There are fundamental differences between the data warehouse and the EDS or the data warehouse and the ODS. These differences were discussed in Chapter 1. The first difference that weakens the materialized view philosophy is that the EDS is volatile

whereas the data warehouse is non-volatile. This means when a change occurs in an operational system, the EDS is updated in place to reflect the change. This further implies that a *record to record mapping* exists between the EDS and the operational systems. Record to record mapping is the mapping between a record in the EDS with the corresponding record in the operational system. This contrasts with the data warehouse architecture where changes in the operational systems are reflected by creating a new snapshot. In such systems, no record to record mapping is maintained between the data warehouse and the operational systems. Thus, the data in the EDS can be seen as a replica or a copy of the data in the legacy data sources rather than a materialized view over the base data. This differs from conventional replication techniques in that the replica is not an exact replica but a transformed replica. Synchronization of replicas becomes more complex as replication techniques not only have to deal with heterogeneity, autonomy, and distribution of data sources but also with complex data transformations.

The materialized view philosophy is further weakened by the very nature of data and processing found in the EDS. The data found in the EDS is current, detailed, and supports operational processing as compared to the data found in the data warehouse that is historical, summarized, and supports informational processing. Another kind of processing that is specific to the EDS and is not found in the data warehouse is the on-line transactional processing of the primary data and off-line transactional processing of the secondary data.

Unlike the data warehouse, the EDS has a dual propagation architecture. Updates on the primary data of the EDS are propagated to the operational systems. Considering the secondary data in the operational systems as views over the primary data in the EDS, a materialized view maintenance architecture is required for the operational systems. Unfortunately, operational systems can be legacy or other unsophisticated systems that do not understand views.

In spite of the complexities involved, the materialized view maintenance approach is a

viable solution for a data warehouse system. Because of the fundamental differences between the EDS and the data warehouse, the approach is not suitable for the EDS architecture. In this chapter a new approach to synchronization is proposed and compared to the materialized view approach. The new approach uses the metadata component of the data warehousing system for synchronization.

The rest of this chapter is organized as follows. Section 4.4 gives a formal classification of the types of data found in the EDS architecture. Section 4.5 proposes the architecture for synchronizing the EDS with the operational systems. It also compares the proposed architecture with the WHIPS (Warehouse Information Project at Stanford) system. Section 4.6 describes the merits of the proposed synchronization architecture. Finally Section 4.7 gives the liabilities of the proposed synchronization architecture.

4.4 Classification of Types of Data in the Two Tier Data Architecture

In this section a classification of types of data in the two tier data architecture is presented. This classification helps in clearly identifying the data in the two tiers that needs synchronization. This classification is illustrated in Figure 7.

As mentioned before, the application systems and the EDS are made up of two kinds of data - the primary and secondary data. One of the steps required for building the EDS is identifying the common business operations among the application systems (of the organization) whose processing should be moved to the EDS. Identification of the common business operations divides the data in an application system into the application specific data and data for the common business operations. The application specific data is the primary data of the application system (APD) and the common business operations data is the secondary data of the application system (ASD). Since, the EDS contains only

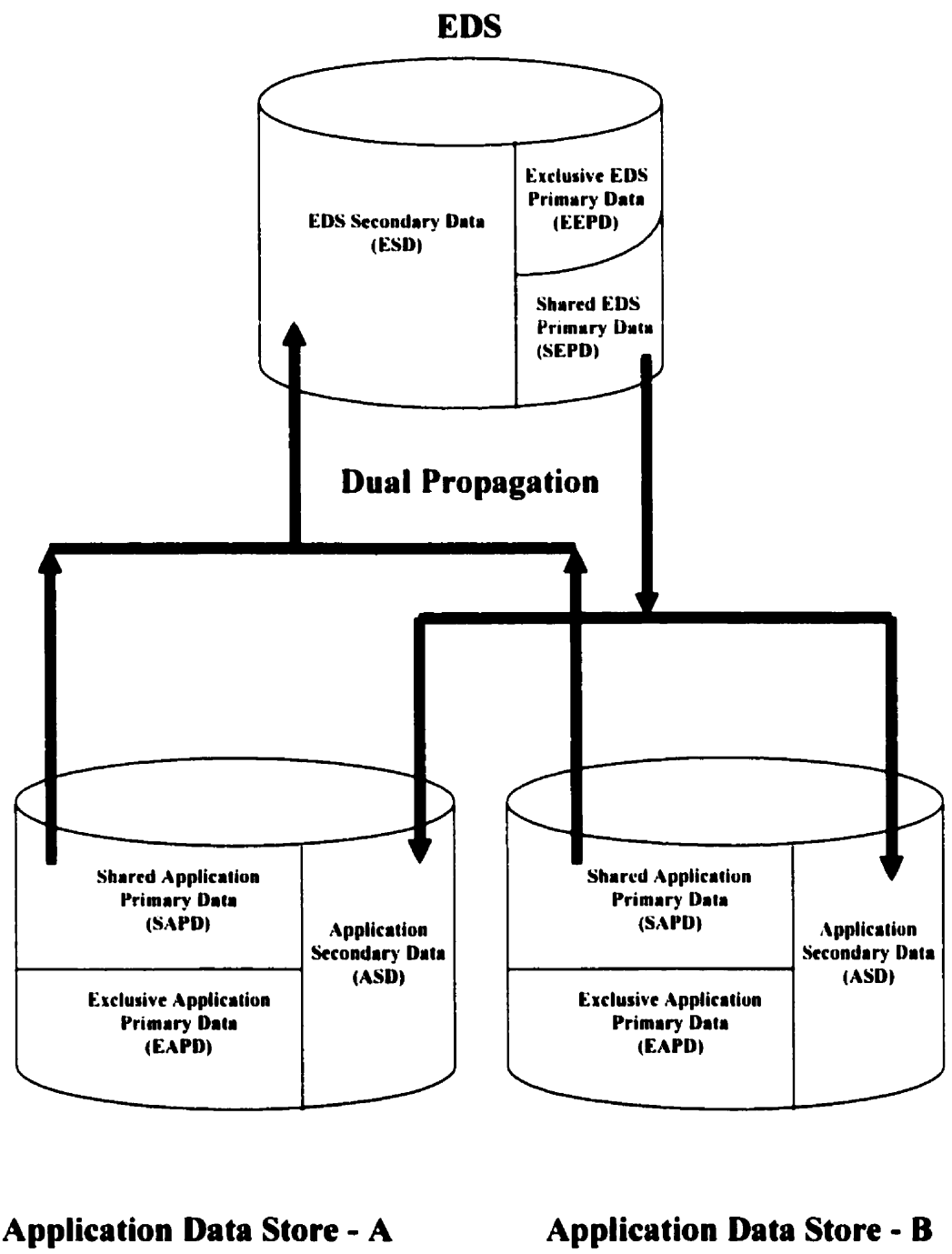

Figure 7 - Two Tier Data Architecture

a subset of the application primary data, it can be further classified into - the exclusive application primary data (EAPD) and the shared application primary data (SAPD). The EAPD is the data that only resides in the application system and is not extracted, transformed, and loaded into the EDS. The SAPD is the data that is extracted, transformed, and loaded into the EDS. The union of the SAPD and the ASD is the total data that is shared by an application system with the EDS.

Formally, as shown in Figure 8, data in an application system data store can be represented as :

Application System Data = APD ∪ ASD
= (EAPD ∪ SAPD) ∪ ASD
= EAPD ∪ (SAPD ∪ ASD)
= exclusive application data U shared data

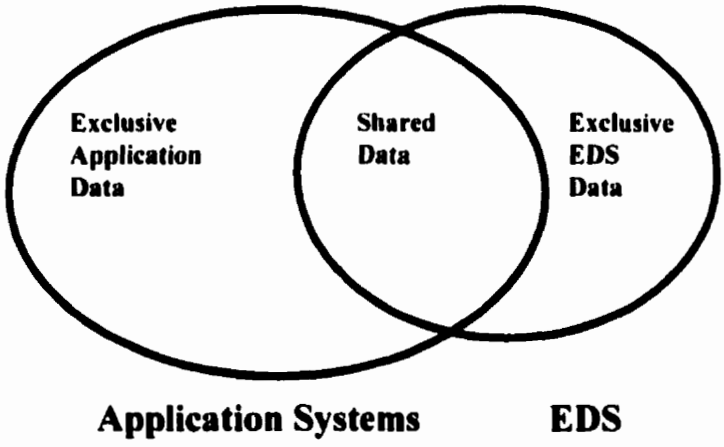

Figure 8 - Types of Data

The EDS is the primary source (or the system of record) for the common business operations. These common business operations are reengineered to support the new business model of the organization. The reengineering of the common business operations may result in storing some new data in the EDS that does not reside in any of the application systems. The EDS also contains a subset of data from the application systems that is extracted, transformed, integrated, and loaded into the EDS. In other words, data in the EDS can be divided into the reengineered common business operations' data and the integrated transformed application systems' data. The reengineered common business operations data is the primary data of the EDS (EPD) and the integrated transformed application systems data is the secondary data of the EDS (ESD). The EPD can be further classified into - the exclusive EDS primary data (EEPD) and the shared EDS primary data (SEPD). The EEPD is the data that only resides in the EDS and is a result of reengineering common business operations. The SEPD is the data belonging to the common business operations that is common to the EDS and the application systems. The union of the SEPD and the ESD is the total data that is shared by the EDS with the application systems.

As shown in Figure 8, data in the EDS can be represented as:

EDS Data = EPD $\cup$ ESD
= (EEPD $\cup$ SEPD) $\cup$ ESD
= EEPD $\cup$ (SEPD $\cup$ ESD)
= Exclusive EDS data $\cup$ shared data

Since updates can happen on the EDS primary data (EPD) and the application primary data (APD), the two tiers must be synchronized. As shown in Figure 7, the shared EDS primary data (SEPD) must be synchronized with the secondary data of application systems (ASDs); and the shared primary data of application systems (SAPDs) must be synchronized with the EDS secondary data (ESD). The two tier data architecture also

consists of the exclusive application primary data (EAPD) and exclusive EDS primary data (EEPD) in the two tiers respectively. The EAPD and EEPD, however, do not need to be synchronized.

As formally derived above, we can visualize the two tier data architecture as containing three kinds of data (Figure 8). They are - the exclusive application data, the exclusive EDS data and the shared data among the EDS and the application systems [3]. There is no synchronization needed for exclusive application data and exclusive EDS data. On the other hand, the data shared among the EDS and the application systems must be synchronized.

4.5 Architecture for Synchronizing the EDS with the Operational Systems.

Once the EDS has been loaded with the initial set of data obtained from the operational sources, the ongoing task is to keep the data synchronized.

In the previous section, the different kinds of data in the two tier data architecture were described. Since data modifications can happen on the primary data of the EDS (EPD) and the primary data of the application systems (APDs), a propagation mechanism is needed to keep the two tiers synchronized. Figure 9 illustrates the basic architecture for synchronizing the EDS with the operational systems.

There are two tiers of data in the EDS architecture. The first tier consists of data belonging to the operational systems and the second consists of data belonging to the EDS. For simplicity, in the architecture, two operational systems are considered but there can be numerous such operational systems. Traditional disk shapes are used to represent application data sources, in the general case these sources may include non-traditional data such as legacy systems, flat files, etc. There is a fair amount of heterogeneity among

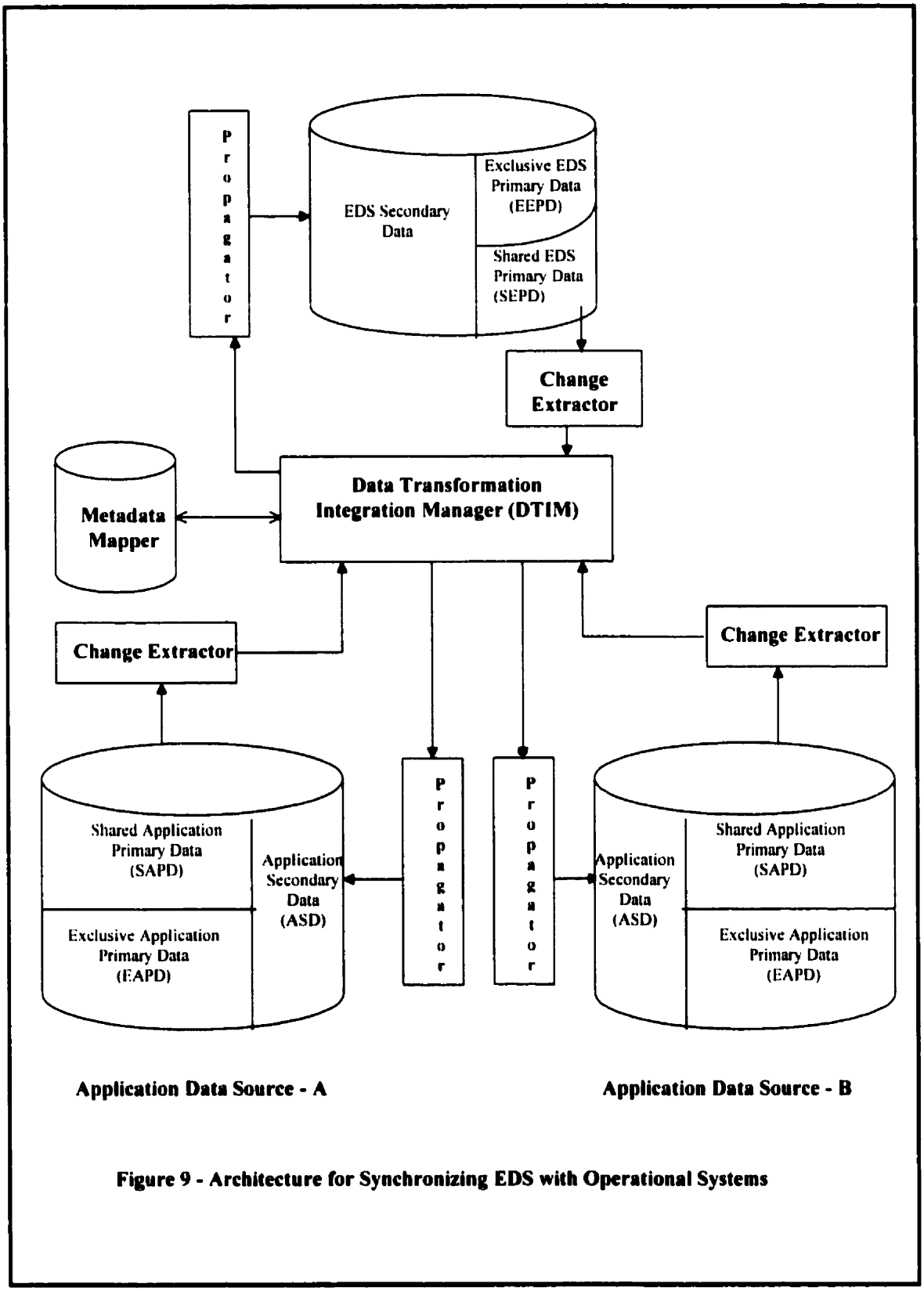

Figure 9 - Architecture for Synchronizing EDS with Operational Systems

the autonomous application data sources shown in the architecture [26,27].

Heterogeneity can occur in terms of hardware, data managers, query languages, data models, etc. Autonomy refers to the distribution of control. It indicates the degree to which individual application systems can operate independently. Although in Figure 9 a single, centralized EDS is illustrated, the EDS certainly may be implemented as a distributed database system [5]. In fact, data parallelism or distribution may be necessary to provide the necessary performance. The EDS can be implemented using state of the art database technology such as an object relational database management system (ORDBMS) like Oracle, Informix, etc.

Data modifications can happen to the shared enterprise primary data (SEPD) of the EDS as well as to the shared application primary data (SAPD) of the application systems. A change extractor is associated with the EDS as well as each application data store. It detects and captures data modifications to the data source with which it is associated. A different change extractor is needed for each application source, since the functionality of the change extractor is dependent on the type of source (database management system, legacy system, etc.) as well as the data provided by the source. As shown in Figure 9, the change extractor captures data modifications only with the shared primary data (SAPD or SEPD) of the associated data source. This way the change extractor can be optimized for detecting and capturing only the relevant information that is needed for synchronization.

Change detection is an open research problem that arises from the warehousing approach [19]. Since the EDS is a full-functionality database system, the change extractor can use active database capabilities [28] like triggers, rules etc., to extract the changes of interest. In the case of the application systems, the change detection is a much more difficult task. The changes are detected in legacy systems by inspecting the log files, modifying the application system to detect the changes of interest, or by using the utility programs to periodically dump and compare successive versions of files. In considering the change detection problem, the application sources have been classified into - cooperative sources,

logged sources, queryable sources, and snapshot sources [9,11].

The change extractor in the EDS architecture is analogous to the monitor component of the WHIPS system. The work done towards change detection in the WHIPS system can be applied to the EDS architecture. The WHIPS project implements trigger-based monitors for cooperative (relational) sources, and snapshot monitors for flat file sources that only provide periodic snapshots of the source data. The algorithms for efficient change detection using snapshots are discussed in [25].

Active research is being done on the problem of change detection. Even though this is a legitimate and interesting research question, it is not the one addressed in this thesis. This thesis assumes that algorithms for change detection exist. Further, the more pessimistic scenario is assumed where the change extractor is not sophisticated enough to detect and send only the relevant changes needed for synchronization to the data transformation integration manager (DTIM) layer. That is, the change extractor (as shown in Figure 9) is not optimized to detect and send only the changes on the shared primary data of the associated data source. All changes (secondary, exclusive primary, or shared primary) to the associated data store will be detected and sent by the change extractor to the DTIM layer.

Once the relevant changes from a data source in a tier are extracted, a mechanism to transform, integrate, and propagate those changes to the other tier is needed. The data transformation integration manager (DTIM) accepts the changes from a data source in one tier and generates the corresponding changes for the other tier. The logic needed to convert a transaction in one tier to the corresponding transaction(s) in the other tier is formulated by the DTIM using the metadata mapper component of the synchronization architecture (see Figure 9).

Metadata is one of the most important aspects of the data warehouse environment [10]. An important component of the data warehouse metadata store is the mapping between

the operational systems and the data warehouse. The typical contents of this mapping are the identification of source fields, simple attribute-to-attribute mappings, attribute conversions, physical characteristic conversions, naming changes, key changes, etc, [8]. In the EDS architecture the metadata mapper plays a major role in the synchronization of the two tiers. The mapping between the EDS and the application data stores is modeled in a metadata model. This mapping is then implemented in terms of relational tables, stored procedures, and functions. The metadata mapper component has the following advantages. 1) Scalability - the EDS is built in an iterative manner. As more data from the application data stores is brought into the EDS, the only change required is adding a suitable mapping in the mapper. 2) Simultaneous development of the metadata store - though metadata is an essential component of the data warehouse architecture, the development of this component is usually ignored. The proposed architecture enables the development of a major component of the metadata store and keeps it current with the data in the EDS and the operational systems.

The EDS synchronization architecture proposed in this thesis is based on two centralized components - the DTIM and the metadata mapper. These components contain all the knowledge needed to accept a change/update from one tier and convert it to the corresponding change(s)/update(s) in the other tier. No queries are posed on the operational systems so no additional information is required for the synchronization. This simplifies the architecture tremendously and the components like wrappers, mediators [19,20,21], and query processors that are a necessary and integral part of the WHIPS system [12] are not needed for the EDS architecture.

The philosophy behind the EDS synchronization architecture is to exploit the metadata component of the data warehouse system to drive the synchronization process. As mentioned above, a very important component of the data warehouse metadata store is the mapping between the operational systems and the data warehouse. If designed properly, the metadata store can contain all the information required to map a change in one tier to the corresponding change(s) in the other tier. As a result no queries need be

posed to the operational systems. This approach is quite different from the WHIPS approach that is based on materialized view maintenance.

In the WHIPS architecture all the information required to maintain the view is not stored in the data warehousing system. Hence, interaction is required with the operational systems in terms of sending queries and receiving answers to those queries. The EDS approach is to make the EDS system self sufficient by storing all the information required for synchronization in the metadata mapper. This minimizes interaction with the operational systems. Chapter 5 discusses, in detail, how the metadata mapper and the DTIM are used to synchronize the EDS with the operational systems. It also shows that the information in the metadata mapper is sufficient to accept a change and convert it to the corresponding change(s) in the other tier.

To summarize, the change extractor associated with a data source in a tier captures the changes and passes these changes to the DTIM layer. The DTIM layer uses the knowledge stored in the metadata mapper to generate the corresponding changes for the second tier. The DTIM then passes the generated changes to the propagator(s) of the second tier.

The propagator converts the logical transaction (changes generated by the DTIM layer) into the physical transaction (changes specific to a source). It then executes the transaction on the data source with which it is associated. Like the change extractor, a different propagator is required for each application source and for the EDS. This is because the functionality of the propagator is dependent on the type of the source (database system, legacy system, etc.) as well as the type of the data manager and the query language associated with the source. The functionality of the propagator can be compared to wrappers/translators [12,19,20,21] to a certain extent. A wrapper logically converts queries expressed over information in the common model into source-specific queries and commands. It also converts data returned by the source into the common model. Similarly, the propagator converts a logical transaction received by the DTIM

layer into a source-specific transaction. That is, it takes as input a transaction expressed in a generic format and converts it into the language of the associated source. The difference between a wrapper and a propagator is that a wrapper converts a query expressed in a common model whereas, the propagator converts a transaction expressed logically or in a generic format. The propagator converts transactions and the wrapper converts queries into the language of a source and as a result in the case of propagators, unlike wrappers, no data is returned from the source and hence no conversion of the returned data is required. Though the functionality of the propagator is a legitimate research area, it is not the focus of this research.

4.6 Merits of the Proposed Synchronization Architecture.

The classification of the data in our architecture clearly identifies the data that needs to be synchronized. The classification explicitly identifies the subsets of data in a data source with which the propagator and the change extractor should be associated.

Collisions occur when users simultaneously update the same data elements in both tiers [3]. Since in the proposed architecture a data element is only maintained by a single data source (primary data of a data source) and is read only in the other data source (secondary data of a data source) the possibility of collisions has been eliminated.

The proposed architecture facilitates both synchronous and asynchronous propagation. Updates in application systems can be propagated to the EDS in synchronous or asynchronous manner. Similarly, updates in the EDS can be propagated to the application systems synchronously or asynchronously. Like the ODS, the EDS can be further classified into Class I, II and III depending on the speed of refresh [3].

Data modifications to any tier must be transformed before they can be propagated to the other tier. The proposed architecture provides a sophisticated solution by storing the transformation logic in the metadata mapper. As mentioned before, the solution provides

scalability and simultaneous development of the metadata store. Tools can be developed that will automate or semi-automate the implementation of the metadata mapper. These tools will read the schema information from the data dictionaries of the data stores involved in the integration architecture. This information can then be presented to the user in a manner that allows the user to easily map entities and their respective attributes from one data store to the other. Based on the mapping, the tool will then produce the respective transformation logic needed for synchronization. Also, techniques and tools can be developed that will automate or semi-automate the process of implementing the change extractors and propagators through a tool kit or a specification based approach.

Before the data from disparate operational systems can be loaded in a data warehousing system (EDS or data warehouse) it must be transformed into a common model. Sometimes these transformations can be difficult, requiring complicated algorithmic conversions. The EDS architecture models these transformations and provides an easy and scalable solution for performing complicated transformations between the EDS and the operational systems.

The proposed architecture minimizes the interaction required with the operational systems required to achieve synchronization. Once the changes are extracted by the change extractor no queries need be sent to the operational systems for the extraction of any additional information. Thus, the proposed architecture is simpler than the WHIPS architecture as it does not have to deal with the issues of global query decomposition, global query optimization, distributed query processing, multi-source warehouse consistency and mediation.

The proposed solution also preserves the autonomy of the operational systems and the EDS. It does not lock data in the operational systems while the changes are propagated to the EDS or vice versa.

Since the proposed architecture stores all the information required for synchronization in

the metadata mapper, the warehouse anomaly that arises due to interleaving of queries with updates arriving from the base data sources is also eliminated in the proposed architecture. The next chapter discusses handling of the warehouse anomaly by the EDS architecture.

Finally, when propagating changes from one system to the other, when both systems are concurrently executing local transactions, mechanisms to ensure the serializability of local and external (propagated) transactions are required. In the proposed architecture such mechanisms are not needed because local transactions will act on the primary data and the external transactions will act on the secondary. Since they are two separate subsets of data, serializability is not an issue. The next chapter also discusses how this separation can be achieved at the attribute level.

4.7 Liabilities of the Proposed Synchronization Architecture.

Moving the common business operations processing to the EDS requires that existing application systems be changed. Application systems should be modified to prevent any insert, update and delete operations on the common business operations data. In other words, the data belonging to the common business operations in application systems should be read only and any screens or application code modifying this data must be changed. The EDS is the primary source (or the system of record) for the common business operations. An application should exist on the EDS to maintain the data belonging to the common business operations. Therefore, insert, update and delete operations on common business operations data must be performed on the EDS only.

Before the integration of application systems into the EDS architecture, application systems were autonomous. With the new architecture, there is a dependency in place between the application systems and the EDS. This dependency is the speed with which synchronization occurs between the EDS and the application systems. The dependency is more if the propagation from the EDS to the application systems is asynchronous. On the

other hand, the dependency is less if the propagation is synchronous. For example, as a part of common business operations processing, client processing has been moved to the EDS. A new client in the EDS is created and it is required to create new accounts for that client. Since new accounts will be created by the application system(s), the application system(s) is(are) dependent on when the new client information will be propagated from the EDS to the application system(s).

Chapter 5.

Synchronization Algorithms

In Chapter 4, an architecture for synchronizing the EDS with the operational systems was proposed and various components of the architecture were discussed in detail. The synchronization solution proposed in this chapter is based on two key components - the metadata mapper and the DTIM layer. This chapter explores these components in detail and illustrates the viability of the proposed synchronization solution that uses metadata for synchronization. The proposed solution uses the metadata model and the synchronization algorithms introduced in this chapter. The metadata model and the synchronization algorithms are implemented as the metadata mapper and the DTIM layer, respectively. This chapter also introduces a prototype for the metadata mapper and the DTIM layer that is based on simple mappings between the EDS and the operational systems. This prototype can be customized and expanded depending on the requirements of the organization.

The rest of the chapter is organized as follows. Section 5.1 introduces various mappings required for synchronization. Section 5.2 explores the metadata mapper and the DTIM layer in detail. It presents the metadata model and the synchronization algorithms. Section 5.3 demonstrates the synchronization algorithms using examples. Section 5.4 discusses how the proposed solution deals with the warehouse anomaly. Section 5.5 demonstrates the correctness of the synchronization algorithms.

5.1 What is needed for synchronization?

This section defines the various mappings required for synchronization. Four kinds of

mappings are required for synchronization : the *entity mapping*, the *attribute mapping*, the *key mapping and* the *record mapping*. In addition to these mappings, conversion algorithms are required to convert the value of an attribute in one system to its corresponding value in the other system. For example, consider an attribute 'gender' in a legacy data store defined as a character data type. This attribute stores 'M' for Male and 'F' for Female. The corresponding mapped attribute in the EDS called the 'gender code' is defined as a numeric datatype, and stores 1 for Male and 0 for Female. If an update happens on the attribute 'gender' in the legacy data store, along with the attribute to attribute mapping between the EDS and the legacy data store, a conversion algorithm that will change M to 1 and F to 0 is also needed.

Though the entity mapping and the attribute mapping are self explanatory, the key and record mapping require some explanation. The key mapping maps the primary key (PK) attributes between the EDS entities and the corresponding legacy system entities. The record mapping maps the tuples between the EDS entities and the corresponding legacy system entities. To illustrate this concept further, consider the three data stores shown in Figure 10 : the EDS, the legacy data store "Mortgage" and the legacy data store "Investment". The entity 'Client' in the EDS maps to the entity 'Customer' in the mortgage data store and the entity 'Investor' in the investment data store. Figure 10, shows the attributes and the primary key attributes (labeled PK) of these entities. Attribute 'clie_num' is the primary key attribute of the entity 'Client', 'cust_num' is the primary key attribute of the entity 'Customer', and 'invs_sin_num' (investor's sin number) is the primary key attribute of the entity 'Investor'. Also, assume for the purpose of this example that the EDS entity 'Client' is the primary source of the client information (primary entity) and the legacy entities 'Customer' and 'Investor' are the secondary sources (secondary entities). That means any inserts or updates to the client information are first performed in the EDS and then propagated to the mortgage and investment data store.

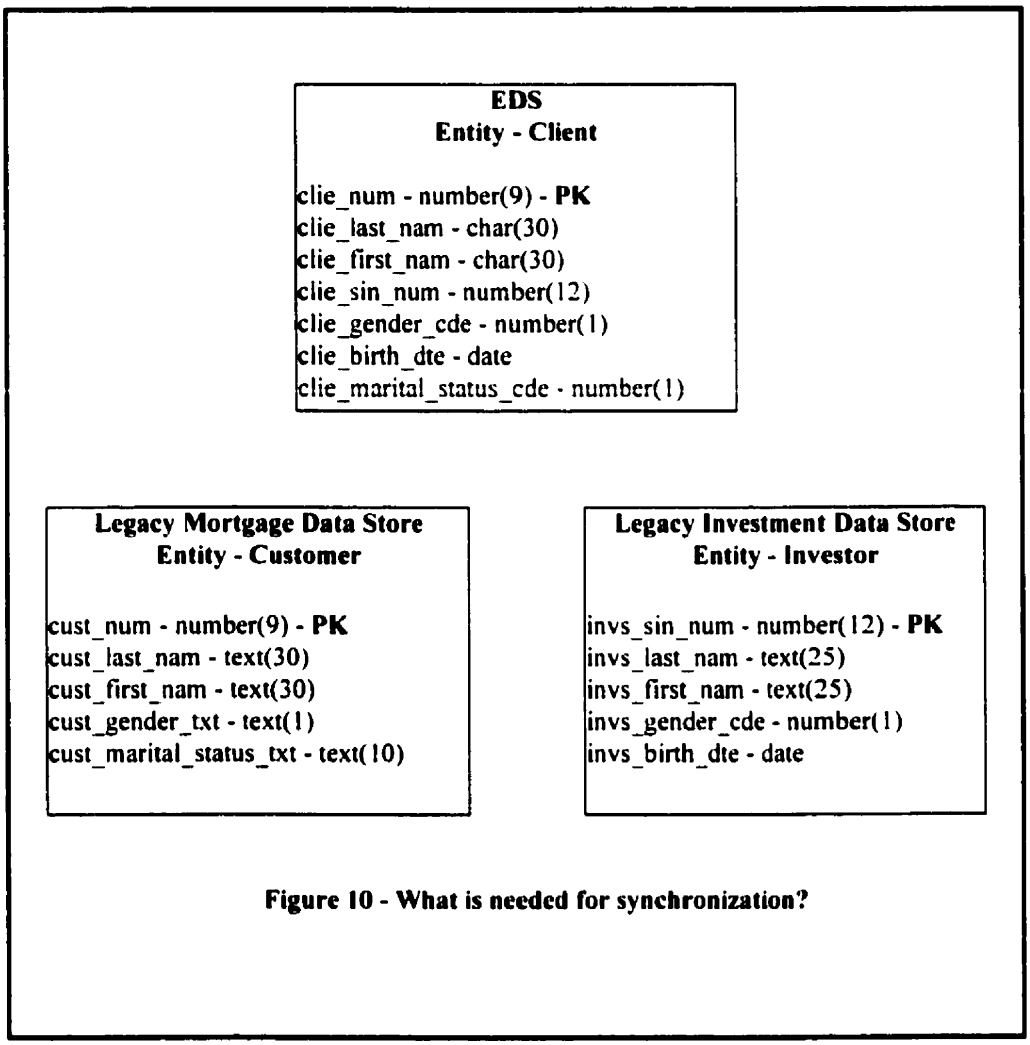

Figure 10 - What is needed for synchronization?

Consider an update on the entity 'Client' in the EDS -

UPDATE *client* ***SET*** *clie_last_nam = 'HARDY'* ***WHERE*** *clie_num = 1234.*

To synchronize this update, besides entity and attribute mappings, a record of the entity 'Client' in the EDS (clie_num = 1234) needs to be mapped to the corresponding record of the entity 'Customer' in the mortgage data store (cust_num = ?) and the corresponding record of the entity 'Investor' in the investment data store (invs_sin_num = ?). To achieve this, two kinds of information are needed. First, the primary key attributes of the respective systems i.e., 'clie_num' for 'Client', 'cust_num' for 'Customer', and 'invs_sin_num' for 'Investor' need to be determined. This piece of information is

referred to as *key to key mapping*. Second, the corresponding tuples of the respective systems that is, the client 1234 in the EDS maps to customer X in the mortgage data store and to investor Y in the investment data store must be determined. This piece of information is referred to as *record to record mapping or cross referencing*.

The record to record mapping is simpler for updates and deletes than inserts. From our example, updates and deletes are just a matter of a lookup from a table or a file that stores - the 'clie_num' 1234 in the EDS which maps to 'cust_num' X in the mortgage data store and 'invs_sin_num' Y in the investment data store. However, insert requires that corresponding keys for the entities 'Customer' and 'Investor' be generated before the mapping can be stored in the lookup table.

To illustrate this further, consider an insert of a new client in the EDS :

INSERT *into client (clie_num, clie_last_nam, clie_first_nam, clie_sin_num, clie_gender_cde, clie_birth_dte, clie_marital_status_cde)*
VALUES *(4567, 'Brown', 'Tom', 576890234, 1, 21/07/55, 1).*

To synchronize the insert on the entity 'Client' in the EDS with the mortgage and investment data stores a new customer and a new investor must be created in the two systems. This is achieved by the following mappings - the entity to entity mapping determines that the entity 'Client' in the EDS maps to the entity 'Customer' in the mortgage data store and to the entity 'Investor' in the investment data store. The attribute to attribute mapping maps the corresponding attributes between the EDS entity 'Client' and the mortgage entity 'Customer' as well as the investment entity 'Investor'. For example, the attribute to attribute mapping determines that the attribute 'clie_last_nam' of the entity 'Client' maps to the attribute 'cust_last_nam' of the entity 'Customer' as well as to the attribute 'invs_last_nam' of the entity 'Investor'. Conversion algorithms are also used to convert the value of 'clie_last_nam' into 'cust_last_nam' and 'invs_last_nam', respectively. The key to key mapping maps the primary key 'clie_num' of the entity 'Client' to the primary key 'cust_num' of the entity 'Customer' and to the

primary key 'invs_sin_num' of the entity 'Investor'. The record to record mapping must map 'clie_num = 4567' to 'cust_num = ?' in the mortgage data store and to 'invs_sin_num = ?' in the investment data store. To achieve this a key generation algorithm is needed to generate a unique value of 'cust_num'. On the other hand, no key generation algorithm is needed for 'invs_sin_num' as the value of the sin number is fixed and determined by the attribute to attribute mapping between the attribute 'clie_sin_num' in 'Client' and 'invs_sin_num' in 'Investor'. Hence, the value of sin number (clie_sin_num = 576890234) must be passed from the EDS as the corresponding value of 'invs_sin_num'. A generated value of 'cust_num' and passed value of 'invs_sin_num' are then stored in the look up table as record to record mappings.

5.2 Synchronization Logic Components

This section discusses the metadata mapper and the DTIM layer in detail. The four kinds of mappings introduced in the previous section are modeled in a metadata model. This model is then implemented as the metadata mapper. The following section discusses the metadata model in detail.

5.2.1. The Metadata Model

Figure 11 shows the metadata model for mapping between the legacy data stores and the EDS. In the two tier data architecture, one tier belongs to the EDS and the other tier belongs to the legacy data stores. The first question that needs to be addressed is - *what are the legacy data sources participating in the integration architecture?* The entity 'Legacy System' in Figure 11 stores the names of the legacy systems participating in the integration architecture. The second question to address is - *what are the entities / tables / files in the respective legacy systems?* The entity 'Legacy System Entities' stores all the entities belonging to the respective legacy systems. Similarly, the entity 'EDS Entities' stores all the entities belonging to the EDS. The entity - 'EDS Legacy Entity Mapping'

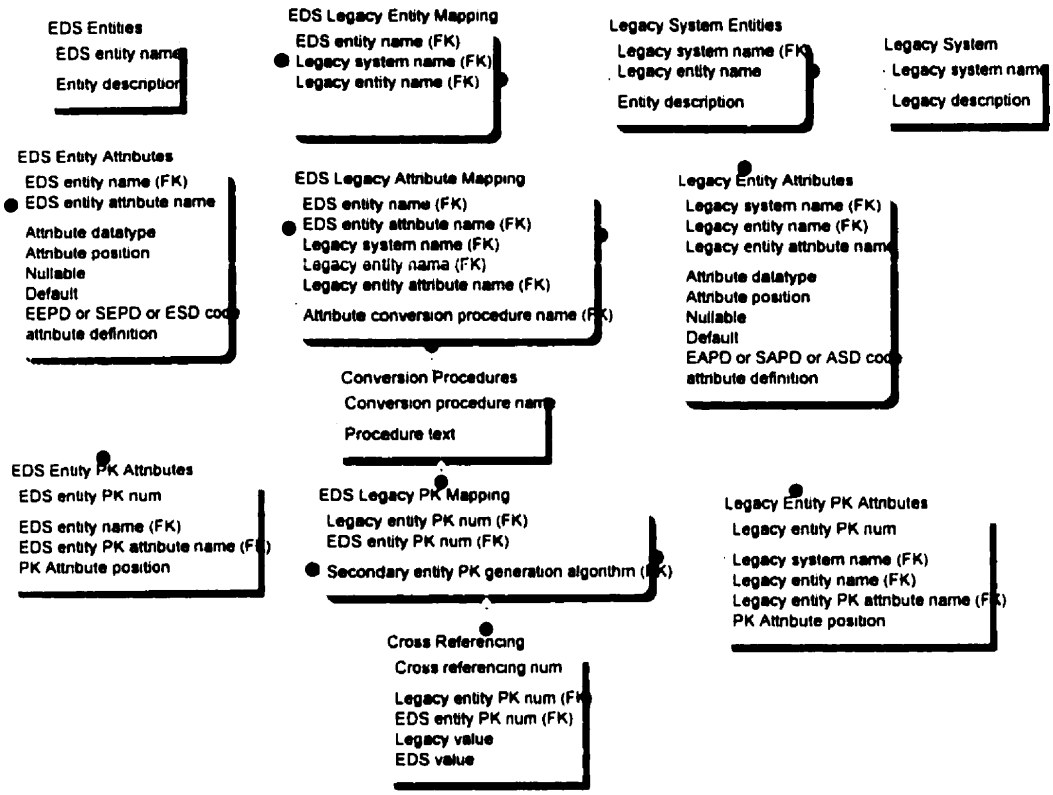

Figure 11 – Metadata Model

stores the mapping between the EDS entities and the legacy systems' entities for which the two tiers must be synchronized. Hence, the entity 'EDS Legacy Entity Mapping' stores the knowledge to perform entity to entity mapping between the EDS and the legacy systems.

Each entity may have many attributes but only some may require synchronization. The entity 'EDS Entity Attributes' stores all the attributes belonging to the respective entities in the EDS. With each attribute it stores its characteristics such as datatype and length, attribute definition, nullable, default, and position. Another important characteristic that

is stored with each attribute is the classification of the attribute to EEPD (Exclusive EDS Primary Data), SEPD (Shared EDS Primary Data) or ESD (EDS Secondary Data). In other words, if this attribute belongs to the exclusive EDS primary data, the shared EDS primary data or the EDS secondary data. Similarly, the entity 'Legacy Entity Attributes' stores all the attributes belonging to the respective entities in the legacy systems. Along with the characteristics of these attributes it also stores the classification of the attribute to EAPD (Exclusive Application Primary Data), SAPD (Shared Application Primary Data) or ASD (Application Secondary Data). This classification shows how a clear separation between the primary and the secondary data in the EDS and the legacy data stores is achieved.

The entity 'EDS Legacy Attribute Mapping' stores the mapping between the EDS entity attributes and the legacy entity attributes that need synchronization. With each mapping it also stores the name of the stored procedure required to convert the value of the primary attribute in one tier to its corresponding secondary attribute value(s) in the other tier. SQL or Advanced SQL scripts (e.g., Oracle's PL SQL) for these stored procedures are stored in the entity - 'Conversion Procedures'.

The Venn diagram shown in Figure 8, broadly characterizes the two tier data architecture as containing three kinds of data. The exclusive application data, the exclusive EDS data, and the data shared between the EDS and the legacy systems. The entity 'EDS Legacy Attribute Mapping' represents the shared data between the EDS and the legacy systems that must be synchronized. All the EDS attributes mapped with the legacy attributes in the entity 'EDS Legacy Attribute Mapping' must have SEPD or ESD code. Similarly, all the legacy attributes mapped with the EDS attributes in the entity 'EDS Legacy Attribute Mapping' must have SAPD or ASD code. Note that the SEPD attributes in the EDS must map to the ASD attributes in the legacy store and the ESD attributes in the EDS must map to the SAPD in the legacy store. 'EDS Entity Attributes' minus 'EDS Legacy Mapping Attributes' gives all the attributes belonging to the exclusive EDS data where no synchronization is needed. All these attributes must be characterized by EEPD code in the

entity 'EDS Entity Attributes'. Similarly 'Legacy Entity Attributes' minus 'EDS Legacy Mapping Attributes' gives all the attributes belonging to the exclusive legacy data where no synchronization is needed. All these attributes must be characterized by EAPD code in the entity 'Legacy Entity Attributes'.

Another important mapping needed for synchronization is the key mapping. The entity 'EDS Entity PK Attributes' stores the attributes forming the primary key for each entity in the EDS. Similarly, the entity 'Legacy Entity PK Attributes' stores the attributes forming the primary key for each entity in the legacy systems.

The entities 'EDS Legacy PK Mapping' and 'Cross Referencing' are responsible for storing the record to record mapping between the EDS and the legacy systems. The entity 'EDS Legacy PK Mapping' stores the mapping between the primary key attributes of the primary entities in one tier with the corresponding primary key attributes of the secondary entities in the other tier. With each mapping it also stores the name of the stored procedure required to generate the primary key value for the secondary entity. SQL or advanced SQL scripts (e.g., Oracle's PL SQL) for the stored procedures that are needed to generate the primary key values for the secondary entities are stored in the entity 'Conversion Procedures'. The entity 'Cross Referencing' stores, for each primary-secondary PK mapping, the corresponding tuple mapping. In other words, it stores the mapping between each tuple of the primary entity in one tier to the corresponding tuple(s) of the secondary entity(s) in the other tier.

5.2.2 The Metadata Mapper

The metadata model just described above is implemented in terms of relational tables using state of art database technology such as an object relational database management system (ORDBMS) like Oracle, Informix, etc. Each entity in the model corresponds to a table in the database. These tables are then populated with the data pertaining to the four kinds of mappings between the EDS and the operational systems including the SQL

scripts required for attribute to attribute and primary key to primary key conversions. The SQL or Advanced SQL (e.g., Oracle's PL SQL) scripts stored in the entity 'Conversion Procedures' are then implemented as stored procedures in the database. The tables and stored procedures together constitute the metadata mapper. Determination and population of mapping data and conversion functions in these tables is a subject of ongoing research but is not the focus of this thesis. As mentioned in Chapter 4, tools can also be developed that will automate or semi automate the implementation of the metadata mapper.

5.2.3 The Data Transformation Integration Manager

This section introduces the synchronization algorithms that constitute the DTIM layer. The synchronization algorithms proposed in this section are based on simple mappings between the EDS and the operational systems. These algorithms are based on one to one mappings between a primary entity in one tier and the secondary entity in the other tier or one to many mappings between a primary entity in one tier and secondary entities in the other tier. To support many to one or many to many mappings between primary entities in one tier with the secondary entity(s) in the other tier, changes are required to the proposed algorithms. Though the characteristics of the EDS discourage such mappings, the algorithms can be easily extended to support them if required. Similarly, complicated cases of attribute to attribute and, PK to PK mappings may also require changes to the proposed algorithms. The purpose of this research is only to build a core framework for synchronization of the EDS with the operational systems. The resulting framework can be easily expanded and modified to support more complicated requirements at a later date.

As mentioned in Chapter 4, this research assumes a pessimistic scenario in that the change extractor is not sophisticated enough to detect only the changes that are needed for synchronization (i.e., changes on only the shared primary data). The change extractor detects and captures all changes to the secondary data, the exclusive primary data, and the shared primary data of the data source with which it is associated. After detection, the

change extractor sends the following parameters to the DTIM layer :

P_System_name - Name of the system EDS or application data store with which the change extractor is associated.

P_Entity_name - Name of the table or entity on which data modification occurred.

P_Operation_type - Type of operation – insert (I), update (U) or delete (D)

P_Key_list - Set of the key attribute(s) and their value(s).

P_Attribute_list- Set of other modified attribute(s) and their value(s). For insert operations, P_attribute_list will also include the key attribute(s) and their value(s). This is needed for cases where PK attributes of one data store map to PK or non PK attributes of the other data store. The values of such attributes are determined through attribute to attribute mapping between the two data stores. Therefore, key attribute(s) and their value(s) are included in the P_Attribute_list.

These parameters collectively represent data modification on the primary data of a system and are referred to as the *input list*. The output of the synchronization algorithms are list(s) with list_name = System_name ||[2] Entity_name created for each system name and entity name that need synchronization as a result of data modification on the primary data of a system. Each list is populated with the corresponding system name, entity name, operation type, Attribute_list, and Key_list. These lists are also referred to as *output lists*. The structure of an output list is as follows :

---

[2] The symbol || signifies concatenation of two variables or constants.

Output_list : System_name || Entity_name ← System_name, Entity_name,
P_Operation_type, Attribute_list, Key_list

Where Attribute_list stores the attributes and their values for the system name and entity name that need synchronization. Similarly, Key_list stores the primary key attributes and their values for the system name and entity name that need synchronization.

For easier manipulation of these lists, two functions may be defined that will be used by the synchronization algorithms described next.

Function GET_VALUE_INLIST (attribute) – This function takes as input an attribute name or a PK attribute name and returns the corresponding value for that attribute in the P_Attribute_list or P_Key_list. For example, consider P_Attribute_list ← clie_first_nam : 'David', clie_last_nam : 'Brown'. The function call GET_VALUE_INLIST (clie_last_nam) will return 'Brown'. Similarly, consider P_Key_list ← clie_num : 3456. The function call GET_VALUE_INLIST (clie_num) will return 3456.

Function GET_VALUE_OUTLIST (system name, entity name, PK attribute name) – This function searches the Key_list in the output list with the output list name = System_name || Entity_name to determine the corresponding PK value for the PK attribute name.

5.2.3.1 Synchronization Algorithms

Algorithm 1 : Perform_synchronization

This algorithm synchronizes the EDS with the operational systems. It takes as input data modifications detected and passed by the change extractor on the primary data of one tier and produces the corresponding data modifications on the secondary data of the other tier. To achieve synchronization, the algorithm performs four kinds of mappings between the EDS and the operational systems. They are - entity to entity, attribute to attribute, key to

key, and record to record mappings as described earlier.

The algorithm Perform_synchronization is formally presented in Figure 12. Input to the algorithm consists of – the primary system name, the primary entity name, the operation type (insert/update/delete), a set of key attributes and their values, and a set of modified attributes and their values. These parameters collectively represent modification on the primary data of a system and are referred to as an *input list*. The algorithm checks the input parameter 'primary system name' (P_System_name) on which data modification occurred. If data modification occurred on the EDS (line 1, Figure 12) three algorithms - Perform_EDS_entity_mapping (Algorithm 2), Perform_EDS_attribute_mapping (Algorithm 3), and Perform_EDS_PK_and_record_mapping (Algorithm 4) are called respectively to perform entity to entity, attribute to attribute, key to key, and record to record mapping between the EDS and the legacy systems (lines 2-4, Figure 12). Also, if the data modification on the EDS is an insert or a delete operation the algorithm Maintain_EDS_cross_referencing (Algorithm 5) is executed to determine and store (or determine and delete) the appropriate record to record mappings between the EDS and the legacy systems (lines 5-6, Figure 12).

If the data modification occurred on the legacy system instead of the EDS the corresponding set of algorithms Perform_legacy_entity_mapping (Algorithm 6), Perform_legacy_attribute_mapping (Algorithm 7), Perform_legacy_PK_and_record_mapping (Algorithm 8) and Maintain_legacy_cross_referencing (Algorithm 9) are called by the algorithm Perform_synchronization (lines 7-11, Figure 12).

As just described there are two sets of algorithms called by Perform_synchronization. If the data modification occurs on an EDS entity the Algorithms 2 to 5 are executed. If the data modification occurs on a legacy entity Algorithms 6 to 9 are executed. Since, the functionality of both these sets of algorithms is quite similar, only Algorithms 2 to 5 are

Figure 12 - Perform Synchronization

Algorithm 1 : Perform_synchronization

input : A data modification (Insert/Update/Delete) on the primary data of a system (EDS or Application Data Store). The change extractor will capture the operation and pass the following parameters to the DTIM layer.

P_System_name - Primary system name;
P_Entity_name - Primary entity name;
P_Operation_type - Type of operation (Insert, Update, Delete);
P_Key_list - Set of key attribute(s) and its(their) value(s);
P_Attribute_list - Set of modified attribute(s) and its(their) value(s).
For inserts, this list will also include the key attribute(s) and its(their) value(s);

These parameters collectively represent a modification on the primary data of a system and are collectively referred to as *input list.*

output : Corresponding data modification(s) (Insert/Update/Delete) on the secondary data of the system(s).

```
begin
        if P_System_name = 'EDS' then                                   (1)
            Perform_EDS_entity_mapping                                  (2)
                (P_Entity_name, P_Operation_type);
            Perform_EDS_attribute_mapping                               (3)
                (P_Entity_name, P_Attribute_list);
            Perform_EDS_PK_and_record_mapping                           (4)
                (P_Entity_name, P_Operation_type, P_Key_list);
            if P_Operation_type = 'I' or 'D' then                       (5)
                Maintain_EDS_cross_referencing                          (6)
                        (P_Entity_name, P_Key_list, output_list(s));
            end if; {if P_Operation_type}
        else
            Perform_legacy_entity_mapping                               (7)
                (P_System_name, P_Entity_name, P_operation_type);
            Perform_legacy_attribute_mapping                            (8)
                (P_System_name, P_Entity_name, P_Attribute_list);
            Perform_legacy_PK_and_record_mapping                        (9)
                (P_System_name, P_Entity_name,
                 P_Operation_type, P_Key_list);
            if P_Operation_type = 'I' or 'D' then                       (10)
                Maintain_legacy_cross_referencing                       (11)
                        (P_System_name, P_Entity_name,
                         P_Key_list, output_list(s));
            end if; {if P_Operation_type}
        end if; {if P_System_name}
end; {End Perform_synchronization}
```

described. Thus, assume the EDS is the system on which data modification occurred i.e., P_System_name = 'EDS'. Thus Perform_synchronization calls the algorithms: Perform_EDS_entity_mapping, Perform_EDS_attribute_mapping, Perform_EDS_PK_and_record_mapping, and Maintain_EDS_cross_referencing for synchronizing the EDS with the operational systems. Discussion on these algorithms follows. The first algorithm that is called or executed by Perform_synchronization is Perform_EDS_entity_mapping (Figure 13).

Algorithm 2 : Perform_EDS_entity_mapping

This algorithm performs entity to entity mapping between the EDS and the Legacy systems. The algorithm Perform_EDS_entity_mapping is formally presented in Figure 13. It takes as input the EDS entity on which data modification occurred and finds the corresponding legacy system names and their entities that need synchronization. To perform this mapping, selection is performed on the entity - 'EDS Legacy Entity Mapping' of the metadata mapper where the attribute 'EDS entity name' equals the input parameter primary entity name (P_Entity_name) (lines 1-2, Figure 13). The parameter P_Entity_name stores the name of the EDS entity on which data modification occurred.

For each Legacy system name and Legacy entity name returned by the selection a list is initialized with the list name = Legacy system name || Legacy entity name (lines 6-7, Figure 13). In other words, a list is initialized for each Legacy system name and Legacy entity name that need synchronization as a result of data modification on the EDS entity. These lists are also referred to as *output lists*. Each list is then populated with the corresponding Legacy system name, Legacy entity name, operation type (P_Operation_type), an empty Attribute_list, and an empty Key_list (line 8, Figure 13). Attribute_list in a <Legacy system name || Legacy entity name list> will be used to store the legacy entity attributes and their values that need to be synchronized as a result of the data modification on the EDS entity. Similarly, Key_list in a <Legacy system name ||

Figure 13 - Perform EDS Entity Mapping

Algorithm 2 : Perform_EDS_entity_mapping

input : P_Entity_name; // EDS entity on which data modification occurred //
P_Operation_type;

output : A list containing the system name, entity name, and operation type for each legacy system name and entity name that needs to be synchronized.

```
var Attribute_list ← ϕ; // empty list to store the corresponding mapped legacy
                          entity attributes //
    Key_list ← ϕ; // empty list to store the corresponding mapped legacy
                          entity key attributes //
begin
        T ← {t_i | t_i ∈ EDS Legacy Entity Mapping ∧ A(t_i),                    (1)
                A (predicate) = (EDS entity name = P_Entity_name)};
        T[Legacy system name, Legacy entity name]                               (2)
          ← {Legacy system name_i ‖ Legacy entity name_i
              | t_i[Legacy system name] || t_i[Legacy entity name] ∧ t_i ∈ T};
        if T = ϕ then                                                           (3)
            display 'No synchronization is needed.                              (4)
             The change belongs to EEPD';
            exit;                                                               (5)
        end if; {if T}
        for each tuple(t_i) in T[Legacy system name, Legacy entity name]        (6)
            initialized a list with list name =                                 (7)
             Legacy system name_i ‖ Legacy entity name_i ;
            add Legacy system name_i, Legacy entity name_i,                     (8)
                  P_Operation_type, Attribute_list, Key_list
            to list with list name = Legacy system name_i ‖ Legacy entity name_i ;
        end for; {for t_i}
end; {End Perform_EDS_entity_mapping}
```

Figure 14 - Perform Legacy Entity Mapping

Algorithm 6 : Perform_legacy_entity_mapping

input : P_System_name; // Legacy system name on which data modification occurred //
P_Entity_name; // Legacy system entity name on which data modification occurred //
P_Operation_type;

output : A list containing the EDS system name, entity name, and operation type for each EDS entity that needs to be synchronized.

var Attribute_list :← ϕ; // empty list to store the corresponding mapped EDS entity attributes //
Key_list ← ϕ; // empty list to store the corresponding mapped EDS entity key attributes //

begin

T ← {t_i | t_i ∈ EDS Legacy Entity Mapping ∧ A(t_i), (1)
A (predicate) = (Legacy entity name = P_Entity_name
∧ Legacy system name = P_System_name)};
T[EDS entity name] (2)
← {EDS entity name$_i$ | t_i[EDS entity name] ∧ t_i ∈ T};
if T = ϕ **then** (3)
display 'No synchronization is needed. (4)
The change belongs to EAPD';
exit; (5)
end if; {if T}
for each tuple(t_i) in T[EDS entity name] (6)
initialized a list with list name = 'EDS' ‖ EDS entity name$_i$; (7)
add 'EDS', EDS entity name$_i$, P_Operation_type, (8)
Attribute_list, Key_list
to list with list name = 'EDS' ‖ EDS entity name$_i$;
end for; {for t_i}
end; {End Perform_legacy_entity_mapping}

Legacy entity name list> will be used to store the legacy entity key attributes and their values that need to be synchronized as a result of the data modification on the EDS entity The contents of Attribute_list and Key_list are determined by Perform_EDS_attribute_mapping (Figure 15) and Perform_EDS_PK_and_record_mapping (Figure 17) respectively.

Note that if the data modification occurs on the legacy system instead of the EDS, the algorithm Perform_legacy_entity_mapping (Figure 14) is instead called by Perform_synchronization. As mentioned before, the functionality of this algorithm is similar to Perform_EDS_entity_mapping. However, this algorithm finds the EDS entity names that need to be synchronized as a result of data modification on the legacy entity instead of the opposite.

The algorithm Perform_EDS_entity_mapping returns control to line 3 of Perform_synchronization after performing entity to entity mapping between the EDS and the operational systems. Perform_synchronization then calls the algorithm Perform_EDS_attribute_mapping (Figure 15).

Algorithm 3 : Perform_EDS_attribute_mapping

The algorithm Perform_EDS_attribute_mapping performs *attribute to attribute mapping* between the EDS entity and the mapped legacy system entities. The algorithm Perform_EDS_attribute_mapping is formally presented in Figure 15. It takes as input the modified EDS entity attributes and their values (P_Attribute_list) and finds the mapped legacy entity attributes and their values that need synchronization. The algorithm determines the content of the empty Attribute_list that is part of each <Legacy system name || Legacy entity name list> determined by the algorithm Perform_EDS_entity_mapping. To determine the content of the empty Attribute_list the algorithm first determines the attributes in P_Attribute_list that belong to the shared EDS

Figure 15 - Perform EDS Attribute mapping

Algorithm 3 : Perform_EDS_attribute_mapping

input : P_Entity_name; // EDS entity name on which data modification occurred //
P_Attribute_list - {A_i:V_i | A is a set of modified attributes,
V is a set of their corresponding values};
// Modified EDS entity attributes and their values //

output : A list containing system name, entity name, operation type, attribute name(s), and attribute value(s) for each legacy system name and entity name that needs to be synchronized.

Var Legacy_entity_attribute_value ← ϕ;
EDS_entity_attribute_value ← ϕ;
begin

```
Q ← {q_i | q ∈ EDS Entity Attributes ∧ C(q),                                (1)
        C (predicate) = (EDS entity name = P_Entity_name
                          ∧ EDS entity attribute name ∈ A_i)};
Q[EDS entity name, EDS entity attribute name, code]                         (2)
   ← {EDS entity name_i || EDS entity attribute name_i || code_i
      | q_i[EDS entity name] || q_i[EDS entity attribute name] || q_i[code]
       ∧ q_i ∈ Q};
for each tuple(q_i) in Q[EDS entity name, EDS entity attribute name, code]   (3)
   if code_i = 'ESD' then                                                   (4)
      display 'Error data modification cannot happen on                     (5)
       EDS secondary data';
   else if code_i = 'EEPD' then
      display 'No synchronization needed. Change belongs to EEPD';          (6)
   else if code_i = 'SEPD' then
      R ← {r_i | r ∈ EDS Legacy Attribute Mapping ∧ D(r),                   (7)
            D (predicate) = (EDS entity name = EDS entity name_i
                              ∧ EDS entity attribute name =
                                EDS entity attribute name_i)};
       R[EDS entity attribute name, Legacy system name,                     (8)
         Legacy entity name, Legacy entity attribute name,
         Attribute conversion procedure name]
         ← {EDS entity attribute name_i ||
            Legacy system name_i || Legacy entity name_i ||
            Legacy entity attribute name_i ||
            Attribute conversion procedure name_i,
            |r_i[EDS entity attribute name] ||
             r_i[Legacy system name] || r_i[Legacy entity name] ||
             r_i[Legacy entity attribute name] ||
             r_i[Attribute conversion procedure name]
             ∧ r_i ∈ R};
       for each tuple (r_i) in                                              (9)
       R[EDS entity attribute name, Legacy system name,
         Legacy entity name, Legacy entity attribute name,
         Attribute conversion procedure name]
          Legacy_entity_attribute_value ← ϕ;                                (10)
          EDS_entity_attribute_value ← ϕ;                                   (11)
```

```
                    EDS_entity_attribute_value ← GET_VALUE
                                                    (EDS entity attribute name,);          (12)
                    if Attribute conversion procedure name, = ϕ then                       (13)
                        Legacy_entity_attribute_value ←                                    (14)
                              EDS_entity_attribute_value;
                    else
                        Legacy_entity_attribute_value ←                                    (15)
                              Attribute conversion procedure name,(
                                                    EDS_entity_attribute_value);
                    end if; {Attribute conversion procedure name}
                    S ← {s,| s ∈ Legacy Entity PK Attributes ∧ E(s),                       (16)
                              E (predicate) = (Legacy system name =
                                                  Legacy system name,
                                                  ∧ Legacy entity name =
                                                  Legacy entity name,
                                                  ∧ Legacy entity PK attribute name
                                                  = Legacy entity attribute name,)};
                    if S = ϕ then                                                          (17)
                    add Legacy entity attribute name,,                                     (18)
                        Legacy_entity_attribute_value
                      to Attribute_list in list with list name =
                              Legacy system name, ‖ Legacy entity name,;
                    else
                    add Legacy entity attribute name,,                                     (19)
                            Legacy entity attribute value
                      to Key_list in list with list name = Legacy system name,
                                                  ‖ Legacy   entity name,;
                  end if; {if S}
                end for; {for r,}
            end if; {if code}
        end for; {for q,}
end; {End Perform_EDS_attribute_mapping}
```

Figure 16 - Perform Legacy Attribute Mapping

Algorithm 7 : Perform_legacy_attribute_mapping

input : P_System_name; // Legacy system name on which data modification occurred //
P_Entity_name; // Legacy entity name on which data modification occurred //
P_Attribute_list - {A_i:V_j | A is a set of modified attributes,
V is a set of their corresponding values};
// Modified legacy entity attributes and values //

output : A list containing system name, entity name, operation type, attribute name(s), and attribute value(s) for each EDS entity that needs to be synchronized.

```
var Legacy_entity_attribute_value ← ϕ;
    EDS_entity_attribute_value ← ϕ;
begin
        Q ← {q_i | q ∈ Legacy Entity Attributes ∧ C(q),                    (1)
                C (predicate) = (Legacy system name = P_System_name
                                  ∧ Legacy entity name = P_Entity_name
                                  ∧ Legacy entity attribute name ∈ A_j)};
        Q[Legacy system name, Legacy entity name,                          (2)
          Legacy entity attribute name, code]
          ← {Legacy system name_i ‖ Legacy entity name_i ‖
             Legacy entity attribute name_i || code_i
             | q_i[Legacy system name] ‖ q_i[Legacy entity name] ‖
               q_i[Legacy entity attribute name] ‖ q_i[code] ∧ q_i ∈ Q};
        for each tuple(q_i) in Q[Legacy system name, Legacy entity name,   (3)
                                 Legacy entity attribute name, code]
            if code_i = 'ASD' then                                         (4)
               display 'Error data modification cannot happen on           (5)
                application secondary data';
            else if code_i = 'EAPD' then
               display 'No synchronization needed. Change belongs to EAPD'; (6)
            else if code_i = 'SAPD' then
               R ← {r_i | r ∈ EDS Legacy Attribute Mapping ∧ D(r),          (7)
                     D (predicate) = (Legacy system name =
                                        Legacy system name_i
                                      ∧ Legacy entity name =
                                        Legacy entity name_i
                                      ∧ Legacy entity attribute name =
                                        Legacy entity attribute name_i)};
               R[EDS entity name, EDS entity attribute name,               (8)
                 Legacy entity attribute name,
                 Attribute conversion procedure name]
                 ← {EDS entity name_i ‖ EDS entity attribute name_i ‖
                    Legacy entity attribute name_i ‖
                    Attribute conversion procedure name_i
                    | r_i[EDS entity name] ‖
                      r_i[EDS entity attribute name] ‖
                      r_i[Legacy entity attribute name] ‖
                      r_i[Attribute conversion procedure name] ∧ r_i ∈ R};
```

```
                for each tuple (r_i) in                                          (9)
                R[EDS entity name, EDS entity attribute name,
                  Legacy entity attribute name,
                  Attribute conversion procedure name]
                   EDS_entity_attribute_value ← ϕ;                               (10)
                   Legacy_entity_attribute_value ← ϕ;                            (11)
                   Legacy_entity_attribute_value ←                               (12)
                              GET_VALUE(Legacy entity attribute name_i );
                   if Attribute conversion procedure name_i = ϕ then             (13)
                        EDS_entity_attribute_value =                             (14)
                                   Legacy_entity_attribute_value;
                   else
                        EDS_entity_attribute_value ←                             (15)
                                   Attribute conversion procedure name_i
                                              (Legacy_entity_attribute_value);
                   end if; {Attribute conversion procedure name}
                   S ← {s_i | s ∈ EDS Entity PK Attributes ∧ E(s),                (16)
                            E (predicate) = ( EDS entity name =
                                                 EDS entity name_i
                                                 ∧ EDS entity PK attribute name
                                              = EDS entity attribute name_i)};
                   if S = ϕ then                                                 (17)
                   add EDS entity attribute name_ii,                             (18)
                          EDS_entity_attribute_value
                     to Attribute_list in list with list name =
                                                      'EDS' || EDS entity name_i;
                   else
                   add EDS entity attribute name_ii, EDS_entity_attribute_value  (19)
                     to Key_list in list with list name =
                                                      'EDS' || EDS entity name_i;
                   end if; {if S}
                end for; {for r_i}
             end if; {if code}
          end for; {for q_i}
end; {End Perform_legacy_attribute_mapping}
```

primary data (SEPD) and therefore need to be synchronized with the legacy systems. To achieve this, a selection is performed on the entity 'EDS Entity Attributes' of the metadata mapper where the attribute 'EDS entity name' equals the input parameter P_Entity_name and the attribute 'EDS entity attribute name' belongs to the input parameter P_Attribute_list (lines 1-2, Figure 15). The parameter P_Entity_name stores the name of the EDS entity on which data modification occurred. The parameter P_Attribute_list stores the modified EDS entity attributes and their values.

For each EDS attribute returned by the selection and belonging to SEPD code (lines 3-4, Figure 15), the corresponding mapped legacy attributes that need synchronization are determined. This is achieved by selecting from the entity 'EDS Legacy Attribute Mapping' that stores the mapping between the EDS entity attributes and the corresponding legacy entity attributes (lines 7-8, Figure 15). The entity 'EDS Legacy Attribute Mapping' along with the attribute to attribute mapping also stores the reference to the stored procedure name required to convert the value of an EDS attribute to the corresponding value of the legacy attribute. For each mapped legacy attribute (line 9, Figure 15) the corresponding legacy value is determined by executing the stored procedure associated with the attribute to attribute mapping (lines 10-15, Figure 15). The mapped legacy entity attributes and their values are then assigned to the Attribute_list of the corresponding <Legacy system name ‖ Legacy entity name list>. Before assigning each legacy attribute name and its value to the Attribute_list the algorithm checks whether the legacy attribute is a non primary key attribute of the legacy entity or not. This is determined by performing a selection on the entity 'Legacy Entity PK Attributes' for the legacy attribute (line 16, Figure 15). This is required because the EDS entity attribute may map to the primary key attribute of the legacy entity or to a non primary key attribute. If the legacy attribute maps to the primary key attribute of the legacy entity it is inserted in the Key_list of the corresponding <Legacy system name ‖ Legacy entity name list> (line 19, Figure 5) otherwise it is inserted, in the Attribute_list of the corresponding <Legacy system name ‖ Legacy entity name list> (line 18, Figure 15).

Note that if the data modification occurs on the legacy system instead of the EDS the algorithm Perform_legacy_attribute_mapping (Figure 16) is instead called by Perform_synchronization. The functionality of this algorithm is similar to Perform_EDS_attribute_mapping. However, the algorithm finds the EDS attributes and their values that need to be synchronized as a result of data modification on the legacy entity instead of the opposite.

After performing attribute to attribute mapping between the EDS and the operational systems the algorithm Perform_EDS_attribute_mapping returns control to line 4 of Perform_synchronization. Perform_synchronization then calls the algorithm Perform_EDS_PK_and_record_mapping (Figure 17).

Algorithm 4 : Perform_EDS_PK_and_record_mapping

The algorithm performs *key to key and record to record mapping* between the EDS entity and the mapped legacy system entities. The algorithm Perform_EDS_PK_and_record_mapping is formally presented in Figure 17. It takes as input the primary key attributes and values (P_Key_list) of the EDS entity and determines the primary key attributes and values for the mapped legacy entity(s). The algorithm determines the content of the empty Key_list that is part of each <Legacy system name || Legacy entity name list> determined by the algorithm Perform_EDS_entity_mapping. To determine the content of the Key_list the algorithm first reads the primary key attributes of the modified EDS entity. This is achieved by performing selection on the entity 'EDS Entity PK Attributes' in the metadata mapper where the attribute 'EDS entity name' equals the input parameter P_Entity_name and the attribute 'EDS entity PK attribute name' belongs to the input parameter P_Key_list (lines 1-2, Figure 17). The parameter P_Entity_name stores the name of the EDS entity on which data modification occurred. The parameter P_Key_list stores the EDS entity's PK attributes and their values. The rest

Figure 17 - Perform EDS Primary Key and Record Mapping

Algorithm 4 : Perform_EDS_PK_and_record_mapping

input : P_Entity_name; // EDS entity name on which data modification occurred //
P_Operation_type;
P_Key_list - {K_i:V_i | K is a set of PK attribute(s),
V is a set of their corresponding value(s)};
// Primary key attribute(s) and value(s) of the EDS entity //

output . A list containing system name, entity name, operation type, attribute name(s), attribute value(s) , primary key name(s), and primary key value(s) for each legacy system name and entity name that needs to be synchronized.

```
var Legacy_entity_PK_value ← ϕ;
begin
        P ← {p_i | p ∈ EDS Entity PK Attributes ∧ C(q),                    (1)
                C (predicate) = (EDS entity name = P_Entity_name
                                ∧ EDS entity PK attribute name ∈ K_i)};
        P[EDS entity PK num, EDS entity PK attribute name]                 (2)
          ← {EDS entity PK num_i , EDS entity PK attribute name_i
              | p_i[EDS entity PK num] ||
                p_i[EDS entity PK attribute name] ∧ p_i ∈ P};
        if P_operation_type = 'I' then                                     (3)
           for each tuple(p_i) in                                          (4)
              P[EDS entity PK num, EDS entity PK attribute name]
              M ← {m_i | n_i || l_i ∧ n ∈ EDS Legacy PK Mapping             (5)
                              ∧ l ∈ Legacy Entity PK Attributes ∧ C(q),
                        C (predicate) =
                                (EDS Legacy PK Mapping.Legacy entity PK num =
                                 Legacy Entity PK Attributes.Legacy entity PK num
                                 ∧ EDS Legacy PK Mapping.EDS entity PK num =
                                 EDS entity PK num_i )};
              M[Legacy system name, Legacy entity name,                    (6)
                 Legacy entity PK attribute name,
                 Secondary entity PK generation algorithm]
                 ← { Legacy system name_i || Legacy entity name_i ||
                     Legacy entity PK attribute name_i ||
                     Secondary entity PK generation algorithm_i,
                       | m_i[Legacy system name] || m_i[Legacy entity name] ||
                         m_i[Legacy entity PK attribute name] ||
                         m_i[Secondary entity PK generation algorithm] ∧ m_i ∈ M};
              for each tuple(m_i) in                                       (7)
                 M[Legacy system name, Legacy entity name,
                     Legacy entity PK attribute name,
                     Secondary entity PK generation algorithm]
                     if Secondary entity PK generation algorithm_i <> ϕ then   (8)
                        Legacy_entity_PK_value ← ϕ;                        (9)
                        Legacy_entity_PK_value ←                           (10)
                             Secondary entity PK generation algorithm_i();
                        add Legacy entity PK attribute name_i,             (11)
```

```
                                Legacy_entity_PK_value
                            to Key_list in list with list name
                                = Legacy system name_i || Legacy entity name_i;
                        end if; {if Secondary entity PK generation algorithm}
                    end for; {for m_i}
                end for; {for p_i}
            else // {operation type is not insert}
                for each tuple(p_i) in                                          (12)
                    P[EDS entity PK num, EDS entity PK attribute name]
                    M ← {m_i | n_i || l_i ∧ n ∈ Cross Referencing                (13)
                                ∧ l ∈ Legacy Entity PK Attributes ∧ C(q),
                            C (predicate) ≡
                                (Cross Referencing.Legacy entity PK num =
                                 Legacy Entity PK Attributes.Legacy entity PK num
                                 ∧ Cross Referencing.EDS entity PK num =
                                 EDS entity PK num_i
                                 ∧ EDS value =
                                 GET_VALUE_INLIST
                                    (EDS entity PK attribute name_i)};
                    M[Legacy system name, Legacy entity name,                   (14)
                      Legacy entity PK attribute name, Legacy value]
                      ← { Legacy system name, || Legacy entity name, ||
                            Legacy entity PK attribute name, || Legacy value,
                     | m_i[Legacy system name] || m_i[Legacy entity name] ||
                      m_i[Legacy entity PK attribute name] ||
                      m_i[Legacy value] ∧ m_i ∈ M};
                    for each tuple(m_i) in                                      (15)
                        M[Legacy system name, Legacy entity name,
                          Legacy entity PK attribute name, Legacy value]
                            add Legacy entity PK attribute name_n               (16)
                                Legacy value,
                            to Key_list in list with list name
                             = Legacy system name_i || Legacy entity name_i;
                    end for; {for m_i}
                end for; {for p_i}
            end if; {if P_Operation_type}
        end; {End Perform_EDS_PK_and_record_mapping}
```

Figure 18 - Perform Legacy Primary Key and Record Mapping

Algorithm 8 : Perform_Legacy_PK_and_record_mapping

input : P_System_name // Legacy system name on which data modification occurred //
P_Entity_name; // Legacy entity name on which data modification occurred //
P_Operation_type;
P_Key_list - {K_i:V_i | K is a set of PK attribute(s),
V is a set of their corresponding value(s)};
// Primary key attribute(s) and value(s) of the legacy entity //

output : A list containing system name, entity name, operation type, attribute name(s), attribute value(s), primary key name(s), and primary key values(s) for each EDS entity that needs to be synchronized.

var EDS_entity_PK_value ← ϕ;
begin

```
P ← {p_i | p ∈ Legacy Entity PK Attributes ∧ C(q),                    (1)
        C (predicate) = (Legacy system name = P_System_name
                          ∧ Legacy entity name = P_Entity_name
                          ∧ Legacy entity PK attribute name ∈ K_i)};
P[Legacy entity PK num, Legacy entity PK attribute name]              (2)
  ← {Legacy entity PK num_i , Legacy entity PK attribute name_i
     | p_i[Legacy entity PK num] ||
       p_i[Legacy entity PK attribute name] ∧ p_i ∈ P};
if P_operation_type = 'I' then                                        (3)
   for each tuple(p_i) in P[Legacy entity PK num,                     (4)
                            Legacy entity PK attribute name]
     M ← {m_i | n_i || l_i ∧ n ∈ EDS Legacy PK Mapping                (5)
                  ∧ l ∈ EDS Entity PK Attributes ∧ C(q),
            C (predicate) =
                  (EDS Legacy PK Mapping.EDS entity PK num =
                   EDS Entity PK Attributes.EDS entity PK num
                   ∧ EDS Legacy PK Mapping.Legacy entity PK num =
                   Legacy entity PK num_i )};
     M[EDS entity name, EDS entity PK attribute name,                 (6)
       Secondary entity PK generation algorithm]
        ← {EDS entity name_i || EDS entity PK attribute name_i ||
           Secondary entity PK generation algorithm_i
         | m_i[EDS entity name] ||
           m_i[EDS entity PK attribute name] ||
           m_i[Secondary entity PK generation algorithm] ∧ m_i ∈ M};
     for each tuple(m_i) in                                           (7)
       M[EDS entity name, EDS entity PK attribute name,
         Secondary entity PK generation algorithm]
         if Secondary entity PK generation algorithm_i <> ϕ then      (8)
           EDS_entity_PK_value ← ϕ;                                   (9)
           EDS_entity_PK_value ←                                      (10)
                Secondray entity PK generation algorithm_i();
          add EDS entity PK attribute name_i, EDS_entity_PK_value     (11)
          to Key_list in list with list name = 'EDS' || EDS entity name_i ;
```

```
                end if; {if Secondary entity PK generation algorithm}
              end for; {for m_i}
            end for; {for p_i}
        else {operation type is not insert}
            for each tuple(p_i) in                                              (12)
                P[Legacy entity PK num, Legacy entity PK attribute name]
                M ← {m_i | n, || l, ∧ n ∈ Cross Referencing                     (13)
                                ∧ l ∈ EDS Entity PK Attributes ∧ C(q),
                        C (predicate) =
                                (Cross Referencing.EDS entity PK num =
                                EDS Entity PK Attributes.EDS entity PK num
                                ∧ Cross Referencing.Legacy entity PK num =
                                Legacy entity PK num,
                                ∧ Legacy value =
                                GET_VALUE_INLIST
                                    (Legacy entity PK attribute name_i)};
                M[EDS entity name, EDS entity PK attribute name, EDS value]     (14)
                    ← { EDS entity name_i || EDS entity PK attribute name_i
                        || EDS value_i
                        | m_i[EDS entity name] ||
                          m_i[EDS entity PK attribute name] ||
                          m_i[EDS value] ∧ m_i ∈ M};
                  for each tuple(m_i) in                                        (15)
                  M[EDS entity name, EDS entity PK attribute name, EDS value]
                        add EDS entity PK attribute name_i, EDS value_i         (16)
                        to Key_list in list with list name = 'EDS' || EDS entity name_i;
                  end for; {for m_i}
            end for; {for p_i}
        end if; {if P_Operation_type}
end; {End Perform_Legacy_PK_and_record_mapping}
```

of the functionality of the algorithm is based on the type of data modification (insert/update/delete) that occurred on the EDS entity. As discussed before, record to record mapping for updates and deletes requires a lookup from a table or file that stores the mapping between primary keys of the EDS entities and the corresponding legacy entities. However, insert operations require the corresponding keys for the mapped entity to first be generated before the mapping can be stored in the lookup table.

The algorithm Perform_EDS_PK_and_record_mapping (Figure 17) checks the input parameter P_Operation_type to determine if it is an insert operation (line 3, Figure 17). If it is an insert operation then for each EDS primary key attribute (line 4, Figure 17) the corresponding mapped legacy PK attributes are determined. This is achieved by selecting from two entities; 'EDS Legacy PK Mapping' and 'Legacy Entity PK Attributes' (lines 5-6, Figure 17). The entity 'EDS Legacy PK Mapping' stores the mapping between primary key attributes of the EDS entity and the corresponding primary key attributes of the legacy entities. This entity along with the PK to PK mapping also stores the reference to the stored procedure name required to generate the primary key value of the mapped legacy PK attribute. The entity 'Legacy Entity PK Attributes' stores the primary key attributes of the Legacy entities. For each mapped legacy PK attribute (line 7, Figure 17) the corresponding legacy PK value is determined by executing the stored procedure associated with the PK to PK mapping (lines 8-10, Figure 17). The mapped legacy entity PK attributes and their values are then assigned to the Key_list of the corresponding <Legacy system name ‖ Legacy entity name list> (line 11, Figure 17).

If the operation is not an insert then for each EDS primary key attribute (line 12, Figure 17) the corresponding mapped legacy PK attributes and their values are determined. This is achieved by selecting from two entities; 'Legacy Entity PK Attributes' and 'Cross Referencing' (lines 13-14, Figure 17). The entity 'Legacy Entity PK Attributes' stores the primary key attributes of the Legacy entities. The entity 'Cross referencing' stores the mapping between tuples of the EDS entity with the corresponding tuples of the legacy entities. The mapped legacy entity PK attributes and their values are then assigned to the

Key_list of the corresponding <Legacy system name || Legacy entity name list> (lines 15-16, Figure 17).

Note that if the data modification occurs on the legacy system instead of the EDS the algorithm Perform_legacy_PK_and_record_mapping (Figure 18) is called by Perform_synchronization instead. The functionality of this algorithm is similar to Perform_EDS_PK_and_record_mapping but, the algorithm finds the EDS entity PK attributes and their values that need to be synchronized as a result of data modification on the legacy entity.

The algorithm Perform_EDS_PK_and_record_mapping returns control to line 5 of Perform_synchronization after performing key to key and record to record mapping between the EDS and the operational systems. If P_Operation_type is an insert or a delete operation then Perform_synchronization calls the algorithm Maintain_EDS_cross_referencing (Figure 19) as well.

Algorithm 5 : Maintain_EDS_cross_referencing

This algorithm maintains record to record mappings between the EDS and the legacy systems. The algorithm Maintain_EDS_cross_referencing is formally presented in Figure 19. It reads the input list (data modifications on the the EDS entity) and output lists (corresponding data modifications on the legacy entities determined by the synchronization algorithms) and then determines and stores or determines and deletes record to record mappings between the EDS PK values and the legacy PK values for insert and delete operations. The algorithm does not support update operations on primary keys of the EDS entities. If required, the algorithm can be modified to support such cases.

The algorithm first reads the primary key attributes of the modified EDS entity. This is achieved by performing a selection on the entity 'EDS Entity PK Attributes' of the metadata mapper where the attribute 'EDS entity name' equals the input parameter

Figure 19 - Maintain EDS Cross Referencing

Algorithm 5 : Maintain_EDS_cross_referencing

input : *input_list* : P_Entity_name;
P_Key_list - {K_i:V_i | K is a set of PK attribute(s),
V is a set of their corresponding value(s)};
output_list : Legacy system name$_i$ || Legacy entity name$_i$ ←
Legacy system name$_{i1}$, Legacy entity name$_{i1}$,
P_Operation_type, Attribute_list, Key_list

output : Record to record mapping between EDS PK values and the corresponding legacy PK values.

var Legacy_value ← ϕ;
EDS_value ← ϕ;
begin
P ← {p_i | p ∈ EDS Entity PK Attributes ∧ C(q), (1)
C (predicate) = (EDS entity name = P_Entity_name
∧ EDS entity PK attribute name ∈ K_i)};
P[EDS entity PK num, EDS entity PK attribute name] (2)
← {EDS entity PK num$_i$, EDS entity PK attribute name$_i$
| p_i[EDS entity PK num] ‖
p_i[EDS entity PK attribute name] ∧ p_i ∈ P};
for each tuple(p_i) in P[EDS entity PK num, EDS entity PK attribute name] (3)
M ← {m_i | n, || l, ∧ n ∈ EDS Legacy PK Mapping (4)
∧ l ∈ Legacy Entity PK Attributes ∧ C(q),
C (predicate) =
(EDS Legacy PK Mapping.Legacy entity PK num =
Legacy Entity PK Attributes.Legacy entity PK num
∧ EDS Legacy PK Mapping.EDS entity PK num =
EDS entity PK num$_i$)};
M[Legacy system name, Legacy entity name, (5)
Legacy entity PK attribute name,
Legacy entity PK num]
← { Legacy system name, ‖ Legacy entity name, ‖
Legacy entity PK attribute name, ‖
Legacy entity PK num,
| m_i[Legacy system name] ‖ m_i[Legacy entity name] ‖
m_i[Legacy entity PK attribute name] ‖
m_i[Legacy entity PK num] ∧ m_i ∈ M};
for each tuple(m_i) in (6)
M[Legacy system name, Legacy entity name,
Legacy entity PK attribute name,
Legacy entity PK num]
Legacy_value ← ϕ; (7)
EDS_value ← ϕ; (8)
EDS_value ← GET_VALUE_INLIST (9)
(EDS entity PK attribute name$_i$);
Legacy_value ← (10)
GET_VALUE_OUTLIST (Legacy system name$_{i1}$,

```
                    Legacy entity name_i, Legacy entity PK attribute name_i);
                if P_Operation_type = 'I' then                                    (11)
                    insert into cross_referencing                                 (12)
                    values(Legacy entity PK num_i, EDS entity PK num_i,
                            Legacy_value, EDS_value);
                else
                    delete from cross_referencing                                 (13)
                    where Legacy entity PK num = Legacy entity PK num_i AND
                            EDS entity PK num = EDS entity PK num_i AND
                            Legacy value = Legacy_value AND
                            EDS value = EDS_value;
                 end if; {if P_Operation_type}
            end for; {for m_i}
        end for; {for p_i}
end; {End Maintain_EDS_Cross_refeencing}
```

Figure 20 - Maintain Legacy Cross Referencing

Algorithm 9 : Maintain_legacy_cross_referencing

input : *input_list* : P_System_name;
P_Entity_name;
P_Key_list - {K_i:V_i | K is a set of PK attribute(s),
V is a set of their corresponding value(s)};
output_list : EDS || EDS entity name$_i$ ←
EDS, EDS entity name$_{i+1}$, P_Operation_type,
Attribute_list, Key_list

output : Record to record mapping between legacy PK values and the corresponding EDS PK values.

var Legacy_value ← ϕ;
EDS_value ← ϕ;
begin

P ← {p_i | p ∈ Legacy Entity PK Attributes ∧ C(q), (1)
C (predicate) = (Legacy system name = P_System_name
Legacy entity name = P_Entity_name
∧ Legacy entity PK attribute name ∈ K_i)};

P[Legacy entity PK num, Legacy entity PK attribute name] (2)
← {Legacy entity PK num$_i$, Legacy entity PK attribute name$_i$
| p_i[Legacy entity PK num] ||
p_i[Legacy entity PK attribute name] ∧ p_i ∈ P};

for each tuple(p_i) in P[Legacy entity PK num, (3)
Legacy entity PK attribute name]

M ← {m_i | n_i || l_i ∧ n ∈ EDS Legacy PK Mapping (4)
∧ l ∈ EDS Entity PK Attributes ∧ C(q),
C (predicate) =
(EDS Legacy PK Mapping.EDS entity PK num =
EDS Entity PK Attributes.EDS entity PK num
∧ EDS Legacy PK Mapping.Legacy entity PK num
= Legacy entity PK num$_i$)};

M[EDS entity name, (5)
EDS entity PK attribute name,
EDS entity PK num]
← { EDS entity name$_i$ ||
EDS entity PK attribute name$_i$ ||
EDS entity PK num$_i$
| m_i[EDS entity name] ||
m_i[EDS entity PK attribute name] ||
m_i[EDS entity PK num] ∧ m_i ∈ M};

for each tuple(m_i) in (6)
M[EDS entity name,
EDS entity PK attribute name,
EDS entity PK num]
Legacy_value ← ϕ; (7)
EDS_value ← ϕ; (8)
Legacy_value ← (9)

```
                    GET_VALUE_INLIST (Legacy entity PK attribute name_j) ;
                EDS_value ←                                                    (10)
                    GET_VALUE_OUTLIST ('EDS',
                        EDS entity name_n, EDS entity PK attribute name_i);
                if P_Operation_type = 'I' then                                 (11)
                    insert into cross_referencing                              (12)
                    values(Legacy entity PK num_j, EDS entity PK num_i,
                        Legacy_value, EDS_value);
                else
                    delete from cross_referencing                              (13)
                    where Legacy entity PK num = Legacy entity PK num_j AND
                        EDS entity PK num = EDS entity PK num_i AND
                        Legacy value = Legacy_value AND
                        EDS value = EDS_value;
                end if; {if P_Operation_type}
            end for; {for m_n}
        end for; {for p_i}
end; {End Maintain_legacy_cross_referencing}
```

P_Entity_name and the attribute 'EDS entity PK attribute name' belongs to the input parameter P_Key_list (lines 1-2, Figure 19). The parameter P_Entity_name stores the name of the EDS entity on which data modification occurred. The parameter P_Key_list stores the EDS entity PK attributes and their values. For each EDS primary key attribute (line 3, Figure 19) the corresponding mapped legacy PK attributes are determined. This is achieved by selecting from two entities; 'EDS Legacy PK Mapping' and 'Legacy Entity PK Attributes' (lines 4-5, Figure 19). The entity 'Legacy Entity PK Attributes' stores the primary key attributes of the Legacy entities. The entity 'EDS Legacy PK Mapping' stores the mapping between PK attributes of the EDS entity and the corresponding primary key attributes of the legacy entities. For each EDS entity PK attribute and the mapped legacy PK attribute determine their corresponding PK values from the input list and output list(s) respectively (lines 6-10, Figure 19). If the parameter P_Operation_type is an insert operation the record to record mapping is inserted in the entity 'Cross Referencing' otherwise it is deleted from the entity 'Cross Referencing' (lines 11-13, Figure 19).

Note that if the data modification occurs on the legacy system instead of the EDS the algorithm Maintain_legacy_cross_referencing (Figure 20) is called by Perform_synchronization instead. As mentioned before, the functionality of this algorithm is similar to Maintain_EDS_cross_referencing. The algorithm, however, maintains record to record mapping between the legacy system and the EDS; the input list contains data modification on legacy entity and the output lists contain corresponding data modification on the EDS entities determined by the synchronization algorithms.

The algorithm Maintain_EDS_cross_referencing returns control to the end of Perform_synchronization after maintaining record to record mapping between the EDS and the operational systems.

The algorithm Perform_synchronization terminates leaving the EDS synchronized with

the legacy systems.

5.3. Examples

In this section the synchronization algorithms are illustrated using some examples. For the purpose of this section consider three data stores; the EDS, the legacy data store 'Investment', and the legacy data store 'Mortgage' as shown in Figure 10. The entity 'Client' in the EDS maps to the entity 'Customer' in the mortgage data store and the entity 'Investor' in the investment data store. Assume, the EDS entity 'Client' is the primary source of the client information and the entities 'Customer' and 'Investor' are the secondary sources. Any data modifications will first be performed in the EDS and then propagated to the investment data store and the mortgage data store. Figure 10 shows the attributes and primary key attributes of these entities. Appendix 1 shows the mappings between the EDS entity 'Client' and the legacy entity 'Customer' and between the EDS entity 'Client' and the legacy entity 'Investor' stored by the metadata mapper. The stored procedures to convert attributes between the EDS and the legacy systems are also included in Appendix 1. The following examples show how the mappings are used to synchronize the EDS entity 'Client' with the legacy entities 'Customer' and 'Investor'.

5.3.1 Example 1

Consider an insert on the entity 'Client' in the EDS.

INSERT INTO *client(clie_num, clie_last_nam, clie_first_nam, clie_sin_num, clie_gender_cde, clie_birth_dte, clie_marital_status_cde)*

VALUES *(4567, 'Rimmer', 'Rodger', 508133212, '1', '25/05/60', 1);*

The change extractor associated with the EDS captures the insert operation and passes the following parameters to the DTIM layer:

P_System_name ← *'EDS'*
P_Entity_name ← *Client*
P_Operation_type ← *'I'*

P_Key_list ← clie_num, 4567
P_Attribute_list ← clie_num, 4567,
clie_last_nam : 'Rimmer',
clie_first_nam : 'Rodger',
clie_sin_num : 508133212,
clie_gender_cde : 1,
clie_birth_dte : '25/05/60',
clie_marital_status_cde : 1

The DTIM layer that is made up of synchronization algorithms, executes the algorithm Perform_synchronization with the above parameters.

Perform_synchronization (Algorithm 1, Figure 12)

Since the condition in line 1 that checks if P_System_name = 'EDS' is true, the algorithm *Perform_EDS_entity_mapping* is executed. This algorithm performs entity to entity mapping between the EDS and the legacy systems.

Perform_EDS_entity_mapping (Algorithm 2, Figure 13)

This algorithm determines the legacy systems and their entities that need to be synchronized as a result of the insert on the EDS entity 'Client'.

Lines 1 and 2 of this algorithm determine these entities to be 'Customer' in the 'Mortgage system' and 'Investor' in the 'Investment system'.

Lines 6,7 and 8 of the algorithm create two lists, one for each system name and entity name with the following contents:

<Mortgage system || Customer list> ← Mortgage system, Customer, 1, Attribute_list, Key_list;

<Investment system || Investor list> ← Investment system, Investor, 1, Attribute_list, Key_list;

Note that Attribute_list and Key_list are initially empty. It will be shown shortly how

their contents are determined by the attribute and the primary key mapping algorithms.

Control then returns to line 3 of Perform_synchronization.

Line 3 calls the algorithm *Perform_EDS_attribute_mapping*. This algorithm performs attribute to attribute mapping between the EDS and the legacy systems. The main purpose of this algorithm is to determine the contents of the Attribute_list in the <Mortgage system ‖ Customer list> as well as the Attribute_list in the <Investment system ‖ Investor list>.

Perform_EDS_attribute_mapping (Algorithm 3, Figure 15)

This algorithm determines the legacy system(s) attributes that need to be synchronized as a result of the insert on the EDS entity 'Client'.

Lines 1 and 2 of the algorithm select from the metadata mapper table 'EDS Entity attributes' all attributes that belong to the EDS entity 'Client' that are in P_Attribute_list. The selection creates a result set Q that has the following rows:

Result Set Q

Row num	EDS entity name	EDS entity attribute name	Code
1	Client	clie_num	EEPD
2	Client	clie_last_nam	SEPD
3	Client	clie_first_nam	SEPD
4	Client	clie_sin_num	SEPD
5	Client	clie_gender_cde	SEPD
6	Client	clie_birth_dte	SEPD
7	Client	clie_marital_status_cde	SEPD

Line 3 loops through each row in the result set Q. For each row in the result set Q, lines 4,5 and 6 determine if the EDS entity attribute has the SEPD code (shared EDS primary data). If the EDS attribute does have the SEPD code, lines 7 and 8 determine the corresponding mapped legacy attributes (result set R). The result set R for row num 2 of the result set Q is shown below. Note since row num 1 of the result set Q does not belong to SEPD code therefore, no attribute mapping is performed for this attribute.

Result Set R for row num 2 of the Result Set Q

Row num of set Q	Row num of set R	EDS entity attribute name	Legacy system name	Legacy entity name	Legacy entity attribute name	Attribute conversion procedure name
2	1	clie_last_nam	Mortgage system	Customer	cust_last_nam	
2	2	clie_last_nam	Investment system	Investor	invs_last_nam	truncate_last_nam

Line 9 then loops through each row in R. For each row in R, lines 13,14 and 15 determine the corresponding value of the legacy entity attribute. Lines 16 and 17 then determine whether the legacy entity attribute is a PK attribute or a non PK attribute. If it is a PK attribute it is added to the Key_list of the corresponding <Legacy system name ‖ Legacy entity name list> (line 19) otherwise the attribute is added to the Attribute_list of the corresponding <Legacy system name ‖ Legacy entity name list> (line 18).

After execution of steps 1 through 15 for each row in the result set Q, the resulting legacy entity attributes with their corresponding values that need synchronization as a result of insert on the EDS entity 'Client' are shown in the following table:

Row num of set Q	Row num of set R	EDS entity attribute name	EDS entity attribute value	Legacy system name	Legacy entity name	Legacy entity attribute name	Attribute conversion procedure name	Legacy entity attribute value
2	1	clie_last_nam	'Rimmer'	Mortgage system	Customer	cust_last_nam		'Rimmer'
2	2	clie_last_nam	'Rimmer'	Investment system	Investor	invs_last_nam	truncate_last_nam	'Rimmer'
3	1	clie_first_nam	'Rodger'	Mortgage system	Customer	cust_first_nam		'Rodger'
3	2	clie_first_nam	'Rodger'	Investment system	Investor	invs_first_nam	truncate_first_nam	'Rodger'
4	1	clie_sin_num	508133212	Investment system	Investor	invs_sin_num		508133212
5	1	clie_gender_cde	1	Mortgage system	Customer	cust_gender_txt	convert_gender	'Male'
5	2	clie_gender_cde	1	Investment system	Investor	invs_gender_cde		1
6	1	clie_birth_dte	'25/05/60'	Investment system	Investor	invs_birth_dte		'25/05/60'
7	1	clie_marital_status_cde	1	Mortgage system	Customer	cust_marital_status_txt	convert_marital_status	'Single'

On execution of steps 16, 17 and 18, for each row in the result set Q, the Attribute_list in

the <Mortgage system ‖ Customer list> as well as the Attribute_list in the <Investment system ‖ Investor list> are as follows:

<Mortgage system ‖ Customer list>.Attribute_list ←

cust_last_nam : 'Rimmer',
cust_first_na : 'Rodger',
cust_gender_txt : 'Male',
cust_marital_status_txt : 'Single'

<Investment system □ Investor list>.Attribute_list ←

invs_last_nam : 'Rimmer',
invs_first_nam : 'Rodger',
invs_gender_cde : 1,
invs_birth_dte : '25/05/60'

On execution of steps 16, 17, and 19 for each row in the result set Q the Key_list in the <Investment system ‖ Investor list> is as follows:

<Investment system ‖ Investor list>.Key_list ← invs_sin_num : 508133212

Control then returns to line 4 of Perform_synchronization.

Line 4 of Perform_synchronization then calls the algorithm *Perform_EDS_PK_and_record_mapping*. This algorithm performs key to key and record to record mapping between the EDS and the legacy systems. The main purpose of this algorithm is to determine the contents of the Key_list in the <Mortgage system ‖ Customer list> as well as the Key_list in the <Investment system ‖ Investor list>.

Perform_EDS_PK_and_record_mapping (Algorithm 4, Figure 17)

This algorithm determines the primary key attributes(s) and values(s) for the mapped legacy entity(s).

Lines 1 and 2 of the algorithm select the PK attributes of the EDS entity 'Client' from the metadata mapper table 'EDS Entity PK Attributes' that are in P_Key_list. The selection

creates result set P that has the following row:

Result Set P

Row num	EDS entity PK num	EDS entity PK Attributes
1	1	clie_num

Line 3 checks the parameter P_Operation_type to determine if it is an insert operation. Since it is an insert operation, line 4 loops through the result set P. For each row in the result set P lines 5 and 6 determine the corresponding mapped legacy PK attributes (result set M). The result set M for the result set P is shown below:

Result Set M

Row num of set P	Row num of set M	Legacy system name	Legacy entity name	Legacy entity PK attribute name	Secondary entity PK generation algorithm
1	1	Mortgage system	Customer	cust_num	generate_cust_num
1	2	Investment system	Investor	invs_sin_num	

Line 7 loops through the result set M. For each row in the result set M, line 8 checks if there exists a 'Secondary entity PK generation algorithm' to generate a primary key value for the mapped legacy PK attribute. If such an algorithm exists, line 10 determines the corresponding legacy PK value by executing 'Secondary entity PK generation algorithm'. Line 11 then adds the legacy entity PK attribute and its value to the Key_list of the corresponding <Legacy system name ‖ Legacy entity name list>.

Looping through the result set M, since row 1 has a 'Secondary entity PK generation algorithm', the algorithm 'generate_cust_num' is executed by line 10 to determine the legacy PK value. Let us assume the value generated for 'cust_num' is 6789 then after execution of line 11, the Key_list in the <Mortgage system ‖ Customer list> is as follows:

<Mortgage system ‖ Customer list>.Key_list ← cust_num : 6789

Since row 2 of the set M does not have a 'Secondary entity PK generation algorithm', nothing gets generated or added to the Key_list in <Investment system ‖ Investor list>.

Note that the contents of the Key_list for <Investment system ‖ Investor list> were determined during execution of the algorithm Perform_EDS_attribute_mapping. The non PK attribute 'clie_sin_num' of 'Client' maps to PK attribute 'invs_sin_num' of 'Investor'. Therefore, the value of 'invs_sin_num' is determined during the attribute to attribute mapping between the EDS and the legacy systems. The synchronization algorithm handles both the cases. Case 1 where it performs record to record mapping for inserts by using key generation algorithms and case 2 where key generation algorithms do not exist and the value of the field is determined as a result of the attribute to attribute mapping.

After performing the mapping between the EDS and the legacy systems the corresponding data modifications as a result of the insert on the entity 'Client' in the EDS are:

(1) *<Mortgage system ‖ Customer list> ←*
Mortgage system, Customer, I,
Attribute_list, Key_list;

where Attribute_list is:

<Mortgage system ‖ Customer list>.Attribute_list ←
cust_last_nam : 'Rimmer',
cust_first_nam : 'Rodger',
cust_gender_txt : 'Male',
cust_marital_status_txt : 'Single'

where Key_list is:

<Mortgage system ‖ Customer list>.Key_list ← cust_num : 6789

And

(2) *<Investment system ‖ Investor list> ←*
Investment system, Investor, I, Attribute_list, Key_list;

where Attribute_list is -

<Investment system ‖ Investor list>.Attribute_list ←
invs_last_nam : 'Rimmer',
invs_first_nam : 'Rodger',
invs_gender_cde : 1,
invs_birth_dte : '25/05/60'

where Key_list is

<Investment system ‖ Investor list>.Key_list ← invs_sin_num : 508133212

The algorithm then returns to line 5 of Perform_synchronization.

Line 5 of Perform_synchronization checks P_Operation_type code to determine if the operation type is an insert. Since it is an insert operation, line 6 executes the algorithm *Maintain_EDS_cross_referencing.*

Maintain_EDS_cross_referencing (Algorithm 5, Figure 19)

This algorithm maintains record to record mapping between the EDS entity 'Client' and the Mortgage system entity 'Customer' as well as between the EDS entity 'Client' and the Investment system entity 'Investor'.

Lines 1 and 2 of the algorithm select the primary key attributes of the EDS entity 'Client' from the metadata mapper table 'EDS entity PK Attributes' that are in P_Key_list. The selection creates result set P that has the following row:

Result Set P

Row num	EDS entity PK num	EDS entity PK Attributes
1	1	clie_num

Line 3 loops through the result set P. For each row in the result set P, lines 4 and 5 determine the corresponding mapped legacy PK attributes (result set M). The result set M for the result set P is as follows:

Result Set M

Row num of set P	Row num of set M	Legacy system name	Legacy entity name	Legacy entity PK attribute name	Legacy entity PK num
1	1	Mortgage system	Customer	cust_num	2
1	2	Investment system	Investor	invs_sin_num	3

Line 6 loops through the result set M. Lines 9 and 10 determine the primary key values for EDS entity PK attributes (result set P) and the mapped legacy PK attributes (result set M). This is achieved by using the functions GET_VALUE_INLIST and GET_VALUE_OUTLIST to read from the input list and output lists. The primary key

values for the EDS entity PK attribute and the mapped legacy entity PK attributes after execution of lines 9 and 10 are:

EDS entity PK num	EDS value	Legacy entity PK num	Legacy value
1	4567	2	6789
1	4567	3	508133212

Line 11 checks if P_Operation_type is an insert operation. Since it is an insert operation, record to record mapping is stored in the cross referencing table.

Control returns to the end of the algorithm Perform_synchronization.

The algorithm Perform_synchronization terminates leaving the EDS synchronized with both the legacy systems.

5.3.2 Example 2

Consider an update on the entity 'Client' in the EDS.

UPDATE *client*
SET *clie_marital_status_cde = 2 and clie_last_num = 'Ronald'*
WHERE *clie_num = 4567;*

The change extractor associated with the EDS captures the update operation and passes the following parameters to the DTIM layer.

P_System_name ← 'EDS'
P_Entity_name ← Client
P_Operation_type ← 'U'
P_Key_list ← clie_num, 4567
P_Attribute_list ← clie_marital_status_cde : 2,
clie_last_nam : 'Ronald'

The DTIM layer executes the algorithm Perform_synchronization with the above parameters.

Perform_synchronization (Algorithm 1, Figure 12)

Since the condition in line 1 that checks whether P_System_name = 'EDS' is true, the algorithm *Perform_EDS_entity_mapping* is executed. This algorithm performs entity to entity mapping between the EDS and the legacy systems.

Perform_EDS_entity_mapping (Algorithm 2, Figure 13)

This algorithm determines the legacy systems and their entities that need to be synchronized as a result of the update on the EDS entity 'Client'.

On execution lines 1 and 2 of this algorithm determine these entities to be 'Customer' in the 'Mortgage system' and 'Investor' in the 'Investment system'.

Lines 6,7 and 8 of the algorithm create two lists, one for each legacy system name and legacy entity name with the following contents:

<Mortgage system ‖ Customer list> ← Mortgage system, Customer, U, Attribute_list, Key_list;

<Investment system ‖ Investor list> ← Investment system, Investor, U, Attribute_list, Key_list;

Please note that Attribute_list and Key_list (contained in the respective lists) are initially empty. It will be shown shortly how their contents are determined by the attribute and PK mapping algorithms.

Control then returns to line 3 of Perform_synchronization;

Line 3 of Perform_synchronization calls the algorithm *Perform_EDS_attribute_mapping*. This algorithm performs attribute to attribute mapping between the EDS and the legacy systems. The main purpose of this algorithm is to determine the contents of the Attribute_list in the <Mortgage system ‖ Customer list> as well as the Attribute_list in the <Investment system ‖ Investor list>.

Perform_EDS_attribute_mapping (Algorithm 3, Figure 15)

This algorithm determines the legacy systems' attributes that need to be synchronized as a result of the update on the EDS entity 'Client'.

Lines 1 and 2 of the algorithm select all attributes that belong to the EDS entity 'Client' and that are in P_Attribute_list from the metadata mapper table 'EDS Entity attributes'. The selection creates result set Q that has the following rows:

Result Set Q

Row num	EDS entity name	EDS entity attribute name	Code
1	Client	clie_marital_status_cde	SEPD
2	Client	clie_last_nam	SEPD

Line 3 loops through each row in the result set Q. For each row in the result set Q, lines 4,5 and 6 determine if the EDS entity attribute has the SEPD code (shared EDS primary data). If the EDS attribute has the code SEPD, lines 7 and 8 determine the corresponding mapped legacy attributes (result set R). The result set R for row num 1 of the result set Q is shown below.

Result Set R for row num 1 of the Result Set Q

Row num of set Q	Row num of set R	EDS entity attribute name	Legacy system name	Legacy entity name	Legacy entity attribute name	Attribute conversion procedure name
2	1	clie_marital_status_cde	Mortgage system	Customer	cust_marital_status_txt	convert_marital_status

Line 9 loops through each row in the set R. For each row in the set R lines 13,14 and 15 determine the corresponding value of the legacy entity attribute. Lines 16 and 17 then determine whether the legacy entity attribute is a PK attribute or a non PK attribute. If it is a PK attribute it is added to the Key_list of the corresponding <Legacy system name ‖ Legacy entity name list> (line 19) otherwise it is added to the Attribute_list of the corresponding <Legacy system name ‖ Legacy entity name list> (line 18).

After the execution of steps 1 through 15 for each row in the result set Q, the resulting legacy entity attributes, with their corresponding values that need synchronization as a

result of update on the EDS entity 'Client', are shown in the following table:

Row num of set Q	Row num of set R	EDS entity attribute name	EDS entity attribute value	Legacy system name	Legacy entity name	Legacy entity attribute name	Attribute conversion procedure name	Legacy entity attribute value
1	1	clie_marital_status_cde	2	Mortgage system	Customer	cust_marital_status_txt	convert_marital_status	'Married'
2	1	clie_last_nam	'Ronald'	Mortgage system	Customer	cust_last_nam		'Ronald'
2	2	clie_last_nam	'Ronald'	Investment system	Investor	invs_last_nam	tuncate_last_nam	'Ronald'

On execution of steps 16, 17 and 18 for each row in the result set Q the Attribute_list in the <Mortgage system ‖ Customer list> as well as the Attribute_list in the <Investment system ‖ Investor list> are as follows:

<Mortgage system ‖ Customer list>.Attribute_list ←
cust_marital_status_txt : 'Married',
cust_last_nam : 'Ronald'

<Investment system ‖ Investor list>.Attribute_list ←
invs_last_nam : 'Ronald'

Control then returns to line 4 of Perform_synchronization.

Line 4 of Perform_synchronization then calls the algorithm *Perform_EDS_PK_and_record_mapping*. This algorithm performs key to key and record to record mapping between the EDS and the legacy systems. The main purpose of this algorithm is to determine the contents of the Key_list in the <Mortgage system ‖ Customer list> as well as Key_list in the <Investment system ‖ Investor list>.

Perform_EDS_PK_and_record_mapping (Algorithm 4, Figure 17)

This algorithm determines the primary key attributes and values for the mapped legacy entities.

Lines 1 and 2 of the algorithm select the PK attribute(s) of the EDS entity 'Client' from the metadata mapper table 'EDS Entity PK Attributes' that are in P_Key_list. The

selection creates the result set P that has the following row:

Result Set P

Row num	EDS entity PK num	EDS entity PK Attributes
1	1	clie_num

Line 3 checks the parameter P_Operation_type to determine if it is an insert operation. Since it is not an insert operation, line 12 loops through the result set P. For each row in result set P lines 13 and 14 determine the corresponding mapped legacy PK attributes and their values (in result set M). The result set M for the result set P is shown below:

Result Set M

Row num of set P	Row num of set M	Legacy system name	Legacy entity name	Legacy entity PK attribute name	Legacy value
1	1	Mortgage system	Customer	cust_num	6789
1	2	Investment system	Investor	invs_sin_num	508133212

Line 15 loops through the result set M. For each row in the result set M, line 16 adds the legacy entity PK attribute and its value to the Key_list of the corresponding <Legacy system name ‖ Legacy entity name list>.

Processing row 1 of the result set M gives:

<Mortgage system ‖ Customer list>.Key_list ← cust_num : 6789

Processing row 2 gives:

<Investment system ‖ Investor list>.Key_list ← invs_sin_num : 508133212

After performing the mapping between the EDS and the legacy systems the corresponding data modifications, as a result of the update on the entity 'Client' in the EDS, are:

(1) *<Mortgage system ‖ Customer list> ← Mortgage system, Customer, U, Attribute_list, Key_list;*

where Attribute_list is:

<Mortgage system ‖ Customer list>.Attribute_list ←
cust_marital_status_txt : 'Married',
cust_last_nam : 'Ronald'

where Key_list is:

<Mortgage system ‖ Customer list>.Key_list ← cust_num : 6789

And

(2) *<Investment system ‖ Investor list>← Investment system, Investor, U,*
Attribute_list, Key_list;

where Attribute_list is -

<Investment system ‖ Investor list>.Attribute_list ←
invs_last_nam : 'Ronald'

where Key_list is

<Investment system ‖ Investor list>.Key_list ← invs_sin_num : 508133212

Control then returns to the end of Perform_synchronization.

The algorithm Perform_synchronization then terminates leaving the EDS synchronized with both the legacy systems.

The above examples illustrate how data modifications on the EDS are propagated to the legacy systems. Similarly, examples can be drawn to show propagation of data modifications from the legacy systems to the EDS.

5.4. Warehouse Anomaly

The warehouse anomaly was introduced in Chapter 4. It is associated with the materialized view maintenance approach to data warehousing. It arises when the queries from the data warehouse are interleaved with the updates arising from the base data sources. Additional mechanisms or algorithms are needed to avoid the warehouse anomaly. The materialized view approach is applicable to the data warehouse architecture rather than the EDS or the ODS architecture. This is due to the fundamental difference

between the data warehouse and the EDS architecture (or the data warehouse and the ODS architecture). These differences were mentioned in Chapter 1 and then revisited in Chapter 4. Chapter 4 concluded that the materialized view approach was not suitable to the EDS architecture due to: 1) the volatile nature of the EDS 2) the record to record mapping found in the EDS 3) the currency of data 4) lack of summarization and 5) the nature of the processing.

The EDS is built for on-line transaction processing of the primary data and off-line transaction processing of the secondary data. The very nature of the processing found in the EDS implies that the design of the EDS should be normalized, requiring record to record mapping and discouraging summarization. The data warehouse on the other hand is built for informational processing. It contains complicated, long running queries that access large amounts of data and therefore summarization or materializing queries is encouraged.

The proposed approach to synchronization in the EDS is based on the philosophy of storing all the information required for synchronization in the metadata mapper. (i.e., the metadata mapper contains all the knowledge required to convert a data modification from one tier to the corresponding data modification in the other tier. As a result, no additional queries are sent from the EDS to the base data sources or *vice versa*. Hence, the warehouse anomaly is not applicable to the EDS architecture. This is now illustrated with a delete anomaly example.

Consider two data stores, the EDS and the legacy data store 'Mortgage' (as shown in Appendix 2). The entity 'Client' in the EDS maps to the entity 'Customer' and 'Address' in the mortgage data store. Assume, the mortgage entities 'Customer' and 'Address' are the primary sources of client and address information and the entity 'Client' in the EDS is the secondary source. This implies any data modifications are first performed on the 'Customer' and 'Address' entities in the mortgage data store and then propagated to the EDS entity 'Client'. In other words, the 'Address' and 'Customer' information from the

mortgage system has been denormalized in the EDS entity 'Client'. If a customer has multiple addresses, such denormalization will never be appropriate for the EDS. This further strengthens the argument that summarization and denormalization are not found in the EDS architecture. For the sake of this discussion (and to keep it simple and realistic) assume that 'Customer' has just one address. In terms of the analogy with warehouse view definition, the EDS entity 'Client' can be considered as a view over two base data sources 'Customer' and 'Address'. Appendix 2 shows the mapping stored in the metadata mapper between the legacy entities 'Customer' and 'Address' and the EDS entity 'Client'.

5.4.1 The Delete anomaly

Consider a delete on the entity 'Customer' in the mortgage data store.

DELETE *address*
WHERE *addr_num = 6789;*

The change extractor associated with the mortgage data store captures the delete operation and passes the following parameters to the DTIM layer.

P_System_name ← *'Mortgage system'*
P_Entity_name ← *'Address'*
P_Operation_type ← *'D'*
P_Key_list ← *addr_num, 6789*
P_Attribute_list ← ϕ

The DTIM layer executes the algorithm Perform_synchronization with the above parameters.

Perform_synchronization (Algorithm 1, Figure 12)

Since the condition in line 1 that checks whether P_System_name = 'EDS' is false, the algorithm *Perform_legacy_entity_mapping* is executed. This algorithm performs entity to

entity mapping between the legacy system and the EDS.

Perform_legacy_entity_mapping (Algorithm 6, Figure 14)

This algorithm determines the EDS entities that need to be synchronized as a result of the delete on the mortgage entity 'Address'.

On execution lines 1 and 2 determine the EDS entity to be 'Client'.

Lines 6,7, and 8 of the algorithm result in creating a list with the following contents:

$$<EDS \parallel Client\ list> \leftarrow EDS,\ Client,\ D,\ Attribute\_list,\ Key\_list;$$

Please note that Attribute_list and Key_list (contained in the above list) are initially empty. It will be shown shortly how their contents are determined by the attribute and PK mapping algorithms.

Control then returns to line 8 of Perform_synchronization.

Line 8 calls the algorithm *Perform_legacy_attribute_mapping*. This algorithm performs attribute to attribute mapping between the legacy system and the EDS. The main purpose of this algorithm is to determine the contents of the Attribute_list in <EDS ‖ Client list>.

Perform_legacy_attribute_mapping (Algorithm 7, Figure 16)

This algorithm determines the EDS entity attributes that need to be synchronized as a result of the delete on the mortgage entity 'Address'.

Lines 1 and 2 of the algorithm select all attributes that belong to the mortgage entity 'Address' and that are in P_Attribute_list from the metadata mapper table 'Legacy Entity attributes'. The selection creates a result set Q that is empty since P_Attribute_list is ϕ.

Control then returns to line 9 of Perform_synchronization.

Line 9 calls the algorithm *Perform_legacy_PK_and_record_mapping*. This algorithm

performs key to key and record to record mappings between the legacy system and the EDS. The main purpose of this algorithm is to determine the contents of the Key_list in the <EDS ‖ Client list>.

Perform_legacy_PK_and_record_mapping (Algorithm 8, Figure 18)

This algorithm determines the primary key attributes(s) and values(s) for the mapped EDS entity(s).

Lines 1 and 2 of the algorithm select the PK attributes of the legacy entity 'Address' from the metadata mapper table 'Legacy Entity PK Attributes' that are in P_Key_list. The selection creates a result set P that has the following row:

Result Set P

Row num	Legacy entity PK num	Legacy entity PK Attributes
1	3	addr_num

Line 3 checks the parameter P_Operation_type to determine if it is an insert operation. Since it is not an insert operation, control passes to line 12 of the algorithm. Line 12 loops through the result set P. For each row in the result set P, lines 13 and 14 determine the corresponding mapped EDS PK attributes and their values (in result set M). The result set M for the result set P is:

Result Set M

Row num of set P	Row num of set M	EDS entity name	EDS entity PK attribute name	EDS value
1	1	Client	clie_num	4567

Line 15 loops through the result set M. Line 16 then adds the EDS entity PK attribute and its value to the Key_list of the corresponding <EDS ‖ EDS entity name list>. After execution of line 16, the Key_list in the <EDS ‖ Client list> is as follows:

<EDS ‖ Client list>.Key_list ← clie_num : 4567

After performing the mapping between the mortgage system and the EDS the

corresponding data modifications as a result of delete on the entity 'Address' in the Mortgage system are:

$<EDS \parallel Client\ list> \leftarrow EDS, Client, D, Attribute\_list, Key\_list;$

where Attribute_list is:

$<EDS \parallel Client\ list>.Attribute\_list \leftarrow \phi$

where Key_list is:

$<EDS \parallel Client\ list>.Key\_list \leftarrow clie\_num : 4567$

Control then returns to the end of Perform_synchronization.

The fact that key to key and record to record mappings are stored in the metadata mapper means the additional query required to determine the customer records associated with the deleted address record is not needed therefore, the anomaly does not occur. In fact the solution to the deletion anomaly discussed in [19] is based on storing primary keys in the warehouse for every base relation involved in the view.

The above example illustrates how the proposed synchronization solution is different from the materialized view approach. As a result the delete or update anomalies found in the data warehouse architecture is not applicable to the EDS architecture.

5.5 Correctness of the Synchronization Algorithms

In this section, the correctness of the synchronization algorithms is discussed. The synchronization algorithms use the mappings stored in the metadata mapper to convert a change in one tier to its corresponding change(s) in the other tier. Therefore, the correctness of these algorithms will depend on:

- Correctness of the metadata model, and
- The mapping data stored in the metadata mapper

Correctness of the metadata model

As discussed there are four kinds of mappings that are required for synchronization. They are - entity to entity mapping, attribute to attribute mapping, key to key mapping and record to record mapping. The metadata model introduced earlier in this chapter models the four kinds of mappings. Since the solution to synchronization is based on these mappings, it is very important that the metadata model models each kind of mapping correctly. In Section 5.2.1, the metadata model was described in detail. It was also shown how each kind of mapping was correctly modeled in the metadata model.

In Chapter 4, a formal classification of the types of data in the two tier data architecture is presented. The data in the EDS was classified into EEPD (exclusive EDS primary data), SEPD (shared EDS primary data), and ESD (EDS secondary data). Similarly, data in the application data store is classified into EAPD (Exclusive EDS primary data), SAPD (Shared application primary data) and ASD (Application shared data). This classification identifies that SEPD (shared EDS primary data) must be synchronized with the secondary data of application systems (ASDs) and the shared primary data of application systems (SAPDs) must be synchronized with the EDS secondary data (ESD). Further, no synchronization is needed for exclusive EDS primary data (EEPD) and exclusive application primary data (EAPD).

This classification clearly defines the types of data in the two tier data architecture and identifies those that need synchronization. The classification also puts forth the requirements for synchronization (i.e., in order to synchronize the EDS with the application systems SEPD must be synchronized with ASDs and SAPDs must be synchronized with ESD). The attribute 'EEPD or SEPD or ESD code' of the entity 'EDS Entity Attributes' in the metadata model classifies the attributes of EDS entities to EEPD, SEPD or ESD code. Similarly, the attribute 'EAPD or SAPD or ASD code' of the entity 'Legacy Entity Attributes' classifies the attributes of legacy entities to EAPD, SAPD or ASD code. With this classification the entity 'EDS Legacy Attribute Mapping' is able to map the SEPD attributes to ASD attributes and SAPD attributes to ESD attributes.

The feasibility of this attribute to attribute mapping by the metadata model shows how the mappings between the EDS and the operational systems can be used to synchronize SEPD with ASDs and SAPDs with ESD. This shows the mappings modeled in the metadata model are correct which further infers the correctness of the metadata model.

The Correctness of the Mapping Data Stored in the Metadata Mapper

The mapping data stored in the metadata mapper is based on the requirements of the organization. For example, Appendix 1 shows the implementation of the metadata mapper for the three entities. The data and algorithms stored in the metadata mapper are specific to the needs of the organization. Population of the mapping data in the metadata mapper is a legitimate and interesting research question, it is not the one addressed in this research. As mentioned before, tools can be developed that will automate or semi-automate population of mapping data in the metadata mapper. This thesis assumes that the mapping between the EDS and the operational systems can be correctly determined and stored in the metadata mapper either with the help of tools or by using alternate methods.

Based on the above assumption and the correctness of metadata model, the synchronization algorithms are correct.

Chapter 6.

Conclusions

6.1 Summary and Contributions

Integrating data from multiple, heterogeneous databases and other information sources has been one of the leading research issues in database research and industry. In this thesis, the research done on data integration was broadly classified into Multidatabase Systems and Data Warehousing. The thesis provides a comprehensive comparison between the two approaches and argues that in spite of certain drawbacks with the data warehousing approach, it is a much simpler and more powerful solution to the data integration problem.

This thesis contributes towards solving the problem of data integration using the data warehousing approach and makes a number of important contributions. First, it defines a new data integration architecture by defining an architectural construct the Enterprise Data Store (EDS). An Enterprise Data Store is a repository of data that represents an integrated view of enterprise operations and is built for corporate wide operational informational processing and transactional processing of common business operations. The research discusses in depth, the characteristics of the EDS and compares them to the ODS. The research also presents the corporate data architecture with the EDS, the data warehouse and the application systems. The research argues that the ODS fails to provide true operational integration because 1) it does not eliminate the operational redundancy of common business operations, and 2) it does not provide a consistent view of data across the application systems and the ODS. The EDS overcomes these limitations.

The second major contribution of this thesis is that it introduces a new approach to synchronization based on using metadata for synchronizing the EDS with the application systems. Metadata is one of the most important aspects of the data warehousing environment. A very important component of the data warehouse metadata store is the mapping between the operational systems and the data warehouse. The research, based on this component of the data warehouse metadata store, identifies four kinds of mappings - entity to entity, attribute to attribute mapping, key to key, and record to record mappings that can be used to synchronize the EDS with the application systems. The mappings are modeled in a metadata model that is implemented as the metadata mapper. The mapping data and algorithms stored in the metadata mapper are used by the synchronization algorithms to synchronize the EDS with the operational systems. Early synchronization architectures (e.g., the WHIPS architecture) in data warehousing are based on materialized view approach. The proposed approach offers two main advantages over the materialized view approach. First, it simplifies the synchronization architecture by taking away complexities like global query decomposition, global query optimization, global concurrency control, and distributed query processing. Second, it facilitates simultaneous development of the metadata store. Though metadata is an essential component of the data warehouse architecture, the development of this component is usually ignored. The proposed architecture enables the development of a major component of the metadata store and keeps it current with the data in the EDS and the operational systems. The thesis also contributes by providing a metadata model to store the mappings between the operational systems and the EDS.

The thesis also makes a significant contribution by proposing an architecture for synchronizing the EDS with the application systems. The architecture gives a classification of different kinds of data found in the two tier data architecture. This classification clearly identifies the data that needs to be synchronized between the two tiers. Also, it clearly differentiates the subsets of data in a data source with which the propagator and the change extractor should be associated. The synchronization architecture is based on two centralized components - the DTIM and the metadata

mapper. These components contain all the knowledge needed to accept a change/update from a tier and convert it to the corresponding change(s)/updates(s) in the other tier. No queries are posed on the application systems as no additional information is required for the synchronization. This simplifies the architecture tremendously and the components like wrappers, mediators, and query processors that are needed by other architectures to deal with issues like global query decomposition and optimization, distributed query processing, mediation, multi-source warehouse consistency are not needed.

One of the advantages offered by the proposed synchronization architecture is that no collision or conflict detection and resolution mechanisms are needed. Since in the proposed architecture a data element is only maintainable by a single data source (primary data of a data source) and is read only in the other data sources (secondary data of data sources), the possibility of collisions has been eliminated.

Another advantage of the proposed architecture is that mechanisms required for ensuring serializability of local and external (propagated) transactions are not required. This is because local transaction will act on primary data of a data source and propagated transactions will act on the secondary. Since they are two separate subsets of data, serializability is not an issue.

Finally, the thesis also contributes by providing a comprehensive set of synchronization algorithms. These algorithms use the mappings stored in the metadata mapper to convert a change in one tier to its corresponding changes in the other tier. These algorithms illustrate the viability of the proposed synchronization solution that uses metadata for synchronization and introduce a prototype of the metadata mapper and the DTIM layer based on simple mapping between the EDS and the operational systems. This prototype can be further customized and expanded depending on the requirements of the organization.

6.2 Future Research

Although this research has made a number of significant contributions in the area of data integration, some open problems still exist. The next few paragraphs present some of the open research problems related to this research.

The proposed solution to synchronize the EDS with the operational systems is based on using the mapping data and algorithms stored in the metadata mapper. This raises a legitimate and interesting question about how the relevant mappings between the EDS and the operational systems are determined and stored in the metadata mapper. Without any tools the manual process of determining the four kinds of mappings (entity, attribute, primary, and record) between the EDS and the operational systems will be a tedious, error prone and time consuming process. Further, once the mappings are determined, relevant data and conversion algorithms need to be stored in the metadata mapper. To facilitate easy determination and storage of mapping data, tools could be developed to automate or semi-automate implementation of the metadata mapper. These tools might read schema information from dictionaries of the data sources involved in the integration architecture. This information can then be presented to the user in such a manner that the user can easily map entities and their respective attributes from one data store to the other. As a result of mapping, the tool could then produce the transformation logic needed for synchronization. Building of such metadata mapping tools can be an interesting future research proposal.

Change detection is an open research problem that arises from the warehousing approach. The solution to the change detection problem is dependent on the underlying application sources. Earlier work towards change detection has classified the application sources into - cooperative sources, logged sources, queryable sources, and snapshot sources. Each type of application source capability provides interesting research problems for change detection. In the EDS architecture, a change extractor is associated with each data source participating in the integration architecture. This is because the functionality of the

change extractor is dependent on the type of source (e.g., legacy system, relational) as well as data provided by the source. Efficient algorithms need to be developed that are optimized for detecting and capturing only the relevant information needed for the synchronization. For example, optimizing the change extractor to detect and capture data modification only on the shared primary data of the associated data source.

Another important component of the EDS architecture is the propagator. The propagator converts the logical transaction passed by the DTIM layer into the physical transaction in the language of the associated data source. Like the change extractor, we need a different propagator for each application source and the EDS. This is because the functionality of the propagator is dependent on the type of the source (e.g., database system, legacy system, etc.) as well as the type of data manager and the query language associated with the source. Algorithms and techniques need to be built for the efficient implementation of the propagators.

A different propagator/change extractor is needed for each data source. Clearly it is undesirable to hard-code a propagator/change extractor for each data source participating in the integration architecture. Hence, a significant research issue is to develop techniques and tools that automate or semi-automate the process of implementing change extractors / propagators through a tool kit or specification based approach.

The synchronization algorithms proposed in this research are based on simple mappings between the EDS and the operational systems. The purpose of this research was to illustrate the viability of the proposed synchronization solution that uses metadata for synchronization; and to build a framework for the synchronization of the EDS with the operational systems. For future research, the metadata model and the synchronization algorithms can be modified and/or expanded to support complicated cases of mappings between the EDS and the operational systems.

Another interesting area for future research would, of course, be the implementation of

the proposed data integration architecture. This will require building the EDS construct, modifying the application sources, and implementing the proposed synchronization architecture with change extractors, propagators, DTIM layer, and metadata mapper.

Bibliography

[1] W.H. Inmon, C. Imhoff, R. Sousa. Creating an Information Ecosystem. In Robert Elliott, editor, *Corporate Information Factory*, pp. 1-11, John Wiley & Sons, Inc., New York, U.S.A, 1998.

[2] Systems Techniques, Inc. Information Architecture : *Managing Customer Relationships*. http://warehouse.chimenet.org/software/datastore/datarepos/whitecia.html, 1995.

[3] W.H. Inmon, C. Imhoff, G. Battas. In Robert Elliott, editor, *Building the Operational Data Store*, John Wiley & Sons, Inc., New York, U.S.A, 1996.

[4] *IEEE Computer*. Special Issue on Heterogeneous Distributed Database Systems, 24(12), 1991.

[5] M.T. Ozsu, P.Valduriez. Distributed Multidatabase Systems. In Christina Burghard and Jennifer Wenzel, editors, *Principles of Distributed Database Systems*, pp. 425-456. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[6] W.H. Inmon. *The Operational Data Store*. Tech Topic, 1(17), Prism Solutions Inc., 1000 Hamilton Court, Sunnyvale, CA 94089, 1993.

[7] W.H. Inmon. *What is a Data Warehouse*. Tech Topic, 1(1), Prism Solutions Inc., 1000 Hamilton Court, Sunnyvale, CA 94089, 1995.

[8] W.H. Inmon. *Defining the System of Record for the Data Warehouse*. Tech Topic, 1(3), Prism Solutions Inc., 1000 Hamilton Court, Sunnyvale, CA 94089, 1993.

[9] J. Widom. Research Problems in Data Warehousing. *In Proceedings of the 4th*

International Conference on Information and Knowledge Management - CIKM'95, pp. 25-30, November, 1995.

[10] Inmon. *Meta Data in the Data Warehouse*. Prism Solutions Inc., 1000 Hamilton Court, Sunnyvale, CA 94089, Tech Topic, 1(6), 1996

[11] J. Hammer, H. Molina, J. Widom, W. Labio, Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin*, 18(2), pp. 41-48, 1995

[12] J. Wiener, H. Gupta, W. Labio, Y Zhuge, H Molina, J. Widom. A System Prototype for Warehouse View Maintenance. *In Proceedings of the Workshop on Materialized Views*, pp. 26-33, June 1996.

[13] D. Lomet and J. Widom, editors. Special Issue on Materialized Views and Data Warehousing, *IEEE Data Engineering Bulletin* 18(2), June 1995.

[14] S. Abiteboul and A. Bonner. Objects and views. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 238-247, Denver, Colarado, May 1991.

[15] E. Bertino. A View Mechanism for Object-Oriented Databases. *In Advances in Database Technology-EDBT'92*, Lecture Notes in Computer Science 580, pp. 136-151, Springer-Verlag, Berlin, March 1992.

[16] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. *In Proceedings of the Seventh International Conference on Very Large Data Bases - VLDB'91*, pp. 577-589, Barcelona, Spain, September 1991.

[17] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. *In Proceedings of the ACM SIGMOD International*

Conference on Management of Data, pp. 316-327, San Jose, California, May 1995.

[18] N. Roussopoulos. Materialized Views and Data warehouses. *In the Proceedings of the 4th KRDB Workshop* Athens, Greece, August 1997.

[19] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom. The TSIMMIS project: Integration of Heterogeneous Information Sources. *In Proceedings of the 100th Anniversary Meeting of the Information Processing* Society of Japan, pp. 7-18, Tokyo, Japan, October 1994.

[20] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3), pp. 38-49,1992.

[21] V. Vassalos, Y. Papakonstantinou. Describing and Using Query Capabilities of Heterogeneous Sources. *In Proceedings of the twenty third Conference on Very Large Databases*, pp. 256-265, Athens, Greece, 1997.

[22] C. Li, R. Yereni, V.Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. Ullman, M. Valiveti. Capability Based Mediation in TSIMMIS. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 564-566, 1998.

[23] J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yerneni, M. Breunig, V.Vassalos. Template-Based Wrappers in the TSIMMIS System. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 532-535, 1997.

[24] Y. Zhuge, H. Garcia-Molina, J. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. *In Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, pp. 146-157, 1996.

[25] W. Labio, H. Garcia-Molina. Efficient Snapshot Differential Algorithms in Data warehousing. Technical report, Dept. of Computer Science, Stanford University,

1995. ftp://db.stanford.edu/pub/labio/1995/window.ps.

[26] K. Barker, M.Evans, J. Anderson. Measuring Autonomy in Heterogeneous Cooperative Systems. Technical Report, TR 92-08, University of Manitoba, Dept. of Computer Science, 1992.

[27] K. Barker. Taxonomy of Heterogeneity in Multidatabase Systems. Technical Report, TR 92-10, University. of Manitoba, Dept. of Computer Science, 1992.

[28] J. Widom and S.Ceri. Active Database Systems: Triggers and Rules for Advanced Database Processing. Morgan Kaufmann, San Francisco, California, 1995.

[29] L. Baekgaard, N. Roussopoulos. Efficient Refreshment of Data Warehouse Views. Technical Report, Institute for Advanced Computer Study and Dept. of Computer Science, University of Maryland at College Park.

[30] Y. Zhuge, J. Wiener, H. Garcia-Molina. Multiple View Consistency for Data Warehousing. *In proceedings of the Thirteenth International Conference on Data Engineering*, pp. 289-300, 1997

[31] N. Huyn. Efficient View Self-Maintenance. *In Proceedings of Workshop on Materialized views*: Techniques and Applications, pp. 17-25, 1996.

[32] D. Quass, A. Gupta, I. Mumick, J. Widom. Making views Self-Maintainable for Data Warehousing. *In Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, pp. 158-169, 1996.

[33] D. Quass. Maintenance Expressions for Views with Aggregation. Workshop on Materialized Views: Techniques and Applications, pp. 110-118, 1996.

[34] W.H. Inmon. *Building the Data Warehouse*, QED Information Sciences, Wellesley,

U.S.A., 1992.

[35] W.H. Inmon. *Enterprise architecture for the 90's*. Tech Topic, Prism Solutions Inc., 1000 Hamilton Court, Sunnyvale, CA 94089, 1(13), 1993.

[36] S. Ram. Heterogeneous Distributed Database Systems. *IEEE Computer*. Special Issue on Heterogeneous Distributed Database Systems, 24(12), December 1991.

[37] W. Kim, J. Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer*. Special Issue on Heterogeneous Distributed Database Systems, 24(12), December 1991.

[38] R. Ahmed. P. Smedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, M. Shan. The Pegasus Heterogeneous Multidatabase System. *IEEE Computer*. Special Issue on Heterogeneous Distributed Database Systems, 24(12), December 1991.

[39] C. Batini, M. Lenzirini, and S.B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computer Surveys*, 18(4) pp. 323-364, December 1986.

[40] C. Chung, Dataplex: An Access to Heterogeneous Distributed Databases. *Communications of the ACM*, 33(1), pp. 70-80, 1990.

[41] Y. Brietbart, P. L. Olson and G.L. Thompson. Database Integration in a Distributed Heterogeneous Database System. *In Proceedings of International Conference on Data Engineering*, IEEE CS Press, pp. 301-310, Los Alamitos, California, 1986.

[42] T.A Landers and R.L. Rosenberg, An Overview of Multibase – A Heterogeneous Database System. *Distributed Databases*, H-J. Schneider, ed., pp. 153-184, North-Holland, Amsterdam, 1982.

Appendix 1

Metadata Mapper for Example 1

EDS Entities

EDS entity name	Entity Description
Client	The entity stores information about all the clients of the company.

Legacy System

Legacy system name	Legacy description
Mortgage system	System responsible for the processing of mortgages. .
Investment system	System responsible for the processing of investments.

Legacy System Entities

Legacy system name	Legacy entity name	Entity Description
Mortgage system	Customer	Mortgage client.
Investment system	Investor	Investment client.

EDS Legacy Entity Mapping

EDS entity name	Legacy system name	Legacy entity name
Client	Mortgage system	Customer.
Client	Investment system	Investor.

EDS Entity Attributes

EDS entity name	EDS entity attribute name	Datatype	Position	Nullable	Default	Code	Definition
Client	clie_num	number(9)	1	No		EEPD	Client number.
Client	clie_last_nam	char(30)	2	No		SEPD	Last name.
Client	clie_first_nam	char(30)	3	No		SEPD	First name.
Client	clie_sin_num	number(12)	4	No		SEPD	Sin number.
Client	clie_gender_cde	number(1)	5	No		SEPD	Gender code.
Client	clie_birth_dte	date	6	Yes		SEPD	Birth date.
Client	clie_marital_status_cde	number(1)	7	No	1	SEPD	Marital status.

Legacy Entity Attributes

Legacy system name	Legacy entity name	Legacy entity attribute name	Datatype	Position	Nullable	Default	Code	Definition
Mortgage system	Customer	cust_num	number(9)	1	No		EAPD	Customer number.
Mortgage system	Customer	cust_last_nam	text(30)	2	No		ASD	Last name.
Mortgage system	Customer	cust_first_nam	text(30)	3	No		ASD	First name.
Mortgage system	Customer	cust_gender_txt	text(1)	4	No		ASD	Gender code.
Mortgage system	Customer	cust_marital_status_txt	text(1)	5	No		ASD	Marital status.
Investment system	Investor	invs_sin_num	number(12)	1	No		ASD	Sin number.
Investment system	Investor	invs_last_nam	text(25)	2	No		ASD	Last name.

Investment system	Investor	invs_first_nam	text(25)	3	No		ASD	First name.
Investment system	Investor	invs_gender_cde	number(1)	4	No		ASD	Gender code.
Investment system	Investor	invs_birth_dte	date	5	No		ASD	Birth date.

EDS Legacy Attribute Mapping

EDS entity name	EDS entity attribute name	Legacy system name	Legacy entity name	Legacy entity attribute name	Attribute conversion procedure name
Client	clie_last_nam	Mortgage system	Customer	cust_last_nam	
Client	clie_last_nam	Investment system	Investor	invs_last_nam	truncate_last_nam
Client	clie_first_nam	Mortgage system	Customer	cust_first_nam	
Client	clie_first_nam	Investment system	Investor	invs_first_nam	truncate_first_nam
Client	clie_sin_num	Investment system	Investor	invs_sin_num	
Client	clie_gender_cde	Mortgage system	Customer	cust_gender_txt	convert_gender
Client	clie_gender_cde	Investment system	Investor	invs_gender_cde	
Client	clie_birth_dte	Investment system	Investor	invs_birth_dte	
Client	clie_marital_status_cde	Mortgage system	Customer	cust_marital_status_txt	convert_marital_status

EDS Entity PK Attributes

EDS entity PK num	EDS entity name	EDS entity PK attribute name	PK Attribute position
1	Client	clie_num	1

Legacy Entity PK Attributes

Legacy entity PK num	Legacy system name	Legacy entity name	Legacy entity PK attribute name	PK Attribute position
2	Mortgage system	Customer	cust_num	1
3	Investment system	Investor	invs_sin_num	1

EDS Legacy PK Mapping

Legacy entity PK num	EDS entity PK num	Secondary entity PK generation algorithm
2	1	generate_cust_num
3	1	

Cross Referencing

Cross referencing num	Legacy entity PK num	EDS entity PK num	Legacy value	EDS value
1	3	1	508133212	4567
2	2	1	6789	4567

Conversion Procedures

Conversion procedure name	Procedure text
truncate_last_nam	input : last_name // upto 30 character long //
	output : trunc_last_name // truncate the input (last name) to 25 characters //
	var : trunc_last_name;

	begin
	trunc_last_name ← substr(last_name,1,25);
	return(trunc_last_name);
	end;
truncate_first_nam	input : first_name // upto 30 character long //
	output : trunc_first_name // truncate the input (last name) to 25 characters //
	var : trunc_first_name;
	begin
	trunc_first_name ← substr(first_name,1,25);
	return(trunc_first_name);
	end;
convert_gender	input : gender_code // 1 for male, 2 for female //
	output : gender_text // M for male, F for female //
	var : gender_text
	begin
	if gender_code = 1 then
	gender_text ← M;
	end if;
	if gender_code = 2 then
	gender_text ← F;
	end if;
	return(gender_text);
	end;
convert_marital_status	input : marital_code // 1 for single, 2 for married , 3 for divorced/
	output : marital_text // single for single, married for married, divorced for divorced //
	var : marital_text

	begin
	if marital_code = 1 then
	marital_text ← 'Single';
	end if;
	if marital_code = 2 then
	marital_text ← 'Married';
	end if;
	if marital_code = 3 then
	marital_text ← 'Divorced';
	end if;
	return(marital_text);
	end;
generate_cust_num	input;
	output : generated_key;
	begin
	generated_key := get next cust_num;
	return(generated_key);
	end;

Appendix 2

Metadata Mapper for Example 2

EDS Entities

EDS entity name	Entity Description
Client	The entity stores information about all the clients of the company.

Legacy System

Legacy system name	Legacy description
Mortgage system	System responsible for the processing of mortgages. .

Legacy System Entities

Legacy system name	Legacy entity name	Entity Description
Mortgage system	Customer	Mortgage customer.
Mortgage system	Address	Customer's address.

EDS Legacy Entity Mapping

EDS entity name	Legacy system name	Legacy entity name
Client	Mortgage system	Customer.
Client	Mortgage system	Address.

EDS Entity Attributes

EDS entity name	EDS entity attribute name	Datatype	Position	Nullable	Default	Code	Definition
Client	clie_num	number(9)	1	No		ESD	Client number.
Client	clie_last_nam	char(30)	2	No		ESD	Last name.
Client	clie_first_nam	char(30)	3	No		ESD	First name.
Client	clie_addr_line1_txt	char(30)	4	No		ESD	Address line 1 text.
Client	clie_addr_city_nam	char(30)	5	No		ESD	City.
Client	clie_addr_country_cde	number(1)	6	No		ESD	Country.

Legacy Entity Attributes

Legacy system name	Legacy entity name	Legacy entity attribute name	Datatype	Position	Nullable	Default	Code	Definition
Mortgage system	Customer	cust_num	number(9)	1	No		SAPD	Customer number.
Mortgage system	Customer	cust_last_nam	text(30)	2	No		SAPD	Last name.
Mortgage system	Customer	cust_first_nam	text(30)	3	No		SAPD	First name.
Mortgage system	Address	addr_num	number(9)	1	No		EAPD	Address number.
Mortgage system	Address	addr_line1_txt	text(30)	2	No		SAPD	Address line 1 text.
Mortgage system	Address	addr_city_nam	text(30))	3	No		SAPD	City.
Mortgage system	Address	addr_country_cde	number(1)	4	No		SAPD	Country.
Mortgage system	Address	addr_cust_num	number(9)	5	No		SAPD	Customer number.

EDS Legacy Attribute Mapping

EDS entity name	EDS entity attribute name	Legacy system name	Legacy entity name	Legacy entity attribute name	Attribute conversion procedure name
Client	clie_num	Mortgage system	Customer	cust_num	
Client	clie_num	Mortgage system	Address	addr_cust_num	
Client	clie_last_nam	Mortgage system	Customer	cust_last_nam	
Client	clie_first_nam	Mortgage system	Customer	cust_first_nam	
Client	clie_addr_line1_txt	Mortgage system	Address	addr_line1_txt	
Client	clie_addr_city_nam	Mortgage system	Address	addr_city_nam	
Client	clie_addr_country_cde	Mortgage system	Address	addr_country_cde	

EDS Entity PK Attributes

EDS entity PK num	EDS entity name	EDS entity PK attribute name	PK Attribute position
1	Client	clie_num	1

Legacy Entity PK Attributes

Legacy entity PK num	Legacy system name	Legacy entity name	Legacy entity PK attribute name	PK Attribute position
2	Mortgage system	Customer	cust_num	1
3	Mortgage system	Address	addr_num	1

EDS Legacy PK Mapping

Legacy entity PK num	EDS entity PK num	Secondary entity PK generation algorithm
2	1	
3	1	

Cross Referencing

Cross referencing num	Legacy entity PK num	EDS entity PK num	Legacy value	EDS value
100	2	1	4567	4567
200	3	1	6789	4567

Conversion Procedures

Conversion procedure name	Procedure line number	Procedure text