Data warehousing technologies for large-scale and right-time data

Xiufeng, Liu

Data Warehousing Technologies for Large-scale and Right-time Data

Xiufeng Liu

Ph.D. Dissertation

A dissertation submitted to the Faculty of Engineering and Science at Aalborg University, Denmark, in partial fulllment of the requirements for the Ph.D. degree in computer science.

Abstract

This thesis is about data warehousing technologies for large-scale and right-time data. Today, due to the exponential growth of data, it has become a common practice for many enterprises to process hundreds of gigabytes of data per day. Traditionally, data warehousing populates data from heterogeneous sources into a central data warehouse (DW) by Extract-Transform-Load (ETL) at regular time intervals, e.g., monthly, weekly, or daily. But now, it becomes challenging for large-scale data, and hard to meet the near real-time/right-time business decisions. This thesis considers some of these challenges and makes the following contributions:

First, this thesis presents a new and efficient way to store triples from an OWL Lite ontology known from the Semantic Web field. In contrast to classic triple-stores where the data with the triple format of $(subject, predicate, object)$ is stored in few, but big, tables with few columns, the presented triple-store spreads the data over more tables that may have many columns. The triple-store is optimized by an extensive use of bulk techniques, which makes it very efficient to insert and extract data. The DBMS-based solution makes it very flexible to integrate with other non-triple data.

Second, this thesis presents a middle-ware system for live DW data. Processing live DW data is one of the most tricky problems in data warehousing. An innovative method is proposed for processing live DW data, which accumulates the data in an intermediate data store, and does data modifications on-the-fly when the data is materialized or queried. The data is made available in the DW exactly *when* needed and users can get bulk-load speeds, but INSERT-like data availability.

Third, this thesis presents the first dimensional ETL programming framework using MapReduce. Parallel ETL is needed for large-scale data, but it is not easy to implement. This presented framework makes this very easy by offering high-level ETL-specific constructs, including those for star schema, snowflake schema, slowly changing dimensions (SCDs) and very large dimensions. The framework can achieve high programming efficiency, i.e., only a few statements needed for implementing a parallel ETL program, and good scalability for processing different DW schemas.

Finally, this thesis presents scalable dimensional ETL for cloud warehouse. Today, organizations gain growing interest in moving data warehousing systems towards the cloud, however, the current data warehousing systems are not yet particularly suited for the cloud. The presented framework exploits Hadoop to parallelize ETL execution and Hive as the warehouse system. It has a shared-nothing architecture, and supports scalable ETL operations on clustered commodity machines. To implement dimensional ETL, this framework can achieve higher programmer productivity than Hive, and its performance is also better.

In summary, this thesis discusses several aspects of the current challenges and problems of data warehousing, including integrating Web data, near real-time/right-time data warehousing, handling the exponential growth of data and cloud data warehousing. This thesis proposes a variety of technologies to deal with these specific issues.

Acknowledgments

I would like to thank a number of people who helped and supported me during my Ph.D. studies. First of all, I would like to thank my supervisor, Professor Torben Bach Pedersen, who gave me this research opportunity. As I had already been in industry for nearly ten years, I did not expect that I still had the opportunity to pursue doctoral degree. During my Ph.D., he taught me how to do research, how to formulate a research idea, and how to write the research papers. He is a dedicated and excellent researcher. He gave me a lot of guidance, help, and advice on my work and writing. I would also like to thank the Assistant Professor, Christian Thomsen. Part of my work in this thesis is the continuous work of his Ph.D. research. During my Ph.D., he gave me a lot of beneficial advice and sparking idea on my research, and gave a lot of help on my paper writing.

I would like to take this opportunity to thank all my colleagues in the Database and Programming Technologies Group at Aalborg University. I had a great time being with them. We have the table soccer competition, beer tasting party and many other small gatherings. Meanwhile, I audited many wonderful presentations of my colleagues in DBLunch (later upgraded to Daisy Seminar), and I was greatly inspired by their talks. Their talks not only broadened my view in the cutting-edge technologies of database, but also helped me understand how to do research in better.

I would also like to thank my parents, my older brothers and sisters, and my friends in China. I would say sorry to my aged parents that I am living so far and not able to visit them often. Thanks for their understanding and encouragement, and thanks my older bothers for taking care of my parents in hard time. Without their rear supports, it is not possible for me to dedicate to the research.

Finally, special thanks go to my wife, Yan, and my son, Yangyang. I am in debt to them for my leaving them in another country at the first two years of my Ph.D., and especially little time with my son. Thanks for their moral support and encouragement. Their love and understanding has helped me to work better and more focused.

Contents

Chapter 1

Introduction

In this rapidly changing age, decision makers need to utilize a wide range of information systems to improve their business decisions. One of the most used information systems is business intelligence (BI), which takes decision makers to a high level by providing them with a thorough understanding of an organization's operations. The term of BI was first proposed by Howard Dresner (later a Gartner Group analyst) in 1989 to describe "*concepts and methods to improve business decision making by using fact-based support systems*" [52]. Another term closely related to BI is data warehouse (DW), which is the information source and the base of business intelligence. According to Kimball and Inmon, DW is defined as a "*subject-oriented, non-volatile, time-variant data repository in support of management decisions*" [43]. Data warehousing is a collection of technologies for integrating data from heterogeneous sources into a central DW. Typically the data integration proceeds in three steps: the data of interest is first extracted from the sources, subsequently transformed and cleansed, and finally loaded into the DW. This is what we refer to as Extract-Transform-Load (ETL). Today, BI is now widely used to describe analytical applications [93]. The BI market has become one of the fastest growing software markets. According to Gartner [32], the worldwide BI market revenue is predicted to grow 8.7 percent to approximately $12.7 billion in 2012. This growth reflects the increasing importance and interest in the BI technologies.

1.1 Background and Motivation

The work that led to this thesis is in relation to the eGovMon project [27] which develops open source software, methodologies for quality evaluation of eGovernment services. The eGovMon project is to help governments deliver services to citizens in better, improve interactions with business and industry, and better government management. The eGovernment services are evaluated using four indicators, including accessibility, transparency, efficiency and impact (ATEI). The project requires a DW system with good scalable capability, and with high performance for the analysis of the four indicators. For example, to evaluate the accessibility which people with disabilities can perceive, understand, navigate, and interact with the web, a large number of web pages are crawled and evaluated automatically by the developed tools. The evaluation produces large amounts of RDF/OWL data. The data warehousing technologies proposed in this thesis are able to make the evaluation results quickly available to analysis. Since the requirements of the eGovMon DW also reflect the recent major concerns of DW community, including open source, support for web data, right-time and near real-time, and handling large-scale data sets, the proposed DW technologies in this thesis are designed to be general, and thus can also be applied to the other projects with these concerns. More specifically, this thesis is motivated by several aspects of the current challenges and issues, which are discussed in the following.

1.1.1 Motivation for Open Source

Today, the BI tools are available under both commercial and open source licenses. In the commercial world, business intelligence tools and technologies have come to their maturity in the last decades. Many BI vendors, such as IBM, SAP, Business Objects, Cognos, etc., provide both individual BI tools and integrated solutions. In the open source world, however, the use of business intelligence tools is still very limited, e.g., most tools are only used standalone, or have to be customized or tailored in compliance to the distinctive needs and requirements of businesses. Nevertheless, using open source BI shows very promising benefits, e.g., low-cost, manageable size, flexibility and reducing the dependence on software vendors. Remarkably, an increasing number of researchers and companies have developed open source BI tools and proposed different solutions in recent years. Thomsen and Pedersen conducted surveys of open source BI tools and the development status in 2005 and 2008 [80, 81], respectively. The surveys investigated the tools that can make a complete BI solution, including Extract–Transform–Load (ETL) tools, DBMS, On-Line Analytical Processing (OLAP) Servers and OLAP Clients. The surveys found that many new BI tools were developed between the two surveys, and there exists mature and powerful tools in all categories (but the functionalities are still short of that found in commercial tools). The surveys also found that the open source DBMSs have the highest maturity, e.g., some can be compared with commercial DBMSs, but the ETL tools are the least mature compared with the tools in other categories. Thus, it is worthwhile to pay our attention and effort to the open source tools for ETL.

1.1.2 Support for Web Data

Today, the Web has become the major platform for publishing information. Nearly all the corporations, organizations, and authorities have their own web sites. People use the Web to study, search and share information, and now the Web is an inseparable part of people's life. The Web has become one of the most important information resources. Resource Description Framework (RDF) and its extension Web Ontology Language (OWL) are used for modeling the Web data by making statements about resources [67]. In recent years, they becomes popular due to the increasing need for an effective underlying data model for the growing Semantic Web. Unfortunately, it remains a challenge to store and query RDF/OWL data in an efficient manner. For the existing DBMS-based stores, they have difficulty in offering high scalability and low latency querying. For the existing file-based stores, they have better performance than DBMS-based stores in general [20], however, they lack the flexibility offered by the DBMS. Thus, it is interesting develop the storage engine that supports saving RDF/OWL data in DBMS, but extensively optimizes the engine to achieve the performance that is comparable to the file-based triple-stores.

1.1.3 Real-time and Right-time

Today, business time is increasingly moving toward real time. As enterprises look to grow their competitive advantage, they are trying to uncover opportunities to capture and respond to business events faster and more rigorously than ever. The duration between the event and its consequent action needs to be minimized. Therefore, one of the emerging trends for data warehousing is the increasing demand for "real-time" or "near real-time" data integration, i.e., the refreshment of DW happens very quickly after a triggering business event [10, 17].

A recent and more sophisticated requirement is to make the data in the DW available *when* users need it, but not necessarily before. This is referred to as "right-time" data warehousing [85]. A user can then specify a certain required freshness for the data. For example, (s)he can specify that (s)he wants a freshness of five minutes. Thus, the system can ensure the user to read the data committed five minutes ago.

Real-time and right-time both collect and integrate information about actionable operational events, and emphasize a lower latency between the events and the decision makings. These events may originate from the sources– operational applications, web click streams and so forth. However, traditionally DWs are refreshed in a periodic manner, usually on a daily basis. The DW refreshment is typically scheduled for off-peak hours where the operational sources and the DW both experience low load conditions, e.g., at night-time. However, today's business users have an increasing demand for up-to-date data analysis to support their decision makings. The DW refreshment can no longer be postponed to off-peak hours, which necessitates investigating the novel data warehousing technologies that support near real-time and right-time data integration.

1.1.4 Support for Large-scale Data

With the evolution of information technologies, the volume of data we attempt to manage in data warehousing is exploding, especially processing the data generated by machines, such as online activities, mobile communications, or social media, etc. It has become common for an enterprise to process hundreds of gigabytes of data per day. Traditional data warehousing technologies face the challenge on how to process large-scale data efficiently as normally the data is processed in a regular interval, and within a certain time window, e.g., at night in every day. To meet this challenge, the architecture of the data warehousing must be carefully reconsidered. Parallelization is the technology that has long been studied, and widely used for improving data processing scalability, such as using powerful symmetric multiprocessing (SMP) machines, or using grid computing and in combination with software. However, these technologies have the limitations that they only have limited scalability, e.g., multithreading is not scaled out to many machines. In recent years, with the appearance of so-called "cloud computing" technology, a revolution of computing paradigm, *MapReduce* [22], becomes popular, which entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. MapReduce has been proved applicable and effective for a wide range of applications, and has become the de-facto standard for large-scale data. Thus, it would be very interesting to introduce MapReduce to data warehousing. For example, we can split the processing of dimensions and facts into smaller computation units, and merge the partial results from these computation units, and constitute the final results in a DW. The data warehousing can benefit from the off-the-shelf MapReduce system mechanisms for communication, load balance, task scheduling and fault tolerance.

1.1.5 Motivation for Cloud Computing

Traditionally, data warehouse systems have a shared-everything or shared-storage architecture with high-end machines. Nevertheless, this architecture limits the scalability when dealing with large-scale data today. The recent emerged cloud computing infrastructure differs dramatically from the infrastructure that most in-house data warehouse systems used today. Instead of using high-end machines, the infrastructure is composed of thousands of commodity servers. The computing resource in the cloud is thus very cheap, and provided as pay-as-you-go service. For example, Amazon EC2 [6] provides powerful on-demand computing resources that can be scaled up or down as needed, while paying per hour for the use of virtual machine instances. Cloud computing enables enterprises to analyze large-scale data faster and more economically than before. In addition, the data warehousing solutions can also be provided as a service to enterprises. Enterprises can make use of the solutions on a subscription basis such that the enterprises can reduce their operation cost. However, the current data warehousing systems are not yet particularly suitable for moving towards the cloud, e.g., typically, the ETL processes do not spread across multiple machine, and the system uses a central DW for storing data. Thus, we need to reconsider the existing data warehousing systems, and propose new DW technologies that are suitable for the cloud.

1.2 Thesis Overview

This section gives an overview of each chapter and the appendix in this thesis.

Chapter 2 introduces an efficient RDBMS-based OWL Lite triple-store. This work is motivated by the eGovMon project [27]. In this project, a large number of web pages are crawled to evaluate their accessibility regularly, which produces a large amount of RDF data sets for the evaluation results. A classical RDF data store, such as 3Store [37], stores RDF data sets in a narrow *triple table*, which has three columns, *subject, predicate* and *object*. However, when large amounts of data have been loaded, the size of the triple table becomes very large. This deteriorates query performance because querying a RDF graph has to do multiple self-joins of the giant triple table. In this chapter, we present an OWL Lite triple-store, 3XL, which supports efficient triple loading and query. 3XL uses a specialized database schema to store the triples. The database schema is derived from the ontology of describing triple data, and makes use of the object-relational features of PostgreSQL, such as inheritance. The triples are partitioned, and stored in many *class tables*. This chapter introduces a number of optimization techniques to speed up the loading of triples into the class tables, including in-memory value holders, partial-commit of the value holders, and bulk-loading the triples from the value holders to the underlying RDBMS. 3XL supports very efficient retrieval for point-wise queries where the subject and/or the predicate is known, as we have found such queries to be the most important for most bulk data management applications. 3XL also supports efficient retrieval for the composite queries which are composed of several point-wise queries for retrieving a complex RDF graph. We evaluate the loading and query performance using both real-world and synthetic data sets, and compare with the DBMS-based solutions and the two state-of-the-art file-based solutions, BigOWLIM and RDF-3X. The results show that the loading performance of 3XL outperforms RDF-3X, and is comparable to BigOWLIM (the current most efficient RDF/OWL store), e.g., when loading 100 million LUBM triples, the loading speed of 3XL is up to 25,000 triples/second on a normal notebook. When doing the 14 LUBM (the Lehigh University Benchmark for OWL knowledge base systems) queries [51] , 3XL shows the best performance for 6 out of the 14 queries while BigOWLIM has the best performance for 7 queries and RDF-3X has the best performance for the remaining query. Thus, 3XL has the comparable performance to the file-based solutions, but offers the flexibility of RDBMS-based solutions.

Chapter 3 presents a right-time ETL middleware for live DW data. Traditionally, data warehousing populates data from heterogeneous source systems into a DW by an ETL process periodically, e.g., daily or monthly. This, in many cases, is not acceptable as the time delay can make decision making lack far behind in time compared to the external business environments. Further, traditional ETL technologies appear incapable of handling so-called *live DW data*, in which data insertions are interleaved with updates and deletions. A typical example is the accumulating facts of which the fact records can be added with the incomplete information of dimensions. For instance, in online shopping a user places an order only with the information of the number of articles, prices, order date, etc. A sale fact record is created in the system, but this record still lacks the other information such as the delivery date and fee, which are accumulated gradually by the later updates. We found that most of the data modifications for live DW data happen shortly after the insertions, while updates are seldom seen for the data inserted long time ago, such as the history data. In the conventional approach, the updates are conducted in a regular interval and in off-line fashion, which is now regarded as not efficient enough. This chapter presents an ETL middleware, called *All-RiTE*, for handling live DW data. All-RiTE supports all the operations for processing live DW data, including the operations *INSERT, UPDATE, DELETE* and *SELECT*, on-the-fly data modifications, the support of processing multiple tables, and processing data in online fashion. All-RiTE exploits a novel memory-based data buffer, *catalyst*, between the sources and the DW. The live DW data is first accumulated in the catalyst, and modified on-the-fly when the data is queried or populated to the DW. The data in the catalyst is also available for read to users. Users can specify the time accuracy of the data to be read from the catalyst, and specify the flush policy *when* to move the data into the DW. This chapter compares All-RiTE with bulk-loading and JDBC. The experimental results show that All-RiTE achieves very good performance in processing live DW data. All-RiTE works 4, 3.9, 8.9 and 3.9 times faster than bulk-loading, RiTE, JDBC insert and JDBC batch loading, respectively when loading data with the on-the-fly modifications. When reading the data from the catalyst, the performance is comparable to reading data from the DW. All-RiTE, thus, can be used as a middleware to integrate with ETL programs that require processing live DW data in a near real-time/right-time fashion.

Chapter 4 describes a dimensional ETL programming framework using MapReduce [22]. With the ever-growing data today, data warehousing technologies face the challenges on how to process data efficiently. Parallelization is one of the key technologies to improve the scalability. Traditional parallelization technologies, such as Parallel Virtual Machine (PVM) [77] and Message Passing Interface (MPI) [30], provide the parallelization ability by using extensive messaging mechanisms. They are mostly designed for tackling processor-intensive problems, and complicated to use. The MapReduce paradigm, however, becomes popular for its high scalability, ease of use, fault-tolerance and cost-effectiveness. It is thus interesting to see how MapReduce can be applied to the parallelization of ETL. Nevertheless, MapReduce is only a generic framework, which lacks the support of the high-level ETL-specific constructs for different DW schemas, including star schema, snowflake schema and slowly changing dimensions (SCDs). This leads to low ETL programmer productivity. The presented framework, *ETLMR*, provides the direct support of the high-level ETL constructs. This makes it easy to implement a parallel ETL program, e.g., only a few code-lines are needed. The framework processes a DW schema using two sequential steps, i.e., dimension processing following by fact processing. This chapter first presents a number of parallelization methods for the dimension processing, including using one MapReduce task to process one dimension table exclusively, called *one dimension, one task (ODOT)*, using all the tasks to process one dimension table, called *one dimension, all tasks (ODAT)*, and using *level-wise* and *hierarchy-wise* approaches to process snowflaked dimensions. In addition, in order to achieve good efficiency, this chapter proposes an offline dimension scheme to process big dimension tables. For fact processing, this chapter proposes the method of using a map-only job to process fact data, which does dimension key value lookups and data transformations in mappers. In each mapper, the processed fact data is first cached in a task-specific buffer in run-time. When the buffer is full, the fact data is loaded into the DW using bulk-loader. The experiments of this chapter first study the scalability of each processing method, then compare the performance with the related work, Pentaho Data Integration (PDI) [62], and finally compare the programming effort with Hive [86] and Pig [57]. The results show that using ETLMR can achieve good scalability to process different dimensional DW schemas, and the performance outperforms PDI. ETLMR has good programming efficiency to implement a parallel ETL program, e.g., To process a partial snowflake dimension schema, ETLMR only needs 14 statements while HiveQL and Pig-Latin need 23 and 40 statements, respectively. ETLMR provides the built-in support for SCDs, which is not trivial for HiveQL and Pig-Latin since they both do not support updates.

Chapter 5 introduces a scalable dimensional ETL framework for cloud warehouse. Due to the rapid growth of data today, there is an increasing demand for the new data warehousing architectures that support scalable operations. However, most existing DW systems are the in-house systems, which are implemented for high-end machines, and store data in a relational DBMS. This architecture becomes insufficient to use for many enterprises when dealing with large-scale data. This chapter introduces a novel scalable dimensional ETL framework, *CloudETL*. Unlike the conventional data warehousing systems, CloudETL uses Hadoop to parallelize ETL execution and Hive as the warehouse system. This enables the data warehousing in cloud environments. In CloudETL, the source data in Hadoop distributed file system (HDFS) is processed into the dimension and fact tables in Hive through user-defined transformers (the components with ETL capabilities). When processing SCDs, we apply updates on both the incremental and existing dimension data that has already been loaded into Hive, i.e., update the effective dates and the version numbers of SCDs. This chapter first introduces the reduce-side updates for SCDs, which the dimension values with an identical business key value are shuffled to a single reducer, sorted and updated according to the changing order of SCDs. Since shuffling data from mappers to reducers is a relatively expensive operation on Hadoop, we optimize CloudETL by doing the pre-updates in mappers such that the size of shuffled data is minimized. CloudETL also provides the built-in support for data co-location in HDFS. When the data is co-located, only mappers are needed, e.g., when processing a big dimension table, such that the performance can be improved significantly. CloudETL processes snowflaked dimension tables through a number of dependent jobs which are planned based on the foreign-key dependencies of the tables. To process fact data, we introduce the lookup index, a sequence file containing the mappings of lookup attribute values to a dimension key value. When processing fact data, CloudETL first reads the lookup indices into the main memory, then retrieves dimension key values through multi-way lookups from the in-memory lookup indices. In addition, this chapter also introduces how to manage the input and output of CloudETL, and how to maintain the data consistency in case of job failures. The sequential number generation for dimension key values and the mechanism of planning jobs are introduced as well. The experimental results show that CloudETL achieves better performance than ETLMR when processing different DW schemas. It also outperforms the dimensional ETL capabilities of Hive, i.e., CloudETL has higher programming efficiency to implement parallel ETL programs for different DW schemas, e.g., only needs 6 statements for SCDs while Hive requires 112 statements, and the performance is also much better.

Appendix A demonstrates how to interact with 3XL system through a graphic user interface (GUI). We show how to generate data dependent database schema by a given OWL Lite ontology, show how to set different parameters to tune the performance of loading, and show how to query the OWL/RDF graphs by using point-wise and composite queries. Appendix B demonstrates the details on how to use ETLMR to implement parallel ETL programs for different DW schemas. We illustrate how easy it is to use ETLMR, e.g., only a few code lines are needed.

The thesis is organized as a collection of individual papers. Each chapter/appendix is self-contained and can be read in isolation. The chapters have been slightly modified during the integration such that, for example, their bibliographies have been combined to one, and references to “this paper” have been changed to references to “this chapter”. Appendix A and B are the demonstrations of the systems presented in Chapter 4 and 5, respectively. The content of each has some overlap to its corresponding chapter.

The papers included in this thesis are listed in the following. Chapter 2 is based on Paper 1, Chapter 3 is based on Paper 2, Chapter 4 is based on Paper 4 which is the complete version of Paper 3, Chapter 5 is based on Paper 5, Appendix A is based on Paper 6 and supplement to Chapter 2, and Appendix B is based on Paper 7 and supplement to Chapter 4.

1. X. Liu, C. Thomsen, and T. B. Pedersen. 3XL: Supporting efficient operations on very large OWL Lite triple-stores. *Information Systems*, 36(4): 765-781, 2011.

2. X. Liu, C. Thomsen, and T. B. Pedersen. All-RiTE: Right-Time ETL for Live DW Data. In *Preparation for Submission*, 12 pages, 2012.

3. X. Liu, C. Thomsen, and T. B. Pedersen. ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce. In *Proceedings of the 13th Data Warehousing and Knowledge Discovery*, pp. 96-111, 2011.

4. X. Liu, C. Thomsen, and T. B. Pedersen. ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce. In *Submission to Transactions on Large-Scale Data and Knowledge Centered Systems*, 30 pages, 2012.

5. X. Liu, C. Thomsen, and T. B. Pedersen. CloudETL: Scalable Dimensional ETL for Hadoop and Hive. In *Submission to Very Large Data Base Endowment*, 12 pages, 2012.

6. X. Liu, C. Thomsen, and T. B. Pedersen. 3XL: An Efficient DBMS-Based Triple-Store. To appear in *the 11th International Workshop on Web Semantics and Information Processing*, 5 pages, 2012.

7. X. Liu, C. Thomsen, and T. B. Pedersen. MapReduce-based Dimensional ETL Made Easy (demo). PVLDB 5(12):1882-1885, 2012.

Chapter 2

3XL: Supporting Efficient Operations on Very Large OWL Lite Triple-Stores

An increasing number of (semantic) web applications store a very large number of (subject, predicate, object) triples in specialized storage engines called triple-stores. Often, triple-stores are used mainly as plain data stores, i.e., for inserting and retrieving large amounts of triples, but not using more advanced features such as logical inference, etc. However, current triple-stores are not optimized for such bulk operations and/or do not support OWL Lite. Further, triple-stores can be inflexible when the data has to be integrated with other kinds of data in non-triple form, e.g., standard relational data.

This chapter presents 3XL, a triple-store that efficiently supports operations on very large amounts of OWL Lite triples. 3XL also provides the user with high flexibility as it stores data in an object-relational database in a schema that is easy to use and understand. It is, thus, easy to integrate 3XL data with data from other sources. The distinguishing features of 3XL include a) flexibility as the data is stored in a database, allowing easy integration with other data, and can be queried by means of both triple queries and SQL, b) using a specialized data-dependent schema (with intelligent partitioning) which is intuitive and efficient to use, c) using object-relational DBMS features such as inheritance, d) efficient loading through extensive use of bulk loading and caching, and e) efficient triple query operations, especially in the important case when the subject and/or predicate is known. Extensive experiments with a PostgreSQL-based implementation show that 3XL performs very well for such operations and that the performance is comparable to state-of-the-art triplestores.

2.1 Introduction

The increasing popularity of (semantic) web applications means that very large amounts of semantic web data, e.g., from ontologies, need to be managed. Most semantic web data is somehow based on the *Resource Description Framework* (RDF) [46], a family of *World Wide Web Consortium* (W3C) specifications for conceptual description/modeling of web resource information. Recently, the *Web Ontology Language* (OWL), a semantic markup language recommended by W3C for publishing and sharing ontologies on the WWW has gained popularity [11]. OWL is layered on top of RDF. Even the least expressive of the three OWL layers (OWL Lite) offers class hierarchies and constraints features, and is very useful for thesauri and other taxonomies. OWL (and RDF) data takes the form of (subject, predicate, object) triples. These triples are typically stored in specialized storage engines called *triple-stores*.

Our initial motivation for this work was the European Internet Accessibility Observatory (EIAO) project [82] where tens of millions of triples describing test results about the accessibility of web pages for people with various kinds of disabilities, e.g., blind people using a screen reader, were generated in the W3C standard EARL RDF language. Here, and in other projects, we have seen that the triple-stores are used mainly as specialized *bulk data stores*, i.e., for inserting and retrieving large amounts of triples (bulk operations). More advanced features such as logical inference etc., are often not used. Additionally, for the basic storage of data about OWL instances, we found that even a subset of the OWL Lite features was enough, namely classes, subclasses, object properties, data properties, domains, ranges, restrictions, onProperty, and maxCardinality. A well-known example of such data is the data generated by the data generator for the de-facto industry standard OWL data management benchmark Lehigh University Benchmark (LUBM) [34].

Similarly to many other projects, the EIAO project involved later integration of the collected EARL RDF data with other non-RDF data when a data warehouse (DW) was built to enable easy analysis of the accessibility results. To integrate data from triples with other kinds of data from relational databases, flat files, XML, etc. can be difficult. In the EIAO case, an extract-transform-load (ETL) application was hand-coded to execute triple queries, interprete triple results and do the many needed transformations to integrate the data into a relational DW. Thus, it was a design criteria for 3XL to allows easy integration with non-triple data.

In this chapter, we present the 3XL triple-store that, unlike most current OWL Lite triple-stores, is specifically designed to support bulk data management operations (load and retrieval) on very large OWL Lite triple-stores and provide the user with flexibility in retrieving the data. The “3” refers to triples, and the “XL” part to “eXtra Large” and “fLeXible” (“3XL” is also the largest standard t-shirt size in Europe). 3XL’s approach has a number of unique characteristics. First, 3XL stores data in a relational database meaning that the user has flexibility as queries can be expressed either in triples or in SQL. Second, for the database schema, 3XL uses a *specialized data-dependent schema* derived from the OWL ontology for the data to store, meaning that the schema is easy to navigate and understand and that an “intelligent partitioning” of data is performed. Third, 3XL uses advanced object-relational features of the underlying ORDBMS, namely table inheritance and the possibility to have arrays as in-lined attribute values. Fourth, 3XL makes extensive use of a number of *bulk-loading* techniques and caching to speed up bulk insertions significantly. Finally, 3XL is specifically designed to support efficient bulk retrieval for queries where the subject and/or the predicate is known, as such queries are the most important for most bulk data management applications. 3XL is implemented on top of the PostgreSQL ORDBMS.

Extensive performance experiments with both real-world and synthetic data shows that 3XL has load and query performance comparable to the best (file-based) triplestores, and that 3XL outperforms other DBMS-based triplestores. We believe this positions 3XL in a unique spot: performance comparable to the best file-based triplestores combined with the high degree of flexibility in accessing and integrating non-triple data offered by a DBMS-based triple-store.

The rest of the chapter is structured as follows. Section 2.2 introduces the 3XL system in general, explains how to generate a 3XL database schema from an input OWL ontology, and finally describes triple addition and queries. Section 2.3 presents the performance study. Section 2.4 presents related work. The last section concludes and provides ideas for future work.

2.2 The 3XL System

2.2.1 Overview

First, we informally describe the general idea about generating a specialized database schema for an OWL ontology in PostgreSQL. The descriptions give an intuition about how 3XL works before this is described in details.

To build the database to store the data in, an OWL ontology is read. This ontology should define all classes, their parent-child relationships and their properties (including domains and ranges). In the database, a *class table* is created for each class. The class table for the class C directly inherits from any class tables for the parent classes of C. This means that if the class table for C's parent P has the attributes a, b, c then the class table for C has at least the attributes a, b, c.

Two attributes are needed for each instance of any class: An ID and a URI. To have these available in all tables, all class tables – directly or indirectly – inherit from a single root class table that represents the OWL class `owl:Thing` that all other

OWL classes inherit from. The class table for `owl:Thing` has the columns ID and URI. All the ID values are for convenience unique integers drawn from the same database sequence (this is explained later).

If the class C has a `DataProperty` d with `maxCardinality` 1, the table for C has a column d with a proper datatype. For a *multiproperty* without a `maxCardinality`,[1] there is a special *multiproperty table*. This multiproperty table has a column that holds the attribute values and a column that holds the IDs for the instances the property values apply to. The ID attribute acts like a foreign key, but it is not declared (this is explained below). We here denote this as a *loose foreign key*. Note that a multiproperty table does not inherit from the class table for `owl:Thing` since multiproperty tables are not intended to represent instances, but only *values* for instances. A multiproperty has only one multiproperty table for a class C and not one for each subclass of C.

Instead of using multiproperty tables, the class tables can have columns that hold *arrays*. In this way it is possible to represent several property values for an instance in the single row that represents the instance in question.

An `owl:ObjectProperty` is handled similarly to how an `owl:DataProperty` is handled. If the object property has `owl:maxCardinality` 1, a column for the property is created in the appropriate class table. This column holds IDs for the referenced objects. If the property is a multiproperty, the value column in the multiproperty table holds ID values.

Example 1 *We now introduce the running example used in the rest of the chapter. To save space we do not use URIs but intuitive names for classes and properties.*

Assume that there are three classes: Document, HTMLVersion, and HTMLDocument, where HTMLDocument is a subclass of Document. Document has the properties title and keyword. The property keyword is the only multiproperty in this example. HTMLVersion has the properties version and approvalDate. Apart from the inherited properties, HTMLDocument has the property usedVersion. The property usedVersion is an `owl:ObjectProperty` with `owl:range` HTMLVersion. The remaining properties are all of kind `owl:DataProperty`.

This results in the database schema drawn in Figure 2.1. Inheritance is shown with arrows as in UML. A loose foreign key is shown as a dotted arrow. For now, please ignore the `map` and `overflow` tables which are explained later.

Note how easy the schema from Example 1 is to use in SQL queries. This makes it easy to integrate the data with other data. Further, the user can exploit all the advanced functionality that the underlying DBMS (PostgreSQL) provides. The user can, however, also choose to query the data by means of (single or “chained”) triples as will be explained later.

[1] OWL Lite only allows `maxCardinality` to be 0, 1, or unspecified.

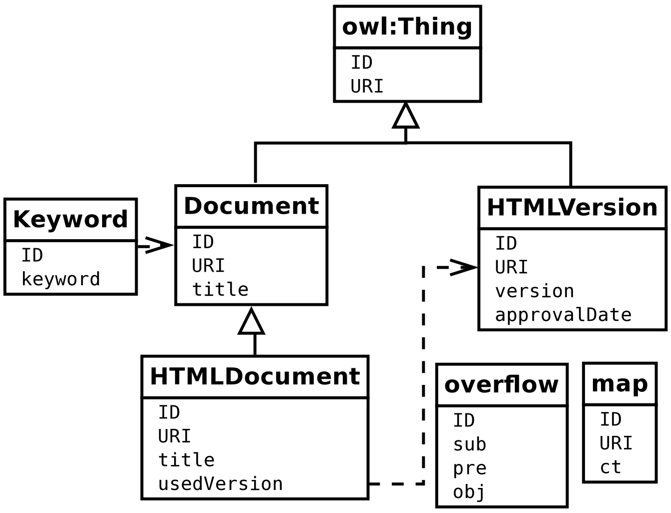


Figure 2.1: A database schema generated by 3XL

When triples are being inserted, 3XL has to find out which class the subject belongs to. This decides which class table to insert the data into. If the property name of the triple is unique among classes, it is easy to decide. Otherwise, 3XL tries to deduce the *most specific class* that the instance for sure belongs to. The subject may, however, be an instance of a class that is not described by the ontology used for the schema generation. In that case, the triple is placed in the `overflow` table.

To be efficient, 3XL does not insert data from a triple into the underlying database as soon the triple is added. Instead, the data is held in a buffer until larger amounts of data can be inserted quickly into the underlying database using bulk load mechanisms. To keep data in a buffer for a while also has the advantage that the type detection described above can make a more precise guess. It is a requirement in OWL Lite that there is a triple giving the `rdf:type` for each individual. Thus, a triple revealing the type should appear sooner or later and has often appeared when the actual insertion into the database takes place.

In Example 1 it may, however, happen that an instance i of the class Document that has been written to the class table for Document later turns out to actually be an instance of the class HTMLDocument. In that case, it is easy to move the row representing i from the class table for Document to the class table for HTMLDocument. Here it is convenient only to have one multiproperty table for keyword since no rows have to moved from the multiproperty table. This also shows why the foreign key from multiproperty tables has to be loose: It is unknown which class table the referenced ID value is located in. However, when querying for a specific ID value for an instance of Document in Example 1, it is enough to use the SQL expression `SELECT ID FROM Document WHERE`.... PostgreSQL then automatically also looks in descendant tables. This also shows why all IDs should be unique across tables and therefore are drawn from the same sequence.

A drawback of the approach where a row representing an instance is moved from one class table T to a class table for a subclass S, is that the subclass may put a `maxCardinality 1` restriction on a property p that in the superclass is a multiproperty. In this case, the multiproperty table for p is not needed to represent data for S instances. It is then possible to let p be represented by a column in S and not by the multiproperty table that has a loose foreign key to T. However, for simplicity we keep using the multiproperty table for p if it already exists and do not add an extra column for p to the class table for S.

When the triple-store is queried, it is done by issuing one or several combined (subject, predicate, object) triples, called *point-wise queries* and *composite queries*, respectively (in addition, the user has the possibility to use SQL queries as the data is stored in a relational database). If a property is given in a point-wise query, this can reveal which class table(s) to look into. If only a subject or object is given, it is possible to look up the URI in the `owl:Thing` class table. This is, however, potentially very expensive, so 3XL, in addition to the previously mentioned class tables, also has a `map` table that maps from a URI to the class table that holds the instance with that URI. For each query triple, the `overflow` table is also searched by 3XL. In comparison to point-wise queries, composite queries are more expressive, as they are conjunctive combinations of several point-wise query triples.

In summary, the idea is to have the data spread over many tables (with many columns). It is fast to find data when the table to look in can be identified easily ("intelligent partitioning"). The tables also have very good potential for being indexed. Indexes may be added on attributes that are often used in queries. We are now ready to give a detailed description of how the specialized database schema of 3XL is generated. After that, we describe how additions to the triple-store are handled. This is followed by a description of how queries are handled.

2.2.2 Schema Generation

In the following, we describe the handling of the supported OWL constructs when the specialized database schema is generated. To generate the database schema, 3XL reads an ontology and builds a model of the classes including their properties and subclass relationships. Based on the built model, SQL DDL statements to create tables are generated and executed. Note that this SQL is not conforming to the SQL standard since it uses PostgreSQL's object-oriented extensions (see more below). In the following, we focus on the resulting schema.

Note that a database schema generated by 3XL always has the table `map(ID, URI, ct)`. As explained later, this table is used to make it fast to find the table that represents a given instance and the ID of the instance. The table `overflow(ID, sub, pre, obj)` is also created in each generated schema. This table holds triples that do not fit in the other tables.

We are now ready to describe how the constructs of OWL Lite are handled in the generation of a specialized database schema. We assume that a database schema D is being generated for the OWL ontology O.

owl:Class An `owl:Class` in O results in a table, called a *class table*, in D. In the following, we denote by $\mathcal{C}_X$ the class table in D for the class X in O. $\mathcal{C}_X$ is used such that for each instance of X that is not an instance of a subclass of X and for which data must be stored in the triple-store, there is exactly one row in $\mathcal{C}_X$. Each represented instance has a URI and is given a unique ID by 3XL.

The special class table $\mathcal{C}_{\texttt{owl:Thing}}$ for `owl:Thing` is always created in D. This special class table does not inherit from any other table and has two columns named ID (of type INTEGER) and URI (of type VARCHAR). Any other class table created in D always inherits from one or more other class tables (see below) and always inherits – directly or indirectly – from $\mathcal{C}_{\texttt{owl:Thing}}$. This implies that the columns ID and URI are available in each class table.

For other class tables than $\mathcal{C}_{\texttt{owl:Thing}}$, other columns may also be present: A class table for a class that is in the `rdfs:domain` of some property P and is a subclass of a restriction saying the `owl:maxCardinality` of the property is 1, also has a column for P. This column is only explicitly declared in the class table for the most general class that is the domain of the property. But class tables inheriting from that class table automatically also have the column. For an example of this, refer to Example 1 where a column for title is declared in the class table for Document.

rdfs:subClassOf For classes X and Y in O where Y is a subclass of X (i.e., the triple (Y, `rdfs:subClassOf`, X) exists in O), there exist class tables $\mathcal{C}_X$ and $\mathcal{C}_Y$ in D as explained above. But $\mathcal{C}_Y$ is declared to inherit from $\mathcal{C}_X$ and thus has at least the same columns as $\mathcal{C}_X$. This resembles the fact that any instance of Y is also an instance of X. So when rows are read from $\mathcal{C}_X$ to find data about X instances, PostgreSQL also reads data from $\mathcal{C}_Y$ since the rows there represent data about Y instances (and thus also X instances). In Example 1, $\mathcal{C}_{\textsf{HTMLDocument}}$ inherits from $\mathcal{C}_{\textsf{Document}}$ since HTMLDocument is a subclass of Document.

Any class X defined in O that is not a subclass of another class implicitly becomes a subclass of `owl:Thing`. Thus, if no other parent is specified for X, $\mathcal{C}_X$ inherits from $\mathcal{C}_{\texttt{owl:Thing}}$ as do $\mathcal{C}_{\textsf{Document}}$ and $\mathcal{C}_{\textsf{HTMLVersion}}$ in the running example.

owl:ObjectProperty and owl:DataProperty A property (no matter if it is an `owl:`
`ObjectProperty` or `owl:DataProperty`) results in a column in a table. If

the property is an `owl:ObjectProperty`, the column is of type INTEGER such that it can hold the ID for the referenced instance. If the property on the other hand is an `owl:DataProperty`, the column is of a type that can represent the range of the property, e.g., VARCHAR or INTEGER.

If the `owl:maxCardinality` is 1, the column is placed in the class table for the most general class in the `rdfs:domain` of the property. Since there is at most one value for each instance, this makes it efficient to find the data since no joining is needed and one look-up in the relevant class table can find many property values for one instance.

If no `owl:maxCardinality` is specified, there may be an unknown number of property values to store for each instance and the idea about storing one property value in a column in the class table breaks. Instead, a column with an array type can be used. Another solution is to create a *multiproperty table*. Each row in the multiproperty table represents one value for the property for a specific instance. In a multiproperty table there are two columns: One to hold the ID of the instance that the represented property value applies to and one for the property value itself. This approach is illustrated for the keyword property in Example 1. In 3XL, it is left as a configuration choice if multiproperty tables or array columns should be used for multiproperties.

rdfs:domain The `rdfs:domain` for a property decides which class table to place the column for the property in in case it has a `owl:maxCardinality` of 1 or in case that array columns are used instead of multiproperty tables. In either case, the column to hold the property values is placed in $\mathcal{C}_T$ where T is the domain.

If multiproperty tables are used and no `owl:maxCardinality` is given, the `rdfs:domain` decides which class table holds (directly or indirectly in a descendant table) the instances for which the property values are given. In other words, this decides where one of the IDs referenced by the multiproperty table exists. Note that no foreign key is declared in D. To understand this, recall that since there is only one multiproperty table for the given property, the most specific type of an instance that has this property may be different from the most general. So although the property has domain X, another class Y may be a subclass of X, and Y instances can be referenced by a property with range X. An example of this is seen in the running example, where keyword is defined to have the domain Document, but an HTML-Document can also have keyword values. So in general there is not only one class table representing the range. Therefore we use a *loose foreign key*. A loose foreign key LFK_ℓ from $\mathcal{C}_X$ to $\mathcal{C}_Y$ is a column ℓ_X in $\mathcal{C}_X$ and a column ℓ_Y in $\mathcal{C}_Y$ with the constraint that if a row in $\mathcal{C}_X$ has the value v for ℓ_X, then at least one row in $\mathcal{C}_Y$ or a descendant table of $\mathcal{C}_Y$ has the value v in the column ℓ_Y. The crucial point here compared to a normal foreign key, is that the referenced value does not have to be in $\mathcal{C}_Y$, but can instead be in one of $\mathcal{C}_Y$'s descendants. Note that a loose foreign key is not enforced by the DBMS; this is left to 3XL to do. If no domain is given in O for the property, it is implicitly assumed to be `owl:Thing`.

rdfs:range The `rdfs:range` is used to decide where to find referenced instances for an `owl:ObjectProperty` and to decide the data type of the column holding values for an `owl:DataProperty`. So, similarly to the case explained above, the range decides which table the other ID of a multiproperty table for an object property references by a loose foreign key. Further, when the range of a property p is known, the object of a triple where the predicate is p, can have its `rdf:type` inferred (although in OWL Lite, it also has to be given explicitly).

owl:Restriction (including owl:onProperty and owl:maxCardinality) In OWL, the way to say that a class C satisfies a certain condition, is to say that C is a subclass of C' where C' is the class of all objects that satisfy the condition [7]. The C' class can be an anonymous class. To construct an anonymous class for which conditions can be specified, the `owl:Restriction` construct is used. For an `owl:Restriction`, a number of things such as `owl:maxCardinality` can be specified.

Following the previous explanations about classes and subclasses this would lead to generating class tables for anonymous restrictions when 3XL generates the database schema. But since all instances of C' (which is actually anonymous) would also be instances of the non-anonymous class C and $\mathcal{C}_{C'}$ would thus be empty, this is more complex than needed. Instead, when 3XL generates the database schema, supported restrictions are "pulled down" to the non-anonymous subclass. So if the restriction C' of which C is a subclass, defines the `owl:maxCardinality` to be 1 for the property P by means of `owl:onProperty`, this means that P can be represented by a column in $\mathcal{C}_C$ and that no class table is generated for C'.

Currently, 3XL's restriction support is limited as only cardinality constraints are handled. As previously described, an `owl:maxCardinality` of 1 results in a column in a class table. Thus we assume that a property with `max:Cardinality` 1 only occurs once for a given subject. This deviates from the OWL semantics where it for a property p with `owl:maxCardinality` 1 can be deduced that o_1 and o_2 are equivalent if the both the triples (s, p, o_1) and (s, p, o_2) are present.

The following table summarizes how OWL constructs from the ontology O are mapped into the database schema D.

The construct ...	**results in ...**
owl:Class	a class table
rdfs:subClass	the class table for the subclass inherits from the class table for the superclass
owl:ObjectProperty or owl:DataProperty	a column in a class table if the owl:maxCardinality is 1 and in a multiproperty table otherwise
rdfs:domain	a column for the property in the class table for the domain if the owl:maxCardinality is 1 and a loose foreign key from a multiproperty table to a class table otherwise.
rdfs:range	a type for the column representing the property.

3XL thus supports a subset of OWL Lite. This subset is enough to represent the real-life semantic data from the EIAO project which served as our initial motivation for 3XL. Later, support for more OWL Lite constructs can be added. For example, we envision that support for `owl:sameAs` could be implemented by representing sameAs-relationships explicitly in a table (not a class table, but a table managed by 3XL similarly to the `map` table). Queries would, however, then have to be rewritten if they involve an instance which is the sameAs another instance. Also, the construct `owl:equivalentClass` could be supported by letting 3XL maintain a mapping (likely in memory for efficiency) between classes and the classes that are physcially represented by a class table.

2.2.3 Addition of Triples

We now describe how 3XL handles triples that are inserted into a specific *model* M which is a database. M has the database schema D which has been generated as described above from the ontology O_S with schematic data. We assume that the triples to insert are taken from an ontology O_I which only contains data about instances, and not schematic data about classes etc. Note that O_I can be split up into several smaller sets such that $O_I = O_{I_1} \cup \cdots \cup O_{I_n}$ where each O_{I_i}, $i = 1, \ldots, n$, is added at a different time. In other words, unlike schema generation which happens only once, addition of triples can happen many times.

First, we focus on the state of M after the addition of the triples in O_I to give an intuition for the algorithms that handle this. Then, we present pseudocode in Algorithms 1–3 and explain the handling of triple additions in more details.

If the subject of a triple is an instance of a class that is not described by O_S, the triple is represented in the `overflow` table. Assume in the following that the subjects of the triples to insert are instances of classes described by O_S.

When a triple (s, p, o) is added to M, 3XL has to decide in which class table and/or multiproperty table to put the data from the triple. Typically, the data in a triple becomes part of a row to be inserted into M. For each different s for which a triple $(s, \texttt{rdf:type}, t)$ exists[2] in O_I, the row $\mathcal{R}_s$ that is made from the triples with the common s is inserted into $\mathcal{C}_t$.

We now consider the effects of adding a triple (s, p, o) where p is a property defined in O_S. First, assume that p is declared to have `owl:maxCardinality` 1. Then $\mathcal{R}_s$'s column for p in $\mathcal{C}_t$ gets the value $\nu(p, o)$ which equals o if p is an `owl:DataProperty` or equals the value of the ID attribute in $\mathcal{R}_o$ if p is an `owl:ObjectProperty`. In other words, the value of a data property is stored directly whereas the value of an object property is not stored as a URI but as the (more efficient) integer ID of the referenced object.

Now assume that no `owl:maxCardinality` is given for p. As previously mentioned, such properties can be handled in two ways. If array columns are used, the situation resembles that of a property with a maximal cardinality of 1. The only difference is that the column for p in $\mathcal{R}_s$ does not get its value set to $\nu(p, o)$. Instead the value of $\nu(p, o)$ is added to the array in the column for p in $\mathcal{R}_s$. If multiproperty tables are used, the row $\big(\iota, \nu(p, o)\big)$ where ι is the value of the ID attribute in $\mathcal{R}_s$ is added to the multiproperty table for p. In other words, the row that is inserted into the multiproperty table has a reference (by means of a loose foreign key) to the row $\mathcal{R}_s$. Further, it has a reference to the row for the referenced object if p is an `owl:ObjectProperty` and otherwise the value of the property.

So for properties defined in O_S, the values they take in O_I are stored explicitly in columns in class tables and multiproperty tables. For other triples, information is not stored explicitly by adding a row. If the predicate p of a triple (s, p, o) is `rdf:type`, this information is stored implicitly since this triple does not result in a row being added to M, but decides in which class table $\mathcal{R}_s$ is put.

The pseudocode listed in Algorithms 1–3 shows how addition of triples is handled. For a so-called *value holder* vh (we will explain it next), we denote by $vh[x]$ the value that vh holds for x. We let the value holders hold lists for multiproperties and denote by $\circ$ the concatenation operator for a list.

When triples are being added to M, 3XL may not immediately be able to figure out which table to place the data of the triple in. For this reason, and to exploit the speed of bulk loading, data to add is temporarily held in a *data buffer*. Data from the data buffer is then, when needed, flushed into the database. This is illustrated in Figure 2.2.

The data buffer does not hold triples. Instead it holds *value holders* (see Algorithm 1, line 1 and Algorithm 2). So for each subject s of triples that have data in the data buffer, there is a value holder associated with it. In this value holder, an associative array maps between property names and values for these properties. In other words, the associative array for s reflects the mapping $p \mapsto \nu(p, o)$. Note that

[2] Recall that the type must be explicitly given.

Algorithm 1 AddTriple

```
Input: A triple $(s, p, o)$
 1: $vh \leftarrow$ GetValueHolder($s$)
 2: if $p$ is defined in $O_S$ then          ▷ $O_S$ is the ontology describing triple data
 3:     if domain($p$) is more specific than $vh$[rdf:type] then
 4:         $vh$[rdf:type] $\leftarrow$ domain($p$)
 5:     if maxCardinality($p$) $= 1$ then
 6:         $vh[p] \leftarrow$ Value($p, o$)
 7:     else
 8:         $vh[p] \leftarrow vh[p] \circ$ Value($p, o$)
 9: else if $p =$ rdf:type and $o$ is more specific than $vh$[rdf:type] and $o$ is described
    by $O_S$ then
10:     $vh$[rdf:type] $\leftarrow o$
11: else
12:     Insert the triple into overflow
```

if the predicate p of a triple (s, p, o) is `rdf:type`, $p \mapsto o$ is also inserted into the associative array in the value holder for s unless the associative array already maps `rdf:type` to a more specific type than o. Actually, 3XL infers triples of the form $(s,$ `rdf:type`$, o)$ based on predicate names, but only the most specialized type is stored (Algorithm 1 lines 3–4). This type information is later used to determine where to place the values held by the value holder. For a multiproperty p, the associative array maps p to a list of values (Algorithm 1, line 8) but for a property q with a maximal cardinality of 1, the associative array maps q to a scalar value (Algorithm 1, line 6). Further, 3XL assigns a unique ID to each subject which is also held by the value holder (Algorithm 2, line 19 when the value holder is created).

Example 2 (Data buffer) *Assume that the following triples are added to an empty 3XL model M for the running example:*
- (http://example.org/HTML-4.0, version, "4.0")
- (http://example.org/HTML-4.0, approvalDate, "1997-12-18")
- (http://example.org/programming.html, title, "How to Code?")
- (http://example.org/programming.html, keyword, "Java")
- (http://example.org/programming.html, keyword, "programming")
Before the triples are inserted into the underlying database by 3XL, the data buffer has the following state.

http://example.org/HTML-4.0		
ID	↦	*1*
rdf:type	↦	*HTMLVersion*
version	↦	*4.0*
approvalDate	↦	*1997-12-18*

http://example.org/programming.html		
ID	↦	*2*
rdf:type	↦	*Document*
title	↦	*How to Code?*
keyword	↦	*[programming, Java]*

Algorithm 2 GetValueHolder

```
Input: A URI $u$ for an instance
if the data buffer holds a value holder $vh$ for $u$ then
    return $vh$
else
    $table \leftarrow$ The class table holding $u$ (found from map)
    if $table$ is not NULL then
        /* Read values from the database */
        $vh \leftarrow$ new ValueHolder()
        Read all values for $u$ from $table$ and assign them to $vh$.
        Delete the row with URI $u$ from $table$
        for all multiproperty tables $mp$ referencing $table$ do
            Read all property values in rows referencing the row for $u$ in $table$ and assign these values to $vh$
            Delete from $mp$ the rows referencing the row with URI $u$ in $table$
        Add $vh$ to the data buffer
        return $vh$
    else
        /* Create a new value holder */
        $vh \leftarrow$ new ValueHolder()
        $vh$[URI] $\leftarrow u$
        $vh$[ID] $\leftarrow$ a unique ID
        $vh$[rdf:type] $\leftarrow$ owl:Thing
        Add $vh$ to the data buffer
        return $vh$
```

Algorithm 3 Value

```
Input: A property $p$ and an object $o$
if $p$ is an owl:ObjectProperty then
    $res \leftarrow$ the ID of the instance with URI $o$ (found from map)
    if $res$ is NULL then
        $res \leftarrow$ (GetValueHolder($o$))[ID]
    return $res$
else
    /* It is an owl:DataProperty */
    return $o$
```

Here the top row of a table shows which subject, the value holder holds values for. The following rows show the associative array. Note that the type for http://example.org /programming.html is assumed to be Document *since this is the most general class in the domains of* title *and* keyword*.*

Now assume that the triple (http://example.org/programming.html, usedVersion, http://example.org/HTML-4.0) is added to M. Then the type detection finds that

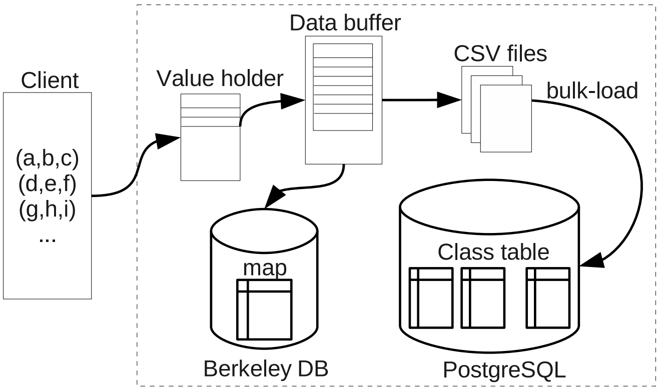


Figure 2.2: Data flow in 3XL

http://example.org/programming.html must be of type HTMLDocument*, so its value holder gets the following state.*

http://example.org/programming.html		
ID	↦	*2*
rdf:type	↦	*HTMLDocument*
title	↦	*How to Code?*
keyword	↦	*[programming, Java]*
usedVersion	↦	*1*

Note how the value holder maps usedVersion *to the ID value for http://example.org /HTML-4.0, not to the URI directly. If the required* `rdf:type` *triples now are inserted, this does not change anything since the type detection has already deduced the types.*

Due to the definition of ν described above, the value holders and eventually the columns in the database hold IDs of the referenced instances for object properties. But when triples are added, the instances are referred to by URIs. So on the addition of the triple (s, p, o) where p is an object property, 3XL has to find an ID for o, i.e., $\nu(p, o)$. If o is not already represented in M, a new value holder for o is created (Algorithm 3, line 4). Depending on the range of p, type information about o may be inferred. If o on the other hand is already represented in M, its existing ID should of course be used. It is possible to search for the ID by using the query `SELECT id FROM` $\mathcal{C}_{\texttt{owl:Thing}}$ `WHERE uri =` o. However, for a large model with many class tables and many rows (i.e., data about many instances) this can be an expensive query. To make this faster, 3XL maintains a table `map(uri, id, ct)` where `uri` and `id` are self-descriptive and `ct` is a reference to the class table where the instance is represented. Whenever an instance is inserted into a class table $\mathcal{C}_X$, the instance's URI and ID and a reference to $\mathcal{C}_X$ are inserted into map. By searching the data buffer and the map table, it is fast to look up if an instance is already represented and to get its ID if it is. The map table exists in the PostgreSQL database, but for performance reasons 3XL does not query/update the map table in the database while adding triples. Instead, 3XL only extracts all rows in the table once when starting a load of triples and places them in a temporary BerkeleyDB database [12] which acts like a cache. With BerkeleyDB it is possible to keep a configurable amount of the data in memory and efficiently and transparently write the rest to disk-based storage.

Similarly, 3XL also needs to determine if the instance s is already represented when adding a triple (s, p, o). Again the map table is used. If s is not already represented, a new value holder is created and added to the data buffer. If s on the other hand is represented, a value holder is created in the data buffer and given the values that can be read from the class table referenced from map and then $\mathcal{R}_s$ and all rows referencing it from multiproperty tables are deleted. In this way, it is easy to get the new and old data for s written to the database as data for s is just written as if it was all newly inserted. This also helps, if due to newly added data it becomes evident that s has a more specialized type than known before. In our implementation, the deletions are not done immediately as shown in the pseudocode. For a better performance, we invoke one operation deleting several rows before inserting new data.

When the data buffer gets full, a part of data in the data buffer is inserted into the database. This is done in a bulk operation where PostgreSQL's very efficient COPY mechanism is used instead of INSERT SQL statements. So the data gets dumped from the data buffer to temporary files in comma-separated values (CSV) format and the temporary files are then read by PostgreSQL. The rdf:types read from the value holders are used to decide which tables to insert the data into. In case, no type is known, owl:Thing is assumed. For unknown property values, NULL is inserted. If multiproperty tables are used, values from a multiproperty are inserted into these instead of a class table.

To exploit that the data might have *locality* such that triples describing the same instance appear close to each other, a *partial-commit* mechanism is employed, in which the least recently used $m\%$ of the data buffer's content is moved to the database when the data buffer gets full (the percentage m is user-configurable). This is illustrated in Figure 2.3. In this way, the system can in many cases avoid reading in the data just written out to the database.

2.2.4 Triple Queries

In this section, we describe the two types of queries, point-wise queries and composite queries, which are implemented in 3XL.

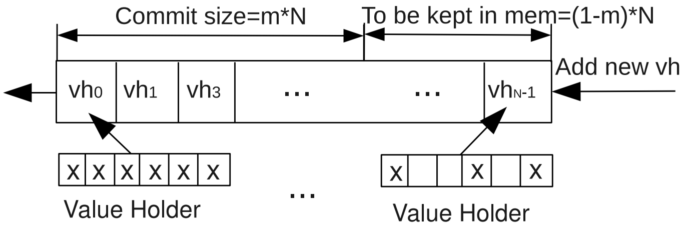


Figure 2.3: The data buffer

2.2.4.1 Point-wise Queries

A point-wise query is a triple, i.e., $Q = (s, p, o)$. Any of the elements in the query triple can take the special value $*$ to match anything. We consider how 3XL handles point-wise queries on the triples in a model M. Since schematic information given in O_S (for which the specialized schema was generated) is fixed, we do not consider queries for schematic information here. Instead we focus on queries for instance data inserted into M, i.e., queries for data in O_I. The result of a query consists of those triples in M where all elements *match* their corresponding elements in the query triple. The special $*$ value matches anything, but for all elements in Q different from $*$, all corresponding elements in a triple $T \in M$ must be identical for T to be included in the result.

Example 3 (Point-wise query) *Consider again the triples that were inserted in Example 2 and assume that only those (and the required triples explicitly giving the* `rdf:type`*) were inserted into M. The result of the query $(*, keyword, *)$ is the set holding the following triples:*
- (http://example.org/programming.html, keyword, Java)
- (http://example.org/programming.html, keyword, programming)
The result of the query $($http://example.org/HTML-4.0$, *, *)$ is the set holding the following triples:
- (http://example.org/HTML-4.0, rdf:type, owl:Thing)
- (http://example.org/HTML-4.0, rdf:type, HTMLVersion)
- (http://example.org/HTML-4.0, approvalDate, 1997-12-18)
- (http://example.org/HTML-4.0, version, 4.0)
i.e., the set containing all the knowledge about http://example.org/HTML-4.0, including all its known types.

For each query, the `overflow` table is searched and the result set of this is unioned with the results of searching the class and multiproperty tables. The `overflow` table is considered with a single SQL statement where all overflow triples with match-ing values are found. In the remaining descriptions, we focus on how the class and multiproperty tables are used to find the remaining triples of the result set.

As there are three elements in the query triple Q and each of these can take an ordinary value or the special value $*$, there are $2^3 = 8$ generic cases to consider. We go through each of them in the following. s, p, and o are all values different from $*$. When we for a subject s say that the class table that holds s is found, it is implicitly assumed that some class table actually holds s. If this is not the case, the result is of course just the empty set. Further, we assume that all data (including `map`'s data) is inserted into the database before the queries are executed.

Case (s, p, o) In this case, the query is for the query triple itself, i.e., the result set is either empty or consists exactly of the query triple. If p equals `rdf:type`, the result is found by looking in the `map` table to see if the class table holding s is $\mathcal{C}_o$ or a descendant of $\mathcal{C}_o$. This is done by using the single SQL query `SELECT ct FROM map WHERE uri =` s which can be performed fast if there is an index on `map(uri, ct).` If s is held by $\mathcal{C}_o$ or a descendant of $\mathcal{C}_o$, Q is returned and otherwise an empty result is returned.

If p is different from `rdf:type`, the result is found by finding the ID for s (from now called $s$id) and the class table where s is inserted (by means of `map`). If that class table has a column or a multiproperty table for p, it is determined if the property p takes the value o for s. To determine this, it is necessary to look for $\nu(p, o)$ in the database as an ID is stored instead of a URI for an `owl:ObjectProperty`. If p takes the value o for s, Q is returned, otherwise the empty result is returned. So this requires an SQL query selecting the class table (if p is represented by a column) or the ID (if p is represented by a multiproperty table) from `map` and either the query `SELECT true FROM classtable WHERE id =` s`id AND` p`column =` $\nu(p, o)$ (if p is not a multiproperty), the query `SELECT true FROM classtable where id =` s`id AND` $\nu(p, o)$ `= ANY(`p`column)` (if p is a multiproperty represented by an array column), or the query `SELECT true FROM` p`table WHERE id =` s`id AND value =` $\nu(p, o)$ (if p is a multiproperty represented by a multiproperty table). In any case, only 2 SQL SELECT queries are needed and – except when p is represented by an array column – indexes on the ID and p columns can help to speed up these queries.

Example 4 (Finding a specific triple) *Let Q =(http://example.org/programming.html, keyword, programming) be a query given in the running example. To answer this query, 3XL executes the following SQL queries since* keyword *is represented by a multiproperty table.*

```
SELECT id FROM map
WHERE uri = 'http://example.org/programming.html'
```

```
SELECT true FROM keywordTable
WHERE id = $id AND value = 'programming'
```

The result from the database is `true` *so the triple exists in the model and 3XL returns Q itself as the result.*

Case $(s, p, *)$ Also in this case, there is special handling of the situation where $p =$ `rdf:type`. Then, the `map` table is used to determine the class table $\mathcal{C}_X$ where s is located. The result set consists of all triples (s, p, C) where C is the class X or an ancestor class of X. So the only needed SQL query is `SELECT ct FROM map WHERE uri =` s. Based on the result of this and its knowledge about class inheritance, 3XL generates the triples for the result.

If p is an `owl:DataProperty`, the class table holding s is found. From this, the row representing s is found and each value for p is read. If p is a multiproperty and multiproperty tables are used, the values for p are found in the multiproperty table instead by using the ID for s as a search criterion. The result set consists of all triples (s, p, V) where V is a p value for s. Again, only 2 SQL SELECTs are needed: One querying `map` and one querying for the value(s) for p from either the class table or the multiproperty table for p. Indexes on `(uri, ct)` and `(uri, id)` in `map` and on the `id`s in the class table/multiproperty table will help to speed up these queries.

If p is an `owl:ObjectProperty`, special care has to be taken as the URIs of the referenced objects should be found, not their IDs. The first step is to find the class table $\mathcal{C}_X$ holding s and the ID of s by means of single SELECT on the `map` table. Assume WLOG that the range of p is R. If p is represented by the column p`column` in $\mathcal{C}_X$, the query `SELECT` $\mathcal{C}_R$`.uri FROM` $\mathcal{C}_X$`,` $\mathcal{C}_R$ `WHERE` $\mathcal{C}_X$`.`p`column =` $\mathcal{C}_R$`.id AND` $\mathcal{C}_X$`.id =` s`id` is used. If p is represented by a multiproperty table `mp`, $\mathcal{C}_R$ is joined with `mp` instead of $\mathcal{C}_X$. If p is a multiproperty represented by an array column, $\mathcal{C}_X$ and $\mathcal{C}_R$ are still joined, but the condition to use is `WHERE` $\mathcal{C}_R$`.id = ANY(`$\mathcal{C}_X$`.`p`column) AND` $\mathcal{C}_X$`.id =` s`id`. The result set holds all triples (s, p, U) where U ranges over the selected URIs.

Case $(s, *, o)$ In this case, the class table holding s is found. Then all property values (including values in multiproperty tables) are searched. The result set consists of all triples (s, P, o) where P is a property that takes the value o for s. So by iterating over the properties defined for the class that s belongs to, the previous (s, p, o) case can be used to find the triples to include. Note that also the special case $(s,$ `rdf:type`$, o)$ should be considered for inclusion in the result set. So for this query type, an SQL query selecting the class table and the ID from `map` is needed. Further, the SQL query `SELECT true FROM mp WHERE ID =` s`id AND value =` $\nu(p, o)$ is needed for each multiproperty table `mp` representing a property p defined for s's class as is the SQL query `SELECT true FROM classtable WHERE id =` s`id AND pc =` $\nu(p, o)$ for each column `pc` representing a property p for s in the class table holding s.

Case $(s,*,*)$ In this case, the class table holding s is found by using `map`. For each property P defined in O_S, each of its values V for s is found. The result set consists of all triples (s,P,V) unioned with the triples in the result set of the query $(s,$ `rdf:type`$,*)$.

In this case the following SQL queries are needed: One selecting the class table and ID from `map`, the query `SELECT` p_1`column, ... ,` p_n`column FROM classtable WHERE id=` s`id` if there are columns representing data properties $p_1,\ldots,p_n$ in the class table holding s, and a query `SELECT value FROM mp WHERE id =` s`id` for each multiproperty table `mp` representing a data property defined for s's class. Again, indexes on the `id` attributes in the class tables and multiproperty tables speed up the queries. Further, SQL to find the URIs for the values of object properties is needed. So for each object property q defined for the class that s belongs to and which is not represented by an array column, the following query is used: `SELECT` $\mathcal{C}_R$`.uri FROM` $\mathcal{C}_R$`,` Φ `WHERE` $\mathcal{C}_R$`.id =` Φ`.`q`column AND` Φ`.id =` s`id`. Here Φ is a multiproperty table for q or the class table holding s and R is the range of q. Indexes on the `id` attributes will again speed up the queries. If q is represented by an array column, $\mathcal{C}_R$`.id = ANY(`Φ`.`q`column)` should hold instead of $\mathcal{C}_R$`.id =` Φ`.`q`column`.

Case $(*,p,o)$ If p equals `rdf:type`, the class table $\mathcal{C}_o$ is found and all URIs are selected from it (including those in descendant tables). The result set consists of all triples (U,p,o) where U ranges over the found URIs. This requires only 1 SQL query: `SELECT uri FROM` $\mathcal{C}_o$.

If p is different from `rdf:type`, 3XL must find the most general class G for which p is defined. If p is represented by a multiproperty table X, the tables X and $\mathcal{C}_G$ are joined and restricted to consider the rows where the column for p takes the value $\nu(p,o)$ and the URIs for these rows are selected by the query `SELECT uri FROM X,` $\mathcal{C}_G$ `WHERE X.id =` $\mathcal{C}_G$`.id AND value =` $\nu(p,o)$. If p is represented in a column in $\mathcal{C}_G$, all URIs for rows that have the value $\nu(p,o)$ in the column for p (either as an element in case p is a multiproperty represented by an array column or as the only value in case p is not a multiproperty) are selected. This is done by using either the query `SELECT uri FROM` $\mathcal{C}_G$ `WHERE` p`column =` $\nu(p,o)$ or the query `SELECT uri FROM` $\mathcal{C}_G$ `WHERE` $\nu(p,o)$ `= ANY(`p`column)`. The result set consists of all triples (U,p,o) where U ranges over the selected URIs. The first of these queries benefits from an index on the column holding data for p, but for the latter a scan is needed as we are only looking for a particular value inside an array.

Example 5 (Find subjects from a $(*,p,o)$ **query)** *Consider the running example and assume that 3XL is given the query $Q = (*$, keyword, programming). The most general class for which* keyword *is specified is* Document *so the SQL query* `SELECT uri FROM keywordTable,` $\mathcal{C}_{\text{Document}}$ `WHERE keywordTable.id =`

$\mathcal{C}_{\mathit{Document}}$.`id AND value = 'programming'` is executed. One URI is found by the query, so the triple (http://example.org/programming.html, keyword, programming) is returned by 3XL.

Case $(*, p, *)$ If p in this case equals `rdf:type`, the result set contains all triples describing types for all subjects in the model. So for each class table $\mathcal{C}_X$, all its URIs (including those in subtables) are found with the SQL query `SELECT uri FROM` $\mathcal{C}_X$ which performs a scan of $\mathcal{C}_X$ and its descendants. The result set consists of all triples (U, p, X) where U ranges over the URIs selected from $\mathcal{C}_X$.

If p is a data property, 3XL handles this similarly to the $(*, p, o)$ case described above with the exception that no restrictions are made for the object (i.e., the parts concerning $\nu(p, o)$ are not included in the SQL) and the values in the column representing p are also selected. Again special care has to be taken if p is an object property. It is then needed to join the class table or multiproperty table holding p values to the class table for the range R of p. Further, the column $\mathcal{C}_R$.`uri` should be selected instead of the column representing p (this is similar to the already described $(s, p, *)$ case). For each row (U, O) in the SQL query's result, a triple (U, p, O) is included in 3XL's result set.

Case $(*, *, o)$ In this case, all triples with the given o as object should be returned. Consider that o could be the name of a class in which case type information must be returned (note that we can ignore the possibility that o is, e.g., `owl:ObjectProperty` as we have assumed that there are no queries for schematic data given in O_S). We handle this part as in the $(*, p, o)$ case (with p = `rdf:type`). But o could also be any other kind of value that some property defined in O_S takes for some instance. To detect if o is another instance, we use the query `SELECT ID FROM map WHERE URI =` o. If the result is empty, we execute the query q =`SELECT * FROM` $\mathcal{C}_X$ `WHERE dp`$_1$ `= o OR ... OR dp`$_n$ `= o` for each class X (the `dp`$_j$'s are the columns holding X's data properties). If the result of the query towards the `map` table, on the other hand, found an ID i, we append `OR op =` i for each column `op` representing an object property in $\mathcal{C}_X$.

Case $(*, *, *)$ In this case, all triples in M should be returned. This can also be done by reusing some of the previously described cases. More concretely the result set for this query consists of a union of all type information triples and the union of all result sets for the queries $(*, p, *)$ where p is a property defined in O_S. Formally, the result set is given by the following where $\Omega(a, b, c)$ denotes the result set for the query (a, b, c) and P is the set of properties defined in O_S: $\Omega(*, \texttt{rdf:type}, *) \cup \left(\bigcup_{p \in P} \Omega(*, p, *)\right)$. In other words, this is handled similarly to how the $(*, *, o)$ case is handled.

2.2.4.2 Composite Queries

Composite queries are composed of a conjunction of several query triples, each of the form (subject, predicate, object). Unknown *variables* used for *linking* triples together, are specified using a string starting with a question mark, while known *constants* are expressed using their full URIs or in an abbreviated form where prefixes can be replaced with shorter predefined values. There are three different patterns for linked query triples:

The first pattern is query triples ordered in a "chain" in which the object of a triple is the subject of the next triple (see Figure 2.4). The second pattern is query triples in a "star" which share a common subject (see Figure 2.5). The third is the combination of the two others. Each query pattern is characterized by *paths* which connect query triples together. A node in the paths is a subject or an object, and an edge connecting two nodes is a predicate. When the query engine constructs SQL statements needed to answer a composite query, a table join is produced for two adjacent nodes if the property linking them is an object property (if the property is a multiproperty, the multiproperty table is also included in the join). If the property is a data property, it can be processed similarly to the ways we have discussed above for the point-wise queries whose predicates are known.

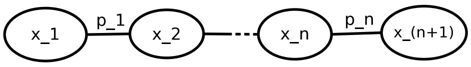


Figure 2.4: "Chain" pattern

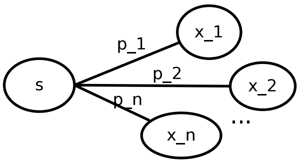


Figure 2.5: "Star" pattern

After the SQL statement is generated, it is directly issued to the underlying DMBS. The advantage of this approach is that, by exploiting the DBMS, we can take advantage of its sophisticated query evaluation and optimization mechanisms for free. Note that there are table joins between different class tables and between class tables and multiproperty tables. As all class tables and multiproperty tables are indexed – typically, there are indices on all the ID columns and loose foreign key columns – the joins are not expensive.

In the following, we give an example to illustrate how a composite query is converted into SQL by the 3XL query engine. The query is used to find all HTML documents with the keyword Java and their corresponding versions.

Example 6 (Composite query) *Consider the composite query (?x, rdf:type, HTMLDocument) (?y, rdf:type, HTMLVersion) (?x, keyword, Java) (?x, usedVersion, ?y) in which* keyword *is a multiproperty of* owl:DataProperty *type, and* usedVersion *is of* owl:-

ObjectProperty type. When the multiproperty is implemented as an array, the composite query gets translated into:
SELECT $C_{\mathit{HTMLDocument}}$*.uri AS x,* $C_{\mathit{HTMLVersion}}$*.uri AS y FROM* $C_{\mathit{HTMLDocument}}$*,* $C_{\mathit{HTMLVersion}}$ *WHERE* $C_{\mathit{HTMLDocument}}$*.usedVersion =* $C_{\mathit{HTMLVersion}}$*.ID AND 'Java' = ANY(*$C_{\mathit{HTMLDocument}}$*.keyword).*

When the multiproperty is implemented as a multiproperty table, the composite query gets translated into:
SELECT $C_{\mathit{HTMLDocument}}$*.uri AS x,* $C_{\mathit{HTMLVersion}}$*.uri AS y FROM* $C_{\mathit{HTMLDocument}}$*,* $C_{\mathit{HTMLVersion}}$*, keywordTable WHERE* $C_{\mathit{HTMLDocument}}$*.usedVersion =* $C_{\mathit{HTMLVersion}}$ *.ID AND* C_{Document}*.id = keywordTable.id AND keywordTable.keyword = 'Java'.*

Figure 2.6 shows the result of this composite query.

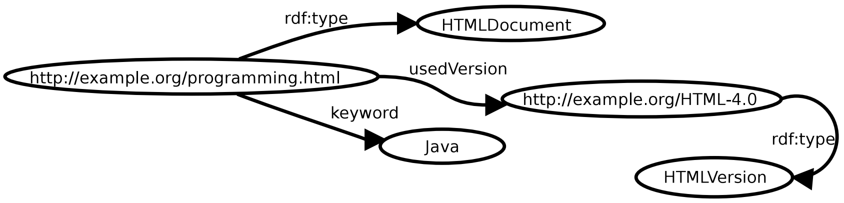

rdf:type
HTMLDocument
http://example.org/programming.html
usedVersion
http://example.org/HTML-4.0
keyword
Java
rdf:type
HTMLVersion

Figure 2.6: The result of the composite query

2.3 Performance Study

2.3.1 Experiment Settings

We first conduct an experimental study to analyze the effectiveness of the various optimizations we made in the implementation. Then, we evaluate the loading and query performance of 3XL in comparison with the two state-of-the-art high-performance triple-stores BigOWLIM [13] and RDF-3X [55]. BigOWLIM is a commercial tool which is implemented in Java as a Storage and Inference Layer (SAIL) for the Sesame RDF database. It supports full RDFS, different OWL variants including most of OWL Lite. RDF-3X is an open source RISC-style engine with streamlined indexing and query processing which provides schema-free RDF data storage and retrieval. Unlike 3XL, the reference systems both use file-based storage. We note that it is not our goal to necessarily be striclty faster than the reference systems, but instead to provide comparable performance in combination with the flexibility of a DBMS-based solution. Finally, we compare the performance of 3XL to that of other DBMS-based triple-stores.

In the experiments, the following two datasets are used:

- **EIAO dataset:** This is a real-world dataset from the European Internet Accessibility Observatory (EIAO) project [82] which developed a tool for performing automatic evaluation of accessibility of web sites. This project serves as the design inspiration for the 3XL triple-store. The EIAO dataset conforms to an OWL Lite ontology [28] which contains 16 classes and 75 properties. Among the properties, 18 are of type `OWL:ObjectProperty`, 57 are of type `OWL:DataProperty`, and 29 are multiproperties.

- **LUBM dataset:** This is a synthetic dataset describing fictitious universities from the Lehigh University Benchmark (LUBM) [34] which is the de-facto industry benchmark for OWL repository scalability. For 3XL, we use an ontology which is based on a subset of the published LUBM ontology, but only uses the 3XL-supported constructs and makes implicit subclass relationships explicit. Our ontology covers 20 classes and 20 properties. Among the properties, 13 are of type `OWL:ObjectProperty`, 7 are of type `OWL:DataProperty`, and 4 are multiproperties. The ontology allows datasets generated by the (unmodified) LUBM generator to be loaded into 3XL. It is available from `people.cs.aau.dk/~xiliu/3xlsystem/`.

We benchmark the performance when loading up to 100 million triples from each dataset (corresponding to 724 universities in the generated LUBM dataset). Before the loading, the original datasets are converted to N-triples format by the Redland RDF parser. All compared systems read input data from the N-triples format. The time spent on parsing is included in the overall loading time. In the query performance study, 14 queries are studied on the LUBM dataset, and 10 queries on the EIAO dataset. Both datasets contain 25 million triples. To reduce caching effects, a query is run 10 times with randomly generated query condition values and the average time is calculated. For example, in the performance study of the query $(s, p, *)$, 10 different values of s are used.

All experiments are conducted on a DELL D630 notebook with a 2.2 GHz Intel(R) Core(TM)2 Duo processor, 3 GB main memory, Ubuntu 10.04 with 32-bit Linux 2.6.32-22 kernel and java-6-sun-1.6.0.20. All the experiment are done under console mode, which all the unnecessary services are disabled including Linux X server. The JVM options "-Xms1024m -Xmx2500m -XX:-UseGCOverheadLimit -XX:+UseParallelGC" are used for both 3XL and BigOWLIM. PostgreSQL 8.3.5 is used as the RDBMS for 3XL with the settings "shared_buffers=512MB, temp_buffers-=128MB, work_mem=56MB, checkpoint_segments=20" and default values for other configuration parameters. BigOWLIM 3.3 is configured with Sesame 2.3.2 as its database, and with the following runtime settings: "owlim:ruleset empty; owlim:entity-index-size 5000000; owlim:cache-memory 200M". This means that no reasoning is done during data loading. The cache memory is calculated by using a configuration spreadsheet included with the BIGOWLIM distribution, and the other settings are as referenced in [13]. For RDF-3X, we follow the setup from [55].

The source code for 3XL, the used datasets and queries, instructions, etc. are available from `people.cs.aau.dk/~xiliu/3xlsystem/`.

2.3.2 Loading Time

We first study the effect of the various optimizations in our implementation. To find the performance contribution of each optimization, we measure the loading time of each by using the LUBM dataset and compare with the non-optimized result. We focus on four aspects: 1) bulk-loading, 2) partial-commit, 3) using BDB to cache the `map` table, and 4) their combination. Figure 2.7 shows the results. First, we see that the loading times grow linearly (with the slight exception of the two middle ones) with increasing dataset size, but at very different rates. When loading without any optimization, data is inserted into the `map` and class tables using SQL INSERTs through JDBC. Before INSERTing, triples with the same subject already existing in the database are removed using DELETEs. Here, it takes a staggering 252 hours to load 100M triples (average speed = 65 triples/sec).Using BDB (next line) has little effect. Switching to buffering triples (by means of value holders) and using JDBC batch INSERTs (next line, in the lower part of the figure) is almost 30 times faster, showing the effectiveness of the 3XL buffering. Adding the partial commit (PC) optimization (line Insert, VH, PC.) is about 13% faster (8.95 versus 10.24 hours) than using normal "full" commit. This is because partial commit takes advantage of data locality to improve the buffer hit rate and thus reduces the number of database accesses when generating value holders (see Section 2.2).

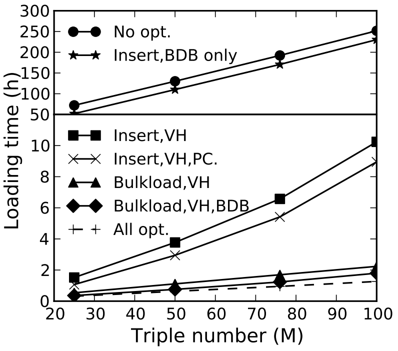


Figure 2.7: Effect of optimizations

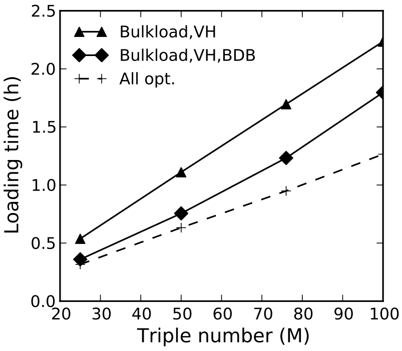


Figure 2.8: Bulkload optimizations

The next big jump in performance comes from using bulkload rather than batch INSERTs (line Bulkload, VH), yielding a further 4-5 fold performance improvement.

Table 2.1: The comparision on loading 100 M triples

	EIAO		LUBM	
	Load time, min	DB size, GB	Load time, min	DB size, GB
3XL-Array	94.3	6.2	67.3	6.1
3XL-MP	90.0	6.2	75.9	6.1
BigOWLIM	86.5	13.0	55.5	13.0
RDF-3X	158.7	5.3	155.7	5.1

The bulkload results are also shown in Figure 2.8 for better readability. Combining bulkload with a Berkeley DB cache for the `map` table further contributes about 50% performance improvement (line Bulkload, VH, BDB). This is due to two reasons: first, the BDB `map` accelerates the identification of the class table during value holder generation, and second, 3XL saves the load time of `map` table data. In our initial implementation, we maintained the `map` table in the database like the class tables: *buffer→CSV→copy to database*. In this scheme, loading 100M triples required 16 commit operations and used 200 minutes to load the `map` table alone. In comparison, the current scheme only needs 10 minutes to load the `map` table, so the optimized scheme of maintaining separate `map` tables for loading and querying is obviously better. Finally, adding partial commit (line All opt.) yields another 15.5% improvement. In summary, the optimization yielded a 202-fold performance improvement, from 255 hours to 1.26 hours, thus demonstrating the effectiveness of the 3XL design and implementation choices.

We have performed experiments to find the optimal configuration parameters, including a) using two different buffer schemes: caching class instances in class-specific buffers versus caching all instances in one common buffer, b) varying the value holder buffer size, c) varying the Berkeley DB buffer size, d) varying the partial commit value, and e) using different cache algorithms including *first in, first out* (FIFO), *least recently used* (LRU) and *least frequently used* (LFU). We found that using one common buffer and the LRU cache algorithm is best for loading performance. The partial commit value depends on the data locality of the dataset, i.e., a lower partial-commit value should be set for a higher data locality, and vice versa. The best sizes for the value holder buffer and BDB cache also depends on the characteristics of the data and the hardware configuration. In addition, using a memory-based file system (tmpfs) [73] to cache the CSV files was tested, but the built-in OS file caching works so well that explicitly using *tmpfs* did not improve performance.

Using the full set of the optimizations, we now compare 3XL with the reference systems by loading 100 M triples from each dataset. The sizes of EIAO and LUBM datasets in N-Triples format are 18.5 GB, and 16.6 GB respectively. However, a characteristic of the two datasets is that EIAO contains many duplicated triples while

LUBM contains only distinct triples. Therefore, based on the above optimization study, we set the partial-commit value to be 0.4 for the EIAO dataset and 0.8 for the LUBM dataset. The value holder buffer size is set to 150,000 and 300,000, respectively. The BDB cache size is 300 MB for both datasets.

The loading results are shown in Table 2.1. For both datasets, 3XL with multiproperties represented in arrays ("3XL-Array") and 3XL with multiproperty tables ("3XL-MP") rank between BigOWLIM and RDF-3X. For the real-world EIAO dataset, 3XL-MP only uses 4% more time than the state-of-the-art triple-store BigOWLIM while 3XL-Array only uses 9% more time. For the synthetic LUBM datasets, 3XL-MP uses 36% more time than BigOWLIM and 3XL-Array uses 21% more time. However, BigOWLIM is using a file-based data store, which is reported [20] to have higher loading performance than relational database-based stores in general. RDF-3X has the slowest load performance, using 83% and 180% more time than BIGOWLIM for the EIAO and LUBM datasets, respectively. For both datasets, the 3XL variants consume less than half of the disk space BigOWLIM requires. The RDF-3X DB size is the smallest, about 15% smaller than the 3XL DB size. Further, 3XL offers a high degree of flexibility in integrating the data with other (non-triple) datasets as 3XL stores data in an intuitive relational database. With this design goal in mind, it is thus very satisfying to achieve a load-performance which is comparable to the state-of-the-art triple-store BigOWLIM.

3XL takes longer time to load the EIAO dataset than the LUBM dataset, although the datasets occupy nearly the same amount of space when loaded. There are two reasons for this. First, as the EIAO ontology has more classes and properties, the loading process needs to operate on more class tables and attributes in the database which takes more time. Second, when processing the duplicated EIAO data, there is a higher possibility that 3XL needs to fetch already inserted data from the database. However, this possibility has been reduced by using a *least recently used* (LRU) cache algorithm and a lower partial-commit value. With the current settings, loading 100 M EIAO triples still involves 207,183 database visits compared to zero when loading the LUBM triples.

In addition to the comparisons with BigOWLIM and RDF-3X, we have also compared with the popular Jena2 systems (filed-based and DBMS-based) [94] by using the LUBM dataset. However, the loading performance of Jena2 systems was slow, and it took 32.6 and 20.4 hours to import 100 M triples to Jena2(DBMS-based) and Jena2(file-based), respectively. In an earlier experiment, we also tried this testing on 3store, which uses a traditional giant *triple table*, but it was not able to scale to load 100 M triples as it did not finish in a reasonable amount of time.

RDF-3X does not exploit an OWL schema for loading the data (as it it is based on RDF, not OWL). Unlike 3XL, RDF-3X does not support repeated loads. If data is loaded into an existing database, the previous data will be overwritten. BigOWLIM

Table 2.2: Load Performance Comparison (Repositories) [13, 20]

RDF data (#Triples)	RATE Triples/second	Configuration	Tool, year
Native 2,023,606	400 db-based	Sun dual UltraSPARC-II 450MHz 1G RAM	RSSDB 2001
Native 279,337	418 db-based	Apple Powerbook G4 1.125GHz 1G RAM	Jena(Mysql) 2004
Native 279,337	4170 file-based	Apple Powerbook G4 1.25GHz 1G RAM	Jena(file) 2004
LUBM 6,890,933	151 db-based	P4 1.80GHz 256M RAM	DLDB 2005
LUBM 1.06 Bill.	12389 db-based	2xXeon 5130 2GHz, 8GB RAM, RAID 4xSATA	OpenLink Virtuoso 2006
LUBM 1.06 Bill.	13100 file-based	AMD 64 2GHz 16G RAM	AllegroGraph 2007
LUBM 100 Mill.	10750 db-based	P4 3.0GHz 2G RAM	Oracle bulk-load scheme 2008
Synthetic RDF 235 Mill.	3840 file-based	AMD Opteron 1GHz, 64-bit JDK, no other info.	KOWARI 2006
LUBM 70 Mill.	6481 file-based	P4 2.8GHz, 1GB, Xmx800, JDK 1.5	Sesame's Native Store 2008
Uniprot 262 Mill.	758 db-based	no info.	RDF Gateway

requires a known schema when doing reasoning during the load. However, no schema is used in this experiment as no reasoning is done. 3XL, on the other hand, exploits the schema for creating a specialized and intuitive database schema. It is, however, still possible to load data that is not described by the OWL schema by means of 3XL's `overflow` table, so 3XL supports the standard open world assumption.

2.3.3 Comparison with Other Bulk-loading Systems

In order to compare with DBMS-based triplestores, we refer to a bulk-loading study published by [13, 20]. Table 2.2 (reproduced from [13, 20]) summarizes the bulk-loading speed rates of a number of tools.

It is of course difficult to compare the results exactly as the datasets as well as the hardware configurations used vary significantly. However, certain things can be deduced. First of all, all the most recent results (2005 and onwards) are based on the *LUBM benchmark* which is also used for 3XL. There are both file-based and DBMS-based (called "db-based") triple-stores in the table. As far as the performance of db-based triple-stores is concerned, the Virtuoso triple-store and the Oracle bulk-loading scheme (described in detail in [20]) have the best performance, with loading speeds of 12,692 and 10,750 triples/second, respectively. However, the Virtuoso scheme is run on much more powerful hardware, meaning that the Oracle scheme is in fact the fastest of the two. The file-based AllegroGraph stores 13,100 triples/second. Virtuoso's implementation has employed parallel loading techniques and storage optimization like bitmap indexes, etc., while the Oracle scheme has used the high-performance commercial DBMS Oracle and made use of its bulk-loading utility *SQL*Loader*. The paper [20] thus establishes the Oracle scheme as the leading DBMS-based bulk-loading scheme. It is thus natural to compare it with 3XL. However, the Oracle license explicitly disallows us to publish performance figures without the consent of Oracle, meaning that open and transparent comparisons are impossible.

We can see that the hardware used for the two setups is almost identical: our CPU is slower, but we have a little more main memory. However, on this almost identical hardware, we can see that 3XL is significantly faster than the Oracle scheme. When loading the LUBM data, 3XL-Array and 3XL-MP handle 24,765 triples/second and 21,959 triples/second, respectively. The difference is so profound that we think we can safely claim that 3XL outperforms the Oracle scheme (which handles 10,750 triples/sec) for bulk-loading.

2.3.4 Query Response Time

We conduct the query testing on the EIAO dataset using 10 queries (Q1–Q10), and on the LUBM dataset using its standard 14 queries[3] (LQ1–LQ14). Each dataset contains 25 M triples. The queries are expressed in the form *(subject, predicate, object)* (with possible "*"-values) for 3XL, and in SPARQL [65] for BigOWLIM and RDF-3X[4]. For example, a point-wise query $(s, p, *)$ with a given subject s and predicate p, can be converted into SPARQL: `select` $?o$ `where` $\{\langle s\rangle\ \langle p\rangle\ ?o$ `.`$\}$.

In 3XL, we have a specialized database schema for classes, properties and different kinds of restrictions. It is thus interesting to study the query performance for: a) different properties, i.e., `owl:ObjectProperty` and `owl:DataProperty`, b) storing multiproperties in arrays versus in tables, and c) the difference between 3XL and the reference systems. We study these by doing the queries on the EIAO dataset, and present the results in Table 2.3. The queries Q1–Q4 are all of the form $(s, p, *)$, but with different types of the predicates, namely single-valued object property, single-valued data property, multi-valued object property, and multi-valued data property, respectively. Overall, in 3XL the queries on data properties are faster than queries on object properties, e.g., Q2 vs. Q1 and queries on single-valued properties

[3] The LUBM queries are available from `swat.cse.lehigh.edu/projects/lubm/query.htm`

[4] The SPARQL queries are available from `people.cs.aau.dk/~xiliu/3xlsystem`

Table 2.3: Query response time for the EIAO dataset with 25 M triples (ms)

	Q1$(s,p_1,*)$	Q2$(s,p_2,*)$	Q3$(s,p_3,*)$	Q4$(s,p_4,*)$	Q5$(*,p,*)$
3XL-Array	53	48	216	48	16943
3XL-MP	69	26	78	58	17255
BigOWLIM	87	85	321	85	2002
RDF-3X	59	81	591	91	99369

	Q6$(*,p,o)$	Q7$(s,*,*)$	Q8$(s,*,o)$	Q9$(*,*,o)$	Q10(Comp.)
3XL-Array	8984	23	38	6333	38243
3XL-MP	8934	68	75	3227	29022
BigOWLIM	898	121	146	104	7398
RDF-3X	33466	46	38	225	139951

are faster than on multiproperties, e.g., Q1 vs. Q3 and Q2 vs. Q4. No significant difference is observed between 3XL-Array and 3XL-MP except for Q3. We use the queries Q5–Q9 to study the performance of point-wise queries different from the $(s,p,*)$ form. Q10 is used to study the performance of a composite query. As shown in the results, 3XL outperforms the two reference systems for Q1–Q4 and Q7–Q8 where the subjects s are given. This is mainly due to 3XL's "intelligent partitioning", where, given a particular subject s, 3XL can very quickly locate the class table holding the relevant data. For the composite query Q10, and the point-wise queries Q5–Q6 with wildcard "*" in the subject but with a given predicate p, all the systems take a longer time than for the other queries as more results are returned. For these three queries, 3XL ranks in the middle. In the case of Q9 with only a given object o and with the subject and the predicate using "*" to match anything, 3XL takes a longer time as this query has to traverse all predicates p_i. Here, we note that 3XL is in fact specifically *designed* to be efficient for queries *with* a subject.

We now proceed to make an evaluation by using the LUBM dataset and its queries (LQ1–LQ14). These queries are all composite queries which are more expressive and complex than the point-wise queries we discussed above. Table 2.4 describes the test results of 3XL and the reference systems. Overall, BigOWLIM has the highest completeness and supports all 14 queries, while RDF-3X only supports 4 queries which is due to its lack of OWL inference. 3XL supports 10 of the LUBM queries. Because of its use of an inheritance database schema, 3XL has some semantic abilities and can reason on the instances of a class and its subclasses. When querying on a class table, all of its subclass tables are queried as well. Therefore, 3XL does support queries that, e.q., query a class and its subclasses. With regard to the query performance, 3XL-MP, in general, outperforms 3XL-Array since the multiproperty table is indexed. Neither of the systems is able to outperform all other systems for all queries.

Table 2.4: Query response time for the LUBM dataset with 25 M triples (ms)

	LQ1	LQ2	LQ3	LQ4	LQ5	LQ6	LQ7
3XL-Array	531	627	1376	127	N/A	(11526)	2623
3XL-MP	139	576	661	114	N/A	(11459)	126
BigOWLIM	185	75219	130	183	90	29830	179
RDF-3X	35	36096	74	N/A	N/A	N/A	N/A
	LQ8	**LQ9**	**LQ10**	**LQ11**	**LQ12**	**LQ13**	**LQ14**
3XL-Array	1675	56835	(1390)	N/A	N/A	N/A	8764
3XL-MP	1612	15083	(137)	N/A	N/A	N/A	8745
BigOWLIM	676	100037	2	154	674	11796	46625
RDF-3X	N/A	N/A	N/A	N/A	N/A	N/A	17006

For LQ1, which selects instances of a given class which reference a certain instance of another class, RDF-3X has the fastest query response time while the times used by BigOWLIM and 3XL-MP are quite similar. For LQ2, which selects instances of 3 classes with a triangular pattern of relationships between the involved instances, both the 3XL variants are more than 2 orders of magnitude faster than the two reference systems. LQ3 is similar to LQ1 in both query characteristics and results. LQ4 selects instances and 3 property values from a class (with many subclasses) based on an object property linking to another class and is highly selective. 3XL-MP takes the least time for this query closely followed by 3XL-Array. RDF-3X does not support this query. LQ5 depends on `rdfs:subPropertyOf` which is not supported by 3XL. LQ6 selects all instances from a given class and its subclasses (an implicit subclass relationship was made explicit in the modified LUBM ontology and the timings for 3XL are therefore shown in parentheses). All the systems take a longer time on this query, but the 3XL variants both outperform BigOWLIM. LQ7 involves more classes and properties, and is more selective than LQ6. 3XL-MP is the fastest followed by BigOWLIM. LQ8 is based on LQ7 but adds one more property to increase the query complexity. The 3XL variants almost have equal performance which is lower than BigOWLIM's. LQ9 involves a triangular pattern of relationships between 3 classes. This query takes much longer time than all the other queries in all systems, but 3XL-MP has considerably better performance. LQ10 selects instances from a class but depends on the same implicit subclass relationship as does LQ6. The numbers shown in parentheses show the time spent by 3XL when the implicit relationship is given explicitly. LQ11, LQ12, and LQ13 depend on inference not supported by 3XL or RDF-3X. The last query LQ14 selects all instances of a given class (without subclasses). This query is similar to LQ6, but uses a subclass of the class used by LQ6. The 3XL variants are both faster than the two reference systems.

In summary of the performance results on the EIAO dataset, 3XL-Array and BigOWLIM both show the best performance in 4 out of the 10 queries, 3XL-MP shows the best performance in 2 queries, and RDF-3X shows the best performance (actually a tie) for a single query. For the LUBM dataset, 3XL-MP shows the best performance for 6 out of the 14 queries while BigOWLIM has the best performance for 7 queries and RDF-3X has the best performance for the remaining query.

In particular when querying triples with a shared subject, 3XL shows very good performance. 3XL has inference capabilities on instances with an inheritance relationship by means of its database schema. RDF-3X cannot answer many of the considered LUBM queries as it has no schema support and cannot do any inference.

In summary, the performance of 3XL is comparable to that of the state-of-the-art file-based triple-stores. This holds both for loading and query times. The performance of 3XL exceeds that of other DBMS-based triplestores. 3XL is designed to use a database schema which is flexible and easy to use, and it is thus very satisfying that the solution achieves a very good performance, while offering the flexibility of the DBMS-based triple-store.

2.4 Related Work

Different RDF and OWL stores have been described before. In this section we describe the most relevant ones. Note that terminology is used with different meanings in different solutions. For example, "class table" is not meaning the same in *RDFSuite* described below and in 3XL.

An early example of an RDF store can be found in *RDFSuite* [4,5]. In the part of the work focusing on storing RDF data, two different representations are considered: *GenRepr* which is a generic representation that uses the same database schema for all RDF schemas and *SpecRepr* which creates a specialized database schema for each RDF schema. It is found that the specialized representation performs better than the generic representation.

In the generic representation, two tables are used. One for resources and one for triples. In a specialized representation, RDFSuite represents the core RDFS model by means of four tables. Further, a specialized representation has a so-called *class table* for each class defined in the RDFS. In contrast to the class tables used by 3XL, RDFSuite's class tables only store the URIs of individuals belonging to the represented class. Both RDFSuite and 3XL use the table inheritance features of PostgreSQL for class tables. RDFSuite's specialized representation also has a so-called *property table* for each property. This is different from 3XL's approach where multiproperty tables only are used if the cardinality for the represented property is greater than 1. In RDFSuite, property tables store URIs for the source and target of each represented property value. Alexaki et al. [4] also suggest (but do not implement) a representation where single-valued properties with literal types as ranges are represented as attributes in the relevant class tables. This is similar to the approach taken by 3XL. In 3XL this is taken a step further and also done for attributes with object values.

In Broekstra et al.'s solution for storing RDF and RDFS, *Sesame* [16], different schemas can be used. Sesame is implemented such that code for data handling is isolated in a so-called *Storage and Inference Layer* (SAIL). It is then possible to plug-in new SAILs. A generic SAIL for SQL92 compatible DBMSes only uses a single table with columns for the subjects, predicates and objects. In a SAIL for PostgreSQL, the schema is inspired by the schema for RDFSuite and is dynamically generated. Again, a table is created for each class to represent. Such a table has one column for the URI. A table created for a class inherits from the tables created for the parents of the class. Likewise, a table is created for each property. Such a table for a property inherits from the tables that represent the parents of the property if it is a subproperty. This SAIL is reported [16] to have a good query performance but disappointing insert performance when tables are created.

In Wilkinson et al.'s [94] tool for RDF storage, *Jena2*, all statements can be stored in a single table. In the statement table, both URIs and literal values are stored directly. Further, Jena2 allows so-called *property tables* that store pairs of subjects and values. It is possible to cluster multiple properties that have maximum cardinality 1 together in one property table such that a given row in the table stores many property values for a single subject. These can be compared to 3XL's class tables. An important difference is, however, 3XL's use of table inheritance to reflect the class hierarchy.

Harris and Gibbins [37] suggest a schema with fixed tables for their RDF triple-store, 3store. One table with columns for subject, predicate and object holds all triples. To normalize the schema, there are also tables for representing models, resources, and literals. Each of these has two columns: one for holding an integer hash value and one for holding a text string. The triple table then references the integer values in these three tables. This approach where all triples are stored in one table is different from the approach taken by 3XL where the data to store is held in many different tables.

Pan and Heflin [59] suggest the tool *DLDB*. The schema for DLDB's underlying database is similar to RDFSuite's. DLDB also defines views over classes. A class's view contains data from the class's table as well as data from the views of any subclasses. Instead of views, 3XL uses table-inheritance. A DLDB version for OWL also exists.

Neumann and Weikum [55,56] suggest a scalable and general solution for storing and querying RDF triples. The system, called *RDF-3X*, does not use a DBMS, but a specialized storage system which applies intensive indexing to enable fast querying. A major difference between RDF-3X and 3XL is that 3XL uses (a subset of) OWL

Lite and supports OWL classes, object and data properties, etc. and thus, unlike RDF-3X, can answer OWL queries as most of those in the LUBM benchmark.

Abadi et al. [1] propose to to use a two-column table `property(subject, object)` for each unique property in an RDF data set. This is implemented in both a *column-store* which stores data by columns and in a more traditional *row-store* which stores data by rows. The proposed schema is reported to be about three times faster than a traditional triple-store in a row-store while it is around 30 times faster in a column-store. In a later evaluation paper, Sidirourgos et al. [71], however, find that in a row-store, the simple triple-store performs as well as the two-column approach if the right indexes are in place. They also find that a column-store provides good performance but that scalability becomes a problem when there are many properties (leading to many tables). 3XL is designed to be fast for queries where the subject and/or predicate is known and where many/all properties should be retrieved. Further, it exploits the object-relational capabilities of the row-store PostgreSQL. We therefore believe that the best choice is to group the single-valued properties of a class together in a class table and allow multi-valued properties in special property tables or arrays in the class table.

Zhou et al. [95] implement the *Minerva* OWL semantic repository integrated with a DL reasoner and a rule inference engine. The imported data is inferred based on a set of rules, and the results are materialized to a DB2 database which has an inference-based schema containing atomic tables, TBox axiom tables, ABox fact tables and class constructor tables.

IBM SHER [26] is a reasoner that allows for efficient retrieval of ABoxes stored in databases. It achieves the efficiency by grouping the instances of a same class into a dramatically simplified summary ABox, and doing queries upon this ABox.

Storage of RDF data has also found its way into commercial database products. *Oracle 10g* and *11g* manage storage of RDF in a central, fixed schema [58]. This schema has a number of tables, including one that has an entry for each unique part of all the triples (i.e., up to three entries are made for one triple) and a table with one entry for each triple to link between the parts in the mentioned table. In a recent paper [20], it is described how Oracle supports efficient *bulk loading* by extensive use of SQL and a hash-based scheme for mapping between values and IDs.

Unlike Minerva's inference-based schema, SHER's summary ABox, and Oracle's use of a variant generic schema representation, 3XL uses PostgreSQL's object-oriented fuctionalities to support its specialized data-dependent schema.

Other repositories designed for OWL also exist. *DBOWL* [53] creates a specialized schema based on the ontology like 3XL does. There is a single table with three attributes to store triples. For each class and property in the OWL ontology, a view is created. 3XL differs from this, as data is partitioned over several physical tables that may have many attributes. *OWLIM* [44] is another solution. It is implemented as a

SAIL for Sesame. Two versions of OWLIM exist: 1) The free SwiftOWLIM which is the fastest but performs querying and reasoning in-memory and 2) the commercial BigOWLIM which is file-based and scales to billions of triples. This is different from 3XL that has its data in an underlying PostgreSQL and exploits the results of decades of research and development in the database community such as atomicity, concurrency control, and abstraction.

2.5 Conclusion and Future Work

In this chapter, we present the 3XL triple-store. Unlike most current triple-stores, 3XL is specifically designed to support easy integration with non-RDF data and at the same time support efficient data management operations (load and retrieval) on very large OWL Lite triple-stores. 3XL's approach has a number of notable characteristics. First, 3XL is DBMS-based and uses a *specialized data-dependent schema* derived from an OWL Lite ontology. In other words, 3XL performs an "intelligent partitioning" of the data which is efficiently used by the system when answering triple queries and at the same time intuitive to use when the user queries the data directly in SQL. Second, 3XL uses advanced object-relational features of the underlying ORDBMS (in this case PostgreSQL), such as table inheritance and arrays as "in-lined" attribute values. The table inheritance represents subclass relationships in a natural way to a user. Third, 3XL is designed to be efficient for bulk insertions. It makes extensive use of a number of bulk loading techniques that speed up bulk operations significantly and is designed to use the available main memory very efficiently, using specialized caching schemes for triples and the `map` table. Fourth, 3XL supports very efficient bulk retrieval for point-wise queries where the subject and/or the predicate is known, as we have found such queries to be the most important for most bulk data management applications. 3XL also supports efficient retrieval for composite queries. 3XL is motivated by our own experiences from a project using very large amounts of triples. Extensive experiments based on the real-world EIAO dataset and the industry standard LUBM benchmark show that 3XL has loading and query performance comparable to the best file-based solutions, and outperforms other DBMS-based solutions. At the same time, 3XL provides flexibility as it is DBMS-based and uses a specialized and intuitive schema to represent the data. 3XL thus bridges the gap between efficient representations and flexible and intuitive representations of the data. 3XL thus places itself in a unique spot in the design space for triple-stores.

The overall lessons learnt can be summarized as follows: 1) Using a specialized schema generated from an OWL ontology is very effective. With this schema, it is fast to find the relevant data which is intelligently partitioned into class tables (and possibly also multiproperty tables). This results in very good query performance.

2) An ORDBMS is a very strong foundation for building such a specialized schema for an OWL ontology. It provides the needed functionality (i.e., table inheritance) to represent class relationships and efficient storage of multi-valued variables in arrays. It also provides a query optimizer, index support, etc. for free. Finally, it provides the user flexibility as it is easy to combine the data with non-RDF data. 3) The right choice of caching mechanisms is very important for the performance. In particular, the often used `map` table should be cached outside the DBMS. For a `map` table too big to fit in memory, an external caching system based on BerkeleyDB is better than using the DBMS. 4) When loading OWL data, using bulk-loading in clever ways has a huge effect. In particular, instead of bulk-loading all available data, only the oldest parts should be loaded keeping the freshest data in memory.

There are a number of interesting directions for future work. First of all, optimization will be continued, focusing on performance improvement of data transferred from memory to database, and supporting queries of form $(*, *, o)$ better. Further, 3XL will be extended to support more of the OWL features, in the first case all of OWL Lite. Finally, 3XL will be integrated with a reasoner running on top of 3XL to allow more reasoning than the class-subclass reasoning on instances currently supported.

Chapter 3

All-RiTE: Right-Time ETL for Live DW Data

Data warehousing traditionally extracts, transforms, and loads (ETL) the data from different source systems into a central data warehouse (DW) in at regular interval, e.g., daily. Data warehousing technologies face the challenge on how to deal with the so-called *live DW data*, such as *accumulating facts* and *early-arriving facts*, which eventually will be updated but also queried in online fashion. For this type of data, traditional SQL INSERTs are used, then typically followed by the updates and possible deletions. This is, however, not efficient to modify the data in the DW. This chapter presents the ETL middleware system *All-RiTE* which enables efficient processing of live DW data with support for SELECTs, INSERTs, UPDATEs and DELETEs. All-RiTE makes use of a novel main memory-based intermediate data store between data producer and the DW to accumulate the live DW data, and does on-the-fly data modifications when the data is materialized to the DW or queried. A number of *policies* are proposed to control *when* to move data from source systems towards the DW. The data in the intermediate data store can be read by data consumers with specified time accuracies. Our experimental studies show that All-RiTE provides a new "sweet spot" combining the best of standard JDBC and bulk load: data availability as with INSERTs, loading speeds even faster than bulk load for short transactions, and very efficient UPDATEs, DELETEs and SELECTs of live DW data.

3.1 Introduction

In data warehousing, the data from various source systems is extracted, transformed, and loaded into the DW. The data is typically refreshed at a regular time interval, e.g., daily. In recent years, there has been an increasing demand for fresher data, i.e., a shorter refreshment period. In some cases, it is even required to have DWs refreshed with the data within minutes or seconds after the triggering event in the real world (e.g., the placement of an order). This is often referred to as "real-time" or "near real-time" data warehousing [10, 17]. Besides, a recent and more sophisticated requirement is to make the data available in the DW *when* the users need it, but not necessarily before. This is referred to as "right-time" data warehousing [85]. Users can specify a certain freshness for the data they see. For example, a user specifies to read the data with a freshness of at least five minutes. Then, the data committed five minutes ago should be available in the DW, but not necessary after that, e.g., one minute ago. The short intervals lead to relatively few insertions in each refreshment compared to, e.g., daily refreshments.

As shorter refresh intervals are used, it is also more likely that the just-inserted data needs to be updated. That is, the data will be updated if some information is not available when the data is inserted. In this chapter, we use the term *live DW data* for the DW data that can be updated and available for reads in online fashion before loaded into the DW. A typical scenario illustrating this involves accumulating facts [42] which are updated when events in the modeled world happen. A fact is, e.g., inserted when (or shortly after) an order is placed. This fact gets updated in the DW whenever the order becomes accepted, packed, sent, and paid. Another example involving live DW data is early-arriving facts [42] where some of the referenced dimension values are unknown at load time. The inserted data then gets updated later when more information is available. For both of the scenarios, it is not efficient to insert data into the DW and then update it shortly after. Better performance can be achieved if the data is updated before it physically gets into the DW. We believe this will often be possible assuming that most updates to live DW data happen within in a short time after the data was created, e.g., within minutes or hours, while changes are rarely made for the data created a long time ago. To support such a scenario efficiently has traditionally involved a complex setup of the ETL program with much hand-coding, though.

Thus, a solution that enables efficient near real-time and/or right-time ETL for live DW data is thus strongly needed. This chapter presents exactly such a solution: The middleware *All-RiTE*. This chapter extends the previous solution, RiTE [85], which only supported insertions, but not updates and deletions. The current solution is named All-RiTE because it supports *all* the traditional operations and provides *Ri*ght *T*ime *E*TL. With All-RiTE, the user uses JDBC INSERTs to add data to the

DW and gets the usual immediate availability, but with much higher efficiency. For long transactions, the performance of All-RiTE is nearly as good as when using bulk loading. For short transactions (the typical case), the performance is much better. In addition, UPDATEs and DELETEs on live DW data are much more efficient when All-RiTE is used. From the user's point of view, it is very easy to start using All-RiTE: only the JDBC drivers have to be changed, and the solution is thus transparent for the involved applications.

The rest of the chapter is structured as follows. In Section 3.2, we give an overview of All-RiTE and its components. In Section 3.3, we consider the supported data processing operations, including INSERT, UPDATE, DELETE, SELECT and "materialization". In Section 3.4, we present optimizations of the All-RiTE implementation. In Section 3.5, we study the performance of All-RiTE. In Section 3.6, we present related work. Finally, in Section 3.7, we conclude the chapter with a summary and pointers to future work.

3.2 The All-RiTE System

In this section, we give an overview of All-RiTE's architecture which is shown in Figure 3.1. The *producer* is an ETL program that loads data from the operational source systems into the DW. The producer is allowed to read, insert, update, and delete DW data. A *consumer* is a DW client that only queries the DW. All-RiTE supports many producers loading different tables in parallel and many consumers reading data in parallel, but for simplicity we only show one producer and one consumer in Figure 3.1.

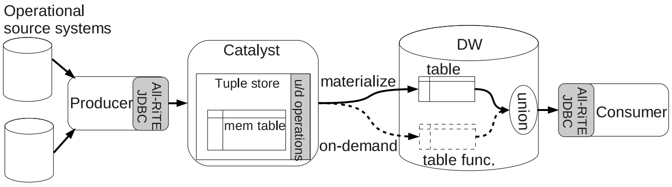


Figure 3.1: System architecture using All-RiTE

The arrows illustrate the data flows. The producer and consumer use All-RiTE through the *All-RiTE JDBC drivers* (shown as grey boxes in Figure 3.1) which extend the standard JDBC interface [79]. Thus, only few and simple changes (explained later) are needed to start using All-RiTE, otherwise the use is transparent. When the consumer inserts data, the data moves from the producer to the consumer through a special component called the *catalyst*. We borrow the chemistry term *catalyst* to represent that the processing of the data that goes through it can get a significant speed-up. Producers, consumers, and the catalyst can be deployed on the same machine or on different machines. In the following, we describe the purpose of the different drivers and the catalyst.

All-RiTE JDBC Producer Driver All-RiTE's ProducerConnection is used by producers. Its interface is very similar to the standard JDBC Connection interface but adds two methods (to be described later). From the programmer's point of view, it can thus be used as a normal JDBC Connection. When data is inserted, the producer connection will, however, keep the new data locally at least until a commit operation is invoked. It may even keep the data locally after the commit operation. When the data should be made available in the DW, a bulk of data is moved towards the DW which is more efficient than moving a single row at the time. The producer connection does not insert the locally held data directly into the DW. Instead, the data is given to the catalyst (described below) and we say the data is *flushed*. When the data is physically in the catalyst, it can be seen from the DW. The producer may also decide that the data should be stored physically in the DW (through a customized commit method discussed later). We then say the data is *materialized*.

The first method ProducerConnection adds to the JDBC Connection interface (or more precisely overloads) is `commit(boolean materializeFlag)` which takes an argument that decides if the data to be committed should be materialized *immediately* or not (the default). The second added method is `setFlushPolicy-(Policy policy)` which is used for setting a *policy* that decides *when* to flush data from the producer local storage to the catalyst. Different kinds of policies can be applied. With *instant flushing* (the default), data is flushed instantly when a commit operation takes place. With *lazy flushing*, the committed data is not flushed until it becomes necessary (i.e., when a consumer requests to read it or the policy decides that it is time to do it). Apart from these methods, ProducerConnection looks like a normal JDBC Connection. It can, however, handle insertions, deletions, and updates specially to obtain high efficiency.

All-RiTE Catalyst The catalyst is the component that temporarily holds data. The held data can be seen from the DW and will eventually be materialized, i.e., physically inserted into the DW and removed from the catalyst. When a loading starts, the catalyst creates a *tuple store* for each table. The tuple store contains a memory table for keeping the data (shown as mem. table), the objects that represents the updates and deletions (shown as "u/d operations"), and indexes for quickly retrieving the data from the memory table (we will discuss more details later). Recall that we assume that most changes of live DW data happen shortly after the data is inserted. All-RiTE does on-the-fly updates and deletions by applying the u/d operations in the catalyst when the data is queried or materialized to the DW.

All-RiTE JDBC Consumer Driver All-RiTE's ConsumerConnection is used by consumers. Its interface is identical to that of JDBC's standard Connection. When prepared statements are created, ConsumerConnection creates specialized prepared statements. These will automatically *register* the query with the catalyst to ensure consistency (details to be given later). Further, they offer the method `ensureAccuracy(long freshness)` which can be used to define the needed freshness of the queried data. If this method is not called, a default accuracy of 0 is assumed, meaning that all committed data should be seen. If, e.g., a freshness of five minutes is specified, data committed five minutes ago (or before) will be seen by the query, but not necessarily data committed three minutes ago. The consumer driver does not read data directly from the catalyst, but read through the table function (shown as table func.), which is a stored procedure that reads data from the catalyst and returns it as rows when invoked. To hide the complexity to users, a view is created, by which the data from the catalyst is merged with the data from a DW table through the SQL UNION operator. Thus the end user can use this view as any other relation and not pay attention to whether the data is physically stored in the catalyst or in the DW. It is automatically ensured that a query only sees a given row once, even if it gets materialized while the query is running.

3.3 Optimizing Operations

In this section, we describe the details of the operations that All-RiTE supports, namely INSERT, UPDATE, DELETE and SELECT. To start using All-RiTE from an existing Java program, only the lines creating the database connection need to be changed. The connection is used to create prepared statements that enable All-RiTE's functionalities.

3.3.1 Running Example

To illustrate the proposed concepts, we use a running example in the rest of the chapter. We consider insertion of rows into the DW table $X(A, B)$. In the example, we consider five rows, $r_i = (i, i)$ for $i = 0, \ldots, 4$, i.e., the rows $(0, 0)$, $(1, 1)$, $(2, 2)$, $(3, 3)$, $(4, 4)$.

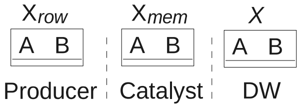


Figure 3.2: The initial state when a load starts

When a load of X is started, the producer driver creates a local buffer called a *row holder* and a memory table in the catalyst for data (to be) inserted into X. These are denoted by X_{row} and X_{mem}, respectively. Figure 3.2 shows the initial states of X_{row},

X_{mem}, and X. A double line in the bottom of a table shows that there are no rows in the table. During the loading, the rows read from the data source are temporarily kept in X_{row}, then flushed to X_{mem}, and finally loaded into X in the DW.

3.3.2 Insert

In the producer, when the prepared statement for INSERT is executed, a row is created from the scalar values set in the prepared statement (currently only explicitly given scalar values are supported). The producer driver assigns a unique identifier, called the *row ID* (the surrogate key), to the row and inserts it into the row holder. The rows in the row holder are *flushed* to the catalyst based on a *policy*. The policy can, e.g., do an *instant flush* or a *lazy flush*.

Instant Flush When the producer does an instant flush, the rows in the row holder are flushed to the catalyst immediately when the producer issues a commit. This is done through the method `commit()` (as in standard JDBC) or the method `commit(boolean materializeFlag)` (added by All-RiTE) to which a materialization flag is given as the argument. If the argument to the latter is `true`, the flushed rows (and those that were already in the catalyst) are materialized to the DW directly after the flush. If the argument is `false` or `commit()` is used, the rows are flushed to the catalyst, but not materialized. The following example illustrates instant flush:

Example 7 (Instant Flush) *Consider inserting the rows of the running example into the table X. The producer has (by means of the prepared statement `INSERT INTO X VALUES (...)`) inserted the three rows, r_0, r_1, and r_2. The producer connection has not sent these two insertions to the DW, but instead holds the rows locally in X_{row}. At time t_0 there is thus only data in X_{row} as shown in Figure 3.3. At time t_1, when the producer has done a commit, the rows have been flushed to the catalyst and are now stored in the memory table X_{mem}.*

Later, the producer starts a new transaction and inserts the two rows r_3 and r_4 into X_{row}. At time t_2 there is thus committed data in the catalyst and uncommitted data in the row holder. At time t_3 after the producer has invoked the `commit` method with the materialization flag set to `true`, all the rows from the row holder and *the catalyst have been materialized and are physically stored in the DW table X.*

Lazy Flush When the producer does a lazy flush, the producer connection stores newly committed rows locally in a so-called *archive* instead of flushing them to the catalyst. However, if a consumer wants to query fresh data, the rows in the archive will automatically be flushed to the catalyst. We show an example of lazy flush below.

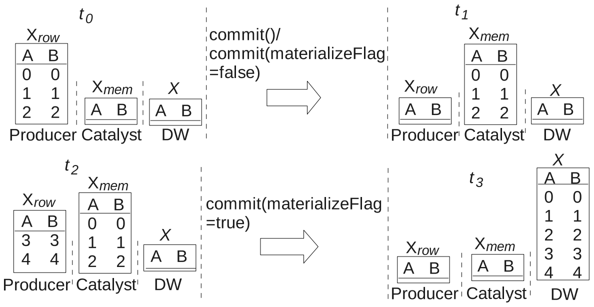


Figure 3.3: Instant flush

Example 8 (Lazy Flush) *At time t_0, the state is as in the previous example, i.e., the producer has inserted the three rows, r_0, r_1, and r_2 which are stored locally in X_{row}. At time t_1, after the producer has done a commit, the three committed rows are stored locally in the archive X_{ar} (and not flushed to the catalyst).*

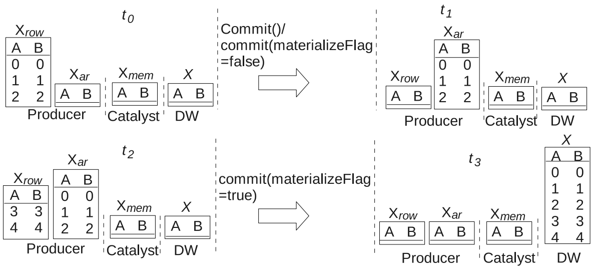


Figure 3.4: Lazy flush

Later, the producer continues to insert the two rows r_3 and r_4. At time t_2, the producer thus holds both committed data (in X_{ar}) and uncommitted data (in X_{row}). At time t_3 after a materialization has taken place by means of `commit (materializeFlag=true)`, all the rows are stored physically in the DW table X.

When a producer wants to materialize rows, the rows in the producer side are first flushed to the catalyst by the producer connection and then physically inserted into the DW by the catalyst. If the rows are not yet materialized to the DW, the consumer can specify the time accuracy of the rows to be read from the producer side and/or the catalyst. Depending on the time accuracy, e.g., with the accuracy set to 0, the consumer can get the latest committed data, i.e., of near real-time availability.

Note that if a consumer in the previous example had needed fresh data at time t_2, (the needed parts of) the committed data would have been flushed to the catalyst. Thus committed, but not flushed, data is made available in the catalyst *on demand.*

This happens when a consumer requests a given accuracy from the catalyst (if no accuracy is explicitly requested, an accuracy of 0 is assumed, meaning that all committed data should be made available to the consumer). If the catalyst does not have data that is fresh enough (judge by using the time index, see Example 9), it will request the producer to flush data fulfilling the requested accuracy. To be able to satisfy such a request, the producer connection has an index mapping commit times to rows in X_{ar}. When it receives a request, it checks the index according to the requested freshness, reads the rows from the archive, and flushes them to the catalyst.

Example 9 (Flush on demand) *Consider that the producer connection stores the committed rows in the archive X_{ar}. Suppose that the rows r_0, r_1, and r_2 are committed at time 1 and the two rows r_3 and r_4 are committed at time 2. The producer connection then maintains two entries in the time index for the rows inserted in the two transactions as shown in Figure 3.5.*

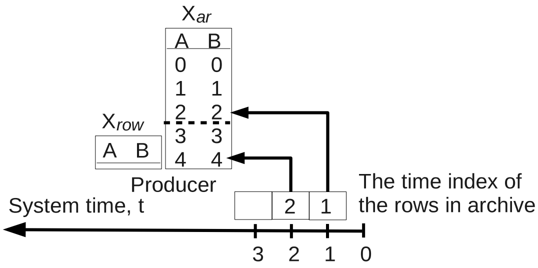


Figure 3.5: Flush rows on demand

Assume that at time 3, the consumer requests to read the data with an accuracy of 2 time units. When the producer driver receives this request, it checks the time index and flushes the three rows r_0, r_1, and r_2 to the catalyst while the other two rows (r_3 and r_4) are left in the archive as their freshness is less than 2 time units.

When the producer flushes rows in the archive, we consider the following two special cases: 1) Suppose that the desired freshness is n time units, the current system time is t, and the commit time of the first row in the archive is t', i.e., the minimal time in the time index. If the requested time satisfies $(t - n) < t'$, no rows will be flushed to the catalyst. 2) If the desired freshness is 0 time units, it means that the consumer wishes to read the most recently committed data. Thus, the producer will flush all the rows in the archive to the catalyst.

Flush Policies As previously stated, a policy decides exactly when to flush data. A policy is simply a function returning a boolean value. This function is called by the producer connection. If policy returns `true`, the producer connection will flush rows from the row holder and row archives to the catalyst immediately; otherwise, it will not flush. The default policy always returns `true` resulting in instant flushing. To use other flush policies, users can set them through the producer connection. The following policies are implemented in All-RiTE:

– *Flush on Demand.* This policy only flushes the rows in the archive on demand. Thus the policy always returns `false` such that a flush only is done after a request from the catalyst.

– *Flush on CPU Workload.* The producer connection can flush rows based on the producer CPU workload. This policy regularly checks the CPU workload. If the CPU load is below a configured value, the policy returns `true` such that the rows are flushed to the catalyst. This policy can make the producer less intrusive on a busy system.

– *Flush at Regular Intervals.* This policy allows the producer driver to flush the rows at regular time intervals, such as every five minutes or every hour. If the time passed since the last flush is longer than the user-specified duration, the policy returns `true` such that a flush is done.

However, users can also implement their own policies easily according to their needs. To create a new policy, only requires implementing an interface (with a boolean value returned). To use a given policy, users can simply enable it by calling `setFlushPolicy(policy)` on the All-RiTE ProducerConnection class.

Add Rows in Tuple Store Recall that when a prepared statement for INSERT is created using the producer connection, All-RiTE creates tuple stores for the DW tables in the catalyst. Each tuple store is corresponding to a single source and a DW table, which contains a single memory table, indices related to the memory table, and other relevant data. The memory table has the same schema as the DW table it is created for. The rows in the memory table are saved in a number of data segments, each of which is a fixed-size main memory buffer. The data segments are backed by disk files such that data amounts larger than the available memory can be stored. A data segment is assigned a unique number, called *segment ID* (denoted by sid), when it is created. Recall that each row is assigned a row ID when it is inserted (we denote the row ID by rid). Both segment IDs and row IDs are indexed to speed up queries. A time index is also created by using the commit times of the rows. We show this in the following example.

Example 10 (Indices) *Consider the five rows in Example 9 which are committed at time 1 and 2. If all the rows are flushed to the catalyst, and suppose (for the sake of the example) that a data segment can only store two rows, there are then three data segments created to hold the rows (shown in the leftmost part of Figure 3.6). In the segment index, each segment ID* sid *maps to a segment. In the data segments, each row is stored with an* rid *assigned to it.*

In All-RiTE, the consumer can read the rows with specified row IDs (as discussed later). The row ID index is then used for finding the data segments that store the

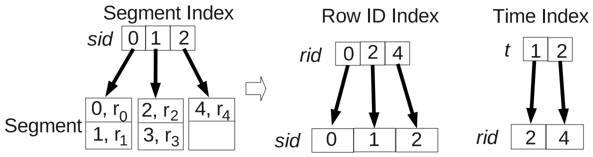


Figure 3.6: Data segments and indices

requested rows. The row ID index is mapping the rid of the first row in a data segment to the sid of the segment (see the middle part in Figure 3.6). The first rid is also the minimal row ID in a data segment, denoted by rid_{min}. The time index (see the rightmost part in Figure 3.6) is mapping commit times to row IDs. The row ID mapped to by a given commit time, is the row ID of the last row in the commit (i.e., the maximal row ID for the transaction).

The row ID index and the time index make it fast to locate rows which then can be read by consumers as explained in Section 3.3.4.

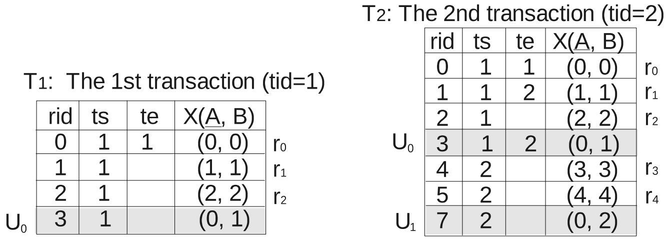
Materialize Data We now describe how the rows in the catalyst are materialized to the DW. As we have discussed earlier, the rows are materialized if the producer commits with the materialization flag set to `true`. In the tuple store, all rows are marked as materialized or not yet materialized (by a boolean flag). If the catalyst receives a materialization instruction from the producer, it first exports all the rows that are not yet materialized into a temporary file and then loads the data in the temporary file into the DW using the DBMS bulk loader.

3.3.3 Update and Delete

In All-RiTE, we conduct the updates and deletions for live DW data in the catalyst after the data for the updates and deletions has been flushed to the catalyst. This section will introduce the process in the following.

In the producer, a prepared statement is created for an UPDATE or DELETE operation. When the prepared statement is executed, the affected rows are not updated/deleted immediately on the producer side. Instead, the producer driver keeps the data about update/delete (“u/d”) operations in a so-called *UD holder*. The producer connection assigns each u/d operation an ID, a unique number taken from the same sequence as row IDs. The ID is used to identify the execution order for every operation. Besides the ID, the scalar values set in the prepared statement, the table name, the commit time, and the expression in the WHERE clause of an UPDATE or DELETE SQL statement are also stored. The data in the *UD holder* is flushed to the catalyst immediately whenever the producer issues a commit (also when lazy flushing is used). In Section 3.3.2, we described that rows may exist in three places during a loading, i.e., in the producer (X_{row} and X_{ar}), in the catalyst (X_{mem}), and in the DW (X). If the data about u/d operations is flushed to the catalyst immediately, the rows in the catalyst, but not yet materialized can also be affected by the u/d operations. The rows that are still in the producer side will eventually be affected by the u/d operations as they will be flushed to the catalyst sooner or later. The rows that have already been materialized into the DW are updated and deleted through normal SQL after the u/d operations are stored in the catalyst.

In the catalyst, we save the insertions, updates and deletions of a row as multiple row versions (see Figure 3.7). The primary key values of a row are used to identify the row and its row versions for updates and deletions. To identify the execution order of insertions, updates and deletions, we assign a sequential number to rid of each row. Besides, we add two additional attribute values of ts and te to each row. The value ts tells from which transaction the row started to exist and the value te tells from which transaction the row got replaced by a new version. A missing value for te in a row means that the row is still valid. This is called *version-based storage*, and we illustrate it using the following example.



T1: The 1st transaction (tid=1)

	rid	ts	te	X(A, B)	
	0	1	1	(0, 0)	r_0
	1	1		(1, 1)	r_1
	2	1		(2, 2)	r_2
U_0	3	1		(0, 1)	

T2: The 2nd transaction (tid=2)

	rid	ts	te	X(A, B)	
	0	1	1	(0, 0)	r_0
	1	1	2	(1, 1)	r_1
	2	1		(2, 2)	r_2
U_0	3	1	2	(0, 1)	
	4	2		(3, 3)	r_3
	5	2		(4, 4)	r_4
U_1	7	2		(0, 2)	

Figure 3.7: Row versions in data segment

Example 11 (Version-based storage) *Consider Example 9 again. Suppose that in the first commit at time 1 we insert the three rows, r_0, r_1 and r_2, then issue an update U_0 = `UPDATE X SET B=1 WHERE A=0`. In the second commit at time 2 we insert the two rows r_3 and r_4, then issue a deletion and an update: D_0 = `DELETE FROM X WHERE A=1` and U_1 = `UPDATE X SET B=2 WHERE A=0`. The two transactions result in the rows and the row versions shown in T_1 and T_2 of Figure 3.7, respectively.*

We assign a sequence number to each transaction, denoted by tid. In the first transaction ($tid = 1$), the three rows r_0, r_1 and r_2 are inserted with ts set to 1 (see the table T_1 in Figure 3.7). In this transaction, the row r_0 is updated by U_0 and therefore a new row is generated and added (see the row with $rid = 3$). At the same time, the value of te of r_0 is updated to the current transaction ID, 1 (see the row with $rid = 0$).

In the second transaction ($tid = 2$), the rows r_3 and r_4 are inserted (see the table T_2 in Figure 3.7). For the deletion D_0 (with $rid = 6$), we just need to update the

te value of the affected row (with $rid = 1$) to 2. This means that this row is not available in or after the transaction with $tid = 2$. Thus, no new row version is stored for the deletion, D_0. For the update, U_1, however, a new row version with the current transaction ID (set in ts) is added (see the row with $rid = 7$) and the value of te in the previous row version is updated, (the te of the row with $rid = 3$ is updated to 2).

The version based storage saves the records representing the updates or deletions with the primary key condition in the WHERE clause (column A is the primary key in this example, denoted by the underline). The catalyst keeps the mappings of the primary key value of the latest row version to its row ID rid in memory. Thus, when a delete or update arrives, the catalyst can quickly locate the position of the older version in data segment in order to update the te value.

To support the updates and deletions on non-primary key conditions, we propose the approach of storing the u/d operations in a *dictionary-based storage*. The structure of this storage is shown in Figure 3.8, which stores the mappings of a commit time to the list of u/d operations within this commit. The rows in the tuple store are updated or deleted on-the-fly by the u/d operations when the rows are read or materialized. We illustrate it using the following example.

Example 12 (Dictionary-based Storage) *Consider Example 9 again. Suppose that we do the updates and deletions with the condition on column B. The transaction at time 1 besides the three insertions of the rows, r_0, r_1 and r_2 also commits the update U_0 = `UPDATE X SET B=1 WHERE B=0` which was executed after the insertions. The transaction at time 2 besides the two insertions of the rows r_3 and r_4 also commits a deletion and an update: D_0 = `DELETE FROM X WHERE B=1` and U_1 = `UPDATE X SET B=0 WHERE B=2`. The two transactions result in the state shown in Figure 3.8.*

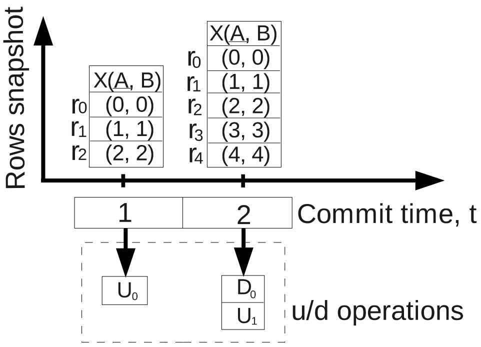


Figure 3.8: Store u/d operations

The upper part of Figure 3.8 shows the snapshot of the rows in the tuple store at the commit times 1 and 2, respectively. Suppose that at time $t > 2$ the rows are

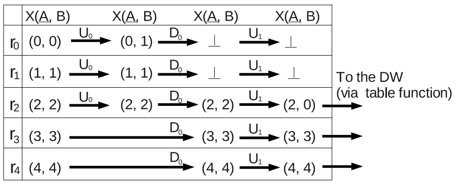

X(A, B)
X(A, B)
X(A, B)
X(A, B)
r0 (0, 0) U0 (0, 1) D0 ⊥ U1 ⊥
r1 (1, 1) U0 (1, 1) D0 ⊥ U1 ⊥
r2 (2, 2) U0 (2, 2) D0 (2, 2) U1 (2, 0)
r3 (3, 3) D0 (3, 3) U1 (3, 3)
r4 (4, 4) D0 (4, 4) U1 (4, 4)
To the DW
(via table function)

Figure 3.9: Apply u/d operations on-the-fly

queried or materialized. The result of applying u/d operations to each row is shown in Figure 3.9 ($\perp$ represents a row has been deleted). The rows, r_0, r_1 and r_2, are applied with U_0, D_0 and U_1, respectively, while r_3 and r_4 are applied with D_0 and U_1. A row is affected if it meets the update or deletion conditions, otherwise, it is unchanged. To apply u/d operations to a row is conducted on-the-fly, and the final result is sent to the table function, while the original row in the tuple store is not modified.

As the original rows in the tuple store are not physically updated or deleted, it is easy to roll back updates and deletions, i.e., by simply deleting an entry from the dictionary storage.

In All-RiTE, we store the u/d operations where the WHERE condition only uses the primary key in the version-based storage, while others are saved in the dictionary-based storage. In the following, we show how to do the on-the-fly updates and deletions when the consumer queries the rows. Algorithm 4 shows the pseudo code.

Suppose that the consumer starts a query at time t. Before rows are read from the tuple store, the consumer connection acquires the current transaction ID tid and uses it throughout the query transaction. All the rows which started to exist in or before the current transaction and have not been replaced before the transaction started will be processed and returned to the consumer (see line 6-19). If we do updates and deletions based on a condition on the primary key, multiple row versions have been created. Lines 7-17, on the other hand, process rows by applying the u/d operations generated by other conditions. First, the entries of *(commit time t', a list of u/d operations)* are read from the dictionary M. The entries in M are sorted in ascending order based on the values of t' and the value of an entry is a list of u/d operations, which are also sorted in ascending order based on the ID values of the u/d operations. Then, we can do the updates and deletions in the right order corresponding to how the prepared statements were executed. Only those u/d operations committed before the query started are applied to rows (see line 8). The u/d operations are applied to a row in sequential order (see line 9-17). To ensure that a u/d operation is applied only to the rows that existed before the u/d operation's prepared statement was executed, we compare the rid of the row and the ID of a u/d operation (note that both IDs are

Algorithm 4 Updates and deletes on-the-fly

```
Input: Query start time $t$, u/d dictionary $M$ and current transaction ID, $tid$
$rows \leftarrow$ Get tuples from the tuple store
for $row \in rows$ do
    $rid \leftarrow row[ID]$
    $ts \leftarrow row[ts]$
    $te \leftarrow row[te]$ or $\infty$;      //$\infty$ if $te$ is undefined
    if $ts \leq tid < te$ then
        for $(t', u/d\ operations) \in M$ do
            if $t' \leq t$ then
                for $ud \in u/d\ operations$ do
                    if $rid < ud.ID$ then
                        $type \leftarrow ud.type$
                        $filter \leftarrow ud.filter$
                        if $type =$'DELETE' and $filter.eval(row) = true$ then
                            $row \leftarrow \perp$
                        if $type =$'UPDATE' and $filter.eval(row) = true$ then
                            $row \leftarrow ud.update(row)$
                    else
                        break
        if $row \neq \perp$ then
            Send $row$ to the consumer
```

drawn from the same sequence). Before a u/d operation is applied to a row, we also need to check if the row matches the update or deletion condition. This is evaluated by the filter in lines 13 and 15 (a filter is created from the expression in the WHERE clause in an UPDATE or DELETE SQL statement). If the filter returns a `true` value, the row is updated or deleted.

Algorithm 4 supports *REPEATABLE READ* transaction isolation level for a query. Consider Example 12 again. Assume that the consumer begins to query rows in a transaction starting between the two commits. If the consumer runs the same query for many times in the query transaction, it will each time only see the u/d operations committed at time 1, i.e., the update U_0. This is because the commit time of D_0 and U_1 is after the start time of the query transaction. Therefore, they are not selected (see line 8 in Algorithm 4). As the transaction ID tid is used to screen the rows throughout the query transaction (see line 6), *REPEATABLE READ* transaction isolation is also ensured for the rows.

3.3.4 Select

We now discuss how the consumer queries are processed. As we mentioned in Section 3.2, rows can be returned both from the catalyst and the DW tables. Behind the scenes, the consumer connection is using the *REPEATABLE READ* isolation level such that the same set of rows from the two places can be read many times. In the following, we first discuss how the rows are read from the catalyst and then discuss how the rows from both places are merged.

The consumer connection can use two approaches for reading the rows from the catalyst: reading rows with IDs in a certain interval and reading rows with specified freshness (or time accuracy). They are used for reading data when the producer is using instant and lazy flush policies, respectively. We present the details in the following.

Read with MinMax Row IDs For a query, the materialized rows are read directly from the DW, while the rows that are not materialized are read from the tuple store in the catalyst. If the rows have been materialized, they can be purged from the catalyst to free space. Care should thus be taken to avoid that when a query starts but before it reads data from the catalyst, some rows become materialized and deleted from the tuple store. To avoid this, we make use of the metadata table *minmax* to *register* rows as “used” before the query starts. The rows that are registered as used will not be deleted even if they have been materialized.

The minmax table has the three columns `tablename`, `min`, `max` which represent the name of a tuple store and the minimal and the maximal row IDs of the rows that should be read from the tuple store. A row in the minmax table thus tells which rows have been committed, but not materialized, in a given tuple store. Note that the materialized rows can be read from the DW and should not be read from the catalyst.

When the rows from the producer are committed, the catalyst updates the value of *max* to the row ID of the last row in the flush (the maximal row ID). When the rows in the tuple store have been materialized to the DW, the catalyst updates the value of *min* to the row ID of the first un-materialized row (the minimal row ID).

We describe the query process by Algorithm 5. When the consumer (driver) starts a query, it starts a new transaction and registers the rows with the row IDs read from the minmax table (see line 2-4). The registration process, however, might fail if a materialization is done between the time when the consumer reads the minmax row IDs and the time when the minmax row IDs are given to the catalyst. In this case, the consumer connection automatically ends the query transaction (recall that a consumer only reads data), starts a new transaction, reads the new minmax row IDs, and registers the rows again. The catalyst gives priority to a consumer going through registration again in order to avoid starvation. If the consumer registers the rows successfully, it can read the rows, and read the same rows for many times in the same query transaction. When the consumer exits the query transaction, the consumer needs to un-register the rows (see line 6).

Algorithm 5 Read the rows with the minmax row IDs

```
Input: The name of table, $name$
1: repeat
2:     Start a transaction
3:     $min, max$ ← ExecuteDBQuery(SELECT min, max FROM minMax WHERE tablename=$name$)
4:     $success$ ← Register the rows with $min \leq row[ID] \leq max$
5: until $success = true$
6: Read the registered rows (allow repeatedly reads)
7: End the transaction and un-register the rows
```

We now describe how the rows are registered and un-registered in the tuple store. All-RiTE allows many consumers to register rows as used at the same time. In the tuple store, a registration counter is used to count the number of the consumers who have registered the rows. (A single counter is used for the registration of all the rows in a data segment, not a counter for each row.) The counter ensures that the registered rows are available to the consumer all the time during a query transaction. The rows that are not registered, but materialized to the DW, can be removed safely from the tuple store. The removal is performed automatically when more space is needed.

Example 13 (Registration) *Assume the producer uses the instant flush, and flushes the three rows r_0, r_1, and r_2 to the catalyst. Again, we assume that a segment can hold no more than two rows. When the rows have been saved in the catalyst, the minmax table is updated to hold the row $(X, 0, 2)$. Further, the registration counters for the* *rows are added (see the first sub-figure in Figure 3.10). Note how, each counter is associated with the lowest row ID in a segment. The first segment holds the rows r_0 and r_1 and thus the minimal row ID is 0. The second segment only holds the row r_2 with row ID 2.*

Suppose that a consumer (Cons. 1) now registers the rows with rowIDs in $[0;2]$ as used. The registration counters for 0 and 2 are therefore updated (see the second sub-figure). The producer continues its work and now flushes the two rows r_3 and r_4. The row r_3 fits in an already existing segment, while a new segment is made for r_4. A new registration counter is added for the new segment (see the third sub-figure). The minmax table is also updated to hold the row $(X, 0, 4)$. Suppose that another consumer (Cons. 2) now starts a query, and registers the rows with rowIDs in $[0;4]$ as used. Then all registration counters are updated (see the fourth sub-figure). The rows r_0, r_1, and r_2, are now registered as used by two consumers.

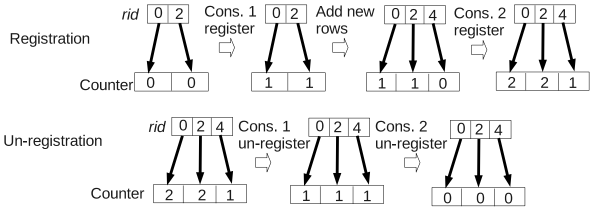


Figure 3.10: Rows registration and un-registration

The process of un-registration is a reverse process of the registration, which is shown in the lower part of Figure 3.10.

Read with Time Accuracy The consumer connection supports reading of data with a specified time accuracy. This is relevant when the producer is using a lazy flush policy (otherwise all committed data is anyway available to the consumer, and using a specified time accuracy becomes not necessary, but not harmful either). The consumer can set the time accuracy (or the freshness of rows) by using the function `ensureAccuracy(freshness)`, an extension to the standard JDBC prepared statement interface. For example, if the prepared statement for a SELECT is set to have a required accuracy of five minutes, at least the rows committed five minutes ago will be available. It may, however, also see newer data, for example if another consumer shortly before had asked for an accuracy of one minute. This extension can thus reduce the amounts of flush operations if the consumers do not need to see the very fresh data.

When given a required accuracy, the consumer connection checks if all rows committed at or before the current time minus the required freshness are available in tuple store. If the tuple store holds all those rows, the catalyst can simply give the rows to the consumer. If the tuple store does not hold some of the rows, the catalyst forwards the request to the producer. Algorithms 6 and 7 define the process. Suppose the consumer asks for the rows with the freshness of n time units. In line 1, the catalyst finds the time t for which all previously committed rows should be available. The catalyst compares t and the latest commit time t_m of the rows in the tuple store (line 2). If $t_m < t$, it means that the archive on the producer side might still hold some needed rows committed between t_m and t. The catalyst thus requests the producer to flush the rows. The catalyst also creates a *locker* with the associated time t (see line 4-5) to suspend the execution of the read thread (see line 8). The locker will resume execution once the requested rows have been flushed to the catalyst, and then the read thread can read the rows and send them to the consumer. The catalyst allows many concurrent consumers, each of which can make the catalyst request data from producers. The created lockers are added to a priority queue, Q_{locker}, in which the lockers are sorted in ascending order based on the time t (see line 6). On the producer side, the rows in the archives are flushed on demand as previously discussed. The rows with commit times no later than the requested time t are flushed to the catalyst. They are flushed in the same order as they were committed. If the producer does not have any data committed at time t, it is required to send an *empty update* telling that no rows were committed at time t.

Algorithm 6 Read the rows with specified time accuracy

```
Input: The data freshness, $n$ time unit
1: $t \leftarrow$ Current system time $- n$
2: $t_m \leftarrow$ Get the latest commit time from the time index
3: if $t > t_m$ then
4:     $locker \leftarrow$ Create a read locker
5:     $locker.time = t$
6:     Add $lock$ to a priority queue, $Q_{locker}$
7:     Request the producer to flush the rows committed no later than $t$
8:     $locker.await(timeout)$        ▷ Wait until it is unlocked by another thread or timeout
9: Send the consumer the rows with commit time $\leq t$
```

Recall that a read thread in the catalyst is suspended while the catalyst requests data from the producer. When the data is flushed to the catalyst, the catalyst therefore compares the commit time of the flushed data to the commit times that sleeping threads are waiting for. If the needed rows have been flushed, the read thread is resumed. The empty update can ensure that all the read threads will be resumed if the producer does not hold the data with the requested commit time. Algorithm 7 shows the steps.

Algorithm 7 Add flushed rows and wake-up read threads

```
Input: A set of rows $R$ and their commit time $t'$
Update the tuple store with $R$ and the time index with $t'$
while $Q_{locker} \neq \emptyset$ do
    $locker \leftarrow$ Get the first element from $Q_{locker}$
    if $locker.time \leq t'$ then
        $locker.signal()$
        Delete $locker$ from $Q_{locker}$
    else
        break
```

Union with the Rows Read from the DW The consumer does not read the rows from the catalyst directly. Instead, it reads the rows through the table function as mentioned in Section 3.2. Recall that the consumer connection executes queries towards the DW in the REPEATABLE READ isolation level. The consumer first reads the minmax row IDs and registers the rows as used. It is then ensured to be able to read the rows that existed in the catalyst when it started. The query will only see the rows that have already existed in the DW before the query began, while the rows materialized after the query started are invisible to the query. Therefore, even if the registered rows are materialized to the DW during the query transaction, the query is still sure to only see the rows once.

When the consumer driver reads the rows from the catalyst, it sends the minmax row IDs and the user-specified time accuracy to the catalyst. If the time accuracy is not set, the consumer driver sends the default value 0 which means that all committed data should be seen.

To hide all complexity from users, the administrator can create a view that unions the (materialized) rows read from the DW table and the rows read from the catalyst through the table function.

Program 1 Create the view to union the rows

```
CREATE VIEW v AS SELECT * FROM dwtable UNION ALL
SELECT * FROM tablefunction('dwtable',
        (SELECT min FROM minmax WHERE tablename='dwtable',
        SELECT max FROM minmax WHERE tablename='dwtable');
```

If the view v is used instead of the $dwtable$, end users do not have to think about where the rows come from. As the consumer connection is using REPEATABLE READ isolation level, a single query on the view can be executed many times and each time gets the same results. However, if the transaction for the query ends, and the query is executed again in a new transaction, the latest updated rows can be seen.

3.4 Tuple Store Optimization

In the catalyst, the data in a tuple store is typically written by one producer and read by many consumers. Lock-based approaches for managing the shared data are not optimal for *read* and *write* performance, e.g., when the producer flushes rows into the catalyst and consumers read them, there is a contention between the write thread and read threads. To cope with this conflict, the tuple store maintains *dual maps*, namely a writeable map and a readable map, as shown in Figure 3.11. The writeable map is only for the write thread to write data into, while the readable map is only for the read threads to read data from.

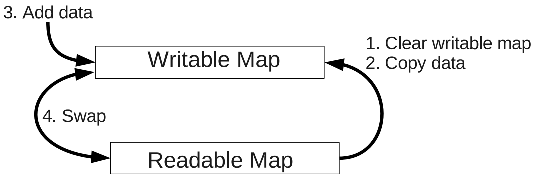


Figure 3.11: Dual data stores

Figure 3.11 describes a scenario where rows are added. Before the rows from the producer are stored in the tuple store, the writeable map is cleared and the data in the readable map is copied into the writeable map. The copy is a shallow copy of the objects stored in the maps, i.e., only the references to the objects are copied and thus it is not expensive. Before the data is added to the writeable map, the unused data is purged from the map. This includes rows that are materialized and not registered as used. Once all the data flushed by the producer has been stored, the writeable map is swapped with the readable map after an exclusive write lock is acquired.

The tuple store maintains separate writeable and readable maps for data segments, row IDs, indices, and u/d operations. The swapping of the these maps from writeable ones to readable ones are done together. This makes it very easy to roll back simply by discarding the data in the writeable maps, i.e., not swap the maps from writeable to readable at the final step. Among the maps, the one that consumes the most memory is the map storing data segments. To allow the system to claim back memory, we use "soft references" [74] to the data segment objects such that infrequently used data segments can be flushed out to files if needed. The data in the files will be read back into the main memory when a consumer queries the data.

3.5 Performance Study

3.5.1 Environment Settings and Test Data

We now present the performance study of All-RiTE. All experiments are carried out on a Dell OptiPlex 960 workstation equipped with a 2.66GHz Intel(R) Core(TM)2 Quad processor, a 320GB SATA hard drive (7200rpm, 16MB Cache and 3.0Gb/s), and 3.0GB RAM running with Fedora 14.0 Linux with kernel 2.6.35. The catalyst, producer, and consumer all run on java-6-sun-1.6.0.25 JVM with the options: -Xms1024m -Xmx2048m. PostgreSQL 8.3.5 is installed on the same workstation and used as the DW DBMS. The DBMS has the following settings, "shared_buffers=512MB, temp_buffers=128MB, work_mem=56MB, maintenance_work_ mem=256MB, checkpoint_segments=20" and default values for other configuration parameters.

We use source data from the TPC-H benchmark [88] but with the schema modified to be a star schema (see Figure 3.12). The data generator for this star schema and the All-RiTE sources are available at `people.cs.aau.dk/~xiliu/rite`. We assume that the orders are created in the data source and that some will be changed by updates and deletions. In the experiments, we first study inserts, on-the-fly updates and deletes, and selects from a table. Then, we study the optimization for the on-the-fly updates and deletions for the tuple store. Finally, we study loading of data into multiple tables. We will compare All-RiTE with the standard JDBC driver, DBMS bulk loader and the system RiTE [85].

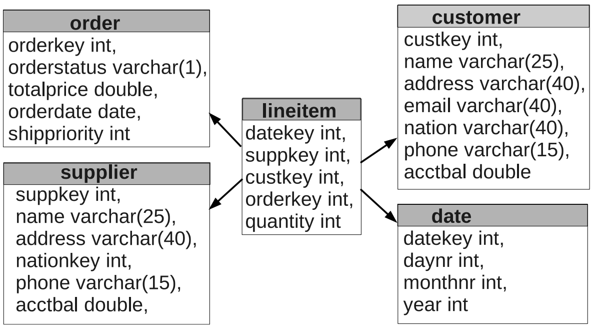


Figure 3.12: Star schema

3.5.2 Insert Only

We consider inserting rows into the `order` dimension table. When using the standard JDBC driver, we insert the data by means of both traditional prepared statements and JDBC batches where values from several executions of a prepared statement are grouped together to reduce communication costs. When using bulk loading, we first write the data to a comma separated file and then let PostgreSQL bulk load the file's content to the table. When using RiTE and All-RiTE, we both load data with and without materialization. All-RiTE uses lazy flushing, but the data is always flushed to the catalyst before the producer finishes. In all experiments, prepared statements are used whenever possible.

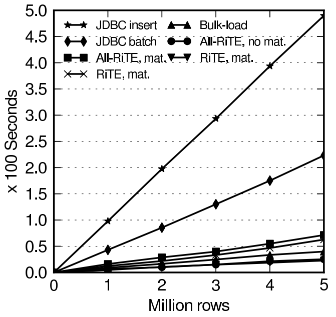


Figure 3.13: Long transaction

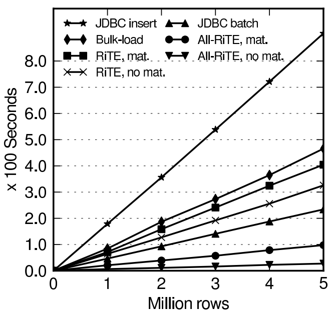


Figure 3.14: Short transaction

We first consider long transactions where all data is inserted before the single commit operation. The results are shown in Figure 3.13. The loading times scale linearly in the number of rows. We compute the throughputs from the slopes of the lines: 10,206 rows/s (JDBC insert), 22,397 rows/s (JDBC batch), 51,376 rows/s (All-RiTE with materialization), 58,342 rows/s (RiTE with materialization), 146,422 rows/s (bulk loading), 188,733 rows/s (All-RiTE without materialization) and 196,576 rows/s (RiTE without materialization).

To better simulate the processing of real-world live DW data, we now load data in short transactions with a commit for every inserted 100 rows. Figure 3.14 shows the test results (note that the y-scale is different to Figure 3.13). We get the following throughputs: 5,522 rows/s (JDBC insert), 10,733 rows/s (bulk loading), 11,376 rows/s (RiTE with materialization), 14,887 rows/s (RiTE without materialization), 21,405 rows/s (JDBC batch), 50,651 rows/s (All-RiTE with materialization) and 187,057 rows/s (All-RiTE without materialization).

In the long transaction, the throughput of RiTE is slightly higher than that of All-RiTE (the lines almost overlap in Figure 3.13) since the implementation of All-RiTE needs to consider handling the updates and deletions. Thus, the system becomes more complex. In the short transaction, the throughput of All-RiTE, however, is much higher than that of RiTE (4.4 and 12.5 times higher for loading with and without materialization, respectively). It is because All-RiTE is designed for processing live DW data (for which short transactions are typically used). All-RiTE can accumulate live DW data in the producer side, and move the data from the producer to the catalyst, and finally to the DW together. In contrast, RiTE does data movement for each commit, and its performance thus degrades dramatically for the short transactions. If we consider the case where the rows are physically stored in the DW, bulk loading has the highest throughput in the long transactions, but the throughput decreases greatly in the short transactions (decreasing up to 92%) since bulk loading is only efficient for big data set. All-RiTE, however, maintains its high performance for short transactions, which is 4.7, 4.4, 9.2, and 2.4 times faster than bulk loading, RiTE, JDBC insert and JDBC batches, respectively. When the burst mode is considered, in which the data is continuously flushed to the catalyst, but not materialized to the DW, the throughput of All-RiTE is 12.6 times higher than that of RiTE. Therefore, All-RiTE can process data quickly for the short transactions, and the advantages for live DW data are clearly seen.

3.5.3 Update/Delete On-the-fly

We now test on-the-fly updates and deletions. For All-RiTE, we load, modify on-the-fly, and finally materialize the rows into the DW. For the alternatives (JDBC, RiTE and bulk loading), we first load the rows into the DW and then modify the rows in the DW through a JDBC connection. Again, we use short transactions to simulate the processing of live DW data. In the experiment, we add a fixed number of rows (10 million) by insertions, but vary the total number of updates and deletions. Further, in each test run –across the transactions– we perform the same number of updates and deletions. For example, 2 million u/d operations consist of 1 million updates and 1 million deletes. The following SQL statements are used: `UPDATE order SET orderstatus=?, totalprice=?, orderdate=?, shippriority=? WHERE orderkey=?` and `DELETE FROM order WHERE orderkey=?`. The question marks are replaced by scalar values when the prepared statement is executed. As the SQL statements both use an *equals condition* on the primary key, a single execution of one of the statements only affects one row. Figure 3.15 shows the results. We compute the average loading throughputs which are: 3,642 rows/s (JDBC insert), 8,015 rows/s (bulk loading), 8,245 rows/s (RiTE), 8,916 rows/s (JDBC batch) and 32,542 rows/s (All-RiTE).

We now specifically test on-the-fly updates and on-the-fly deletions. We consider the previous SQL statements again, but now use a range condition. Thus we change the WHERE clause to `orderkey < ?`. In each run, we set a different scalar value to use in the range condition. For example, setting it to 2 million makes 2 million rows be updated or deleted. Figure 3.16 and 3.17 show the results for updates and deletions, respectively. We compute the average loading throughputs. For the updates on-the-fly, the throughputs are 6,675 rows/s (JDBC insert), 12,224 rows/s (bulk loading), 16,197 rows/s (RiTE), 22,340 rows/s (JDBC batch) and 63,591 rows/s (All-RiTE). For the on-the-fly deletions, the throughputs are 6,883 rows/s (JDBC insert),

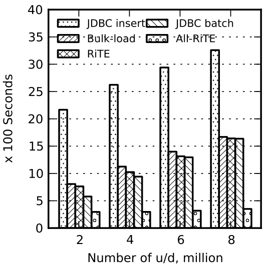


Figure 3.15: On-the-fly updates/deletions, 10million rows

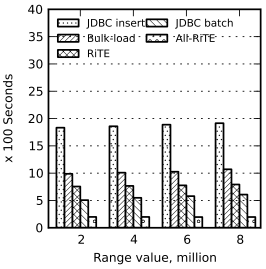


Figure 3.16: Load with updates by range, 10million rows

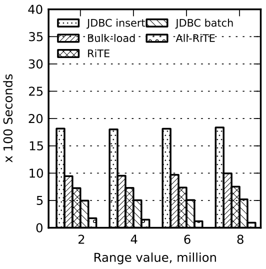


Figure 3.17: Load with deletions by range, 10million rows

12,954 rows/s (bulk loading), 16,996 rows/s (RiTE), 24,681 rows/s (JDBC batch) and 94,036 rows/s (All-RiTE).

In the above experiments, All-RiTE again achieves the highest throughputs. This is due to the implementation for performing the on-the-fly updates and deletions in which the u/d operations lead to different row versions stored in the data segments. The version-based storage achieves very efficient updates and deletions. When doing updates and deletions with range conditions, All-RiTE can also achieve efficient updates and deletions because it can quickly retrieve a u/d operation from the hash dictionary and apply it to a row. The other methods using standard JDBC, RiTE and bulk loading, however, do not provide built-in support for on-the-fly updates and deletions. We, thus, modify the data through the underlying DW DBMS after the data is loaded. This takes more time and the underlying DBMS also causes some overhead such as the time used for writing the transaction logs. For the other approaches, their times are more or less the same since they modify the rows through the DW DBMS. The time difference is mainly resulting from the loading. In our experiment for insert, we have already seen that the performance of bulk loading is between JDBC insert and JDBC batch, and slightly better than RiTE for short transactions. This also applies for on-the-fly updates and deletions. All-RiTE again has the highest efficiency, 4, 3.9, 8.9, and 3.6 times faster than bulk loading, RiTE, JDBC insert and JDBC batch, respectively.

3.5.4 Select

We now study the performance of select operation. We compare All-RiTE with the select from the table in DW, and the select of RiTE. In each test, the rows are first loaded into the memory table or the table in the DW. We select all the rows into the Linux null device, `/dev/null` and measure the average time for five repeated tests. The results are plotted in Figure 3.18. For All-RiTE, the following two methods for reading rows from the memory table are evaluated: M1) We read raw bytes from data segments, convert the bytes into rows of Java types (in order to apply filters/where conditions to the rows), and convert the selected rows back into sequences of bytes to be transferred to the table function (see the line *Read mem. table by M1*). M2) We directly transfer the bytes from data segments to the table function and then select the rows in the table function (see the line *Read mem. table by M2*). Figure 3.18 shows that the time scales nearly linearly in the number of rows selected for each. From the slopes of the lines, we compute the following throughputs: 63,692 rows/s (from mem. table by M1), 82,608 rows/s (RiTE), 99,955 rows/s (from mem. table by M2), and 107,655 rows/s (from ordinary table). The results show that the throughput of M2 is 57% and 21% higher than M1 and RiTE, respectively. This is because M1 has to do two data conversions from and to byte sequences. However, when we use M2, the throughput is close to the throughput of selecting the rows directly from an ordinary table in the DW.

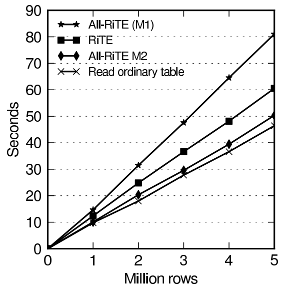


Figure 3.18: Select all rows

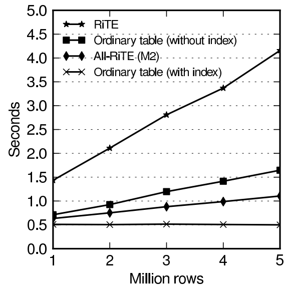


Figure 3.19: Select 50k rows

Instead of selecting all rows, we now compare the selects on a fixed-number of rows using the condition `WHERE orderkey between 1 and 50000`. We use the select approach M2 for All-RiTE, and compare with RiTE and the selects of the rows from the ordinary `order` table (with and without index on `orderkey` column, respectively). Figure 3.19 shows the test results when the number of the rows is scaled from 1 to 5 million. As seen, the select from an ordinary table with the condition on an indexed column maintains the constant throughput, 99,581 rows/s. For RiTE and All-RiTE, since the reading of the rows from catalyst is through the table function with minmax row IDs (see Program 1 in Section 3.3.4), it requires to scan all the rows between the minmax row IDs. Thus, it is neutral to compare with the selects on an ordinary table without index. In this case, All-RiTE has the best throughput, which is 45,284 – 78,864 rows/s. The ranges of the throughput for RiTE and the ordinary table are 12,025 – 34,843 rows/s and 30,376 – 70,224 rows/s respectively.

Thus, All-RiTE shows the advantage over an ordinary table for the selects with the condition non-indexed columns.

3.5.5 Concurrent Read and Write

In the tuple store, we use dual maps, one for writing and one for reading. We now compare this scheme with a simple scheme storing data in a single map. The single map was used in our original design with synchronization to control the concurrency of producers and consumers accessing the common data. *Reads* were given a higher priority than *writes*.

We study the influence on write performance by concurrent reads for the above two designs. We load five million rows into the `order` table in short transactions. We execute the SQL statement `SELECT count(*) FROM order` continuously to simulate concurrent reads. We measure the times to insert data with and without the concurrent reads for the two designs (see Figure 3.20). We can see that: 1) When a single map is used, the concurrent reads drastically degrade the write performance. The time to insert the data is five times higher than *without concurrent reads*. This is because the priority of read threads is higher than the priority of the write thread in this design. 2) When the dual maps are used, the concurrent reads have very limited impact on the write performance. The time it takes to *write with concurrent reads* is only slightly higher than the time it takes to write without concurrent reads.

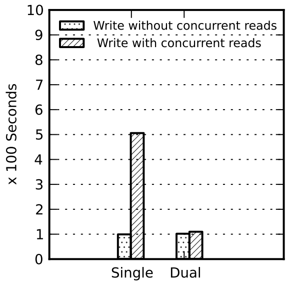


Figure 3.20: Tuple store designs

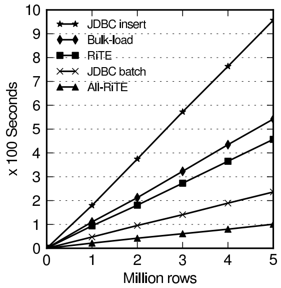


Figure 3.21: Concurrent read/write

We now use the data store design with dual maps in All-RiTE, and compare its loading performance with the standard JDBC driver, RiTE and bulk loading by scaling the amount of rows from one to five millions (see Figure 3.21). We compute the write throughputs from the slopes of the lines: 5,333 rows/s (JDBC insert), 7,621 rows/s (RiTE), 10,417 rows/s (bulk loading), 21,142 rows/s (JDBC batch), and 48,954 rows/s (All-RiTE). The performance advantage of All-RiTE thus remains when loading concurrently with reads. We now compare with the results without concurrent reads (see Figure 3.14). The throughput of RiTE drops most, 33% lower, while the others drop slightly, between 1% and 4% lower. Compared with RiTE, the design of the data store in All-RiTE achieves good effect. The influence on write performance is only 3.3%, which is mainly due to the impact on the resource competition among the reads.

In addition, we have also studied the influence on read performance by concurrent writes in All-RiTE for the two data store designs. We find that the impacts on both are insignificant, i.e., the read throughput remains around 99,900 rows/s (using the reading method M2). The concurrent writes also have little impact to the reads of RiTE (about 82,600 rows/s) since the reads prioritizes the writes.

3.5.6 Loading Multiple Tables

We now study All-RiTE's support for processing multiple tables and how it affects the performance. In this experiment, we use multiple threads to load the star schema. Each thread loads data into a separate table. The number of rows for the *lineitem* fact table is scaled from two to ten millions. We use a fixed number of rows for the four dimension tables: one million rows for the *order* table, and 10,000 rows for each of the other three. Each thread uses short transactions. Figure 3.22 shows the results (RiTE is not tested for it does not support loading multiple tables together). We use the number of *lineitem* facts as the X coordinates since the thread that loads the facts determines the overall time. The results show that All-RiTE outperforms the standard JDBC driver and bulk loading, also when processing multiple tables.

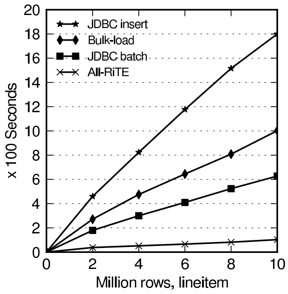


Figure 3.22: Load for star schema

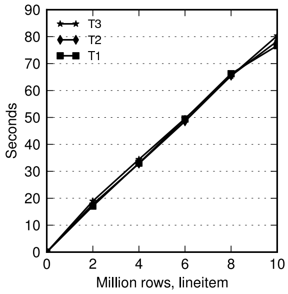


Figure 3.23: Load for multiple tables, 1 CPU

To further study the support for loading of multiple tables, we design the following test cases: T1) We use a single thread to load facts into *lineitem* table scaled from two to ten million. T2) We create a new fact table in the DW, but with the same schema as *lineitem*. We partition the facts into two equal data sets, and use two threads to load the two tables. The data from both threads goes through a single catalyst into the target tables. T3) We use the same scenario as T2, but use two separate catalysts. We do the loading on *one* CPU core (the others are disabled by starting up system with the Linux kernel parameter, `maxcpus=1`) to be able to see if the multiple tuple stores/catalysts affect the performance. The results are shown in Figure 3.23 in which the lines for the three tests almost overlap. We find the throughput to be 122,324 rows/s. Thus, using All-RiTE to load the same number of rows but into different tables introduces no extra overhead.

3.5.7 Summary of Performance Study

We have performed experiments to evaluate All-RiTE by comparing with the standard JDBC driver, RiTE and DBMS bulk loading. All-RiTE provides a new "sweet spot" for processing live DW data in short transactions. For short transactions, All-RiTE's performance is 4.7, 4.4, 9.2, and 2.4 times better than bulk loading, RiTE, JDBC insert, and JDBC insert with batches, respectively. This is an important contribution since short transactions are typically used for live DW data. Thus, with All-RiTE it is possible to insert data quickly and also make it available to consumers quickly.

For long transactions, All-RiTE has the comparable performance to RiTE, and outperforms traditional JDBC insert and batch. Compared with bulk loading, All-RiTE has better performance for the burst mode, while bulk loading is better for materializing the data into DW. And while the performance of bulk loading degrades dramatically as transaction sizes are reduced, All-RiTE is nearly not affected by this.

Unlike bulk loading, RiTE and the standard JDBC driver, All-RiTE has built-in support for on-the-fly updates and deletions. Thus All-RiTE can process live DW data very efficiently. Our experiments show that All-RiTE works 4, 3.9, 8.9 and 3.6 times faster than bulk loading, RiTE, JDBC insert, and JDBC batch, respectively.

We have also evaluated the select operation. All-RiTE has the comparable performance to the selection on the ordinary table in DW. When using the select with the condition on a non-index column, All-RiTE is faster than the select on an ordinary table and RiTE.

3.6 Related Work

In recent years, real-time data warehousing has received much attention. The papers [10, 41, 91] review state-of-the-art, analyze the problems, and discuss the challenges for the real-time data warehousing. [15, 72, 92] introduce ETL modeling techniques for real-time refreshment. All-RiTE extends a previous solution, RiTE [85]. RiTE only supports INSERTs and only from a single producer. All-RiTE supports several producers and does also support modifications of the inserted data (i.e., live DW data) which requires more advanced techniques. Further, All-RiTE can support larger data amounts due to its use of data segments and offers several implementation optimization. [68] uses replicated tables to hold near-real-time data, and the data is updated into the original tables when the DW is offline. All-RiTE, on the other hand, uses the catalyst to accumulate near-real-time data before materializing it to the DW, and also supports SELECT, UPDATE and DELETE operations. Users can control the movement of data between producers and the catalyst (i.e., when to flush) and between the catalyst and the DW (i.e., when to materialize). Bruckner et al. [17, 69] present a container-based ETL architecture which allows ETL tasks to do near-real-time data integration in a J2EE container. The container-based ETL, however, lacks flexibility and causes performance loss due to the container. In comparison, All-RiTE is middleware which easily can be integrated with other ETL programs just by switching the database drivers. [54] summarizes the continuous loading of state-of-art. All-RiTE gives the producer full control over units of work to commit together and is more flexible with respect to the freshness of the data to read. Other works [33, 47, 63] for online data warehouse updates are orthogonal to All-RiTE.

To store the results of updates and deletions as different row versions in All-RiTE is similar to type-2 slowly changing dimensions which traces the changes of a dimension by multiple versions and MVCC (multiversion concurrency control). The latter is widely used in DBMSs, including PostgreSQL, Berkeley DB, MySQL, etc. Regarding the tuple store design, a piece of related work is C-store [75] which has a hybrid architecture with a writable store component optimized for inserts and updates as well as a readable store component optimized for query performance. C-store is a column-oriented DBMS used for read-mostly systems such as customer relationship management systems (CRM). Unlike C-store, All-RiTE is a (non-proprietary) middleware system used by ETL programs that process live DW data.

3.7 Conclusion and Future Work

In this chapter, we have presented All-RiTE, a right-time ETL middleware system for live DW data. It supports multiple data processing operations, including INSERT, UPDATE, DELETE and SELECT. All-RiTE consists of a main-memory based catalyst and database drivers for producers and consumers. The drivers are extended from the standard JDBC interfaces which makes it easy and transparent to integrate All-RiTE with other ETL programs. All-RiTE provides built-in support for on-the-fly updates and deletions which are particularly suitable for processing live DW data such as accumulating facts and early-arriving facts. We have presented the innovative storage structure for implementing on-the-fly updates and deletions and the data storage for managing read/write concurrency and optimizing their performance. We have evaluated All-RiTE through extensive experiments and compared with the standard JDBC driver, RiTE and bulk loading. The results show that All-RiTE's loading performance in short transactions is 4.7, 4.4, 9.2 and 2.4 times faster than bulk loading, RiTE, JDBC insert and JDBC batch, respectively. When performing on-the-fly updates and deletions, All-RiTE works 4, 3.9, 8.9 and 3.6 times faster than bulk loading, RiTE, JDBC insert and JDBC batch, respectively. This provides a new “sweet spot” providing the best of both worlds: JDBC-like data availability with bulk load speed.

There are some interesting improvements left for future work. First, supporting parallel processing of data for a table could be an interesting direction, e.g., several processes process data chunks partitioned from the source data simultaneously. Second, the catalyst could also be implemented as a module in the underlying DBMS for a better performance. Third, it would be also interesting to allow indexes and constraints to be declared on the memory table.

Chapter 4

ETLMR: A Scalable Dimensional ETL Framework based on MapReduce

Extract-Transform-Load (ETL) flows periodically populate data warehouses (DWs) with data from different source systems. An increasing challenge for ETL flows is to process huge volumes of data quickly. MapReduce is establishing itself as the de-facto standard for large-scale data-intensive processing. However, MapReduce lacks support for high-level ETL specific constructs, resulting in low ETL programmer productivity. This chapter presents a scalable dimensional ETL framework, *ETLMR*, based on MapReduce. ETLMR has built-in native support for operations on DW-specific constructs such as star schemas, snowflake schemas and slowly changing dimensions (SCDs). This enables ETL developers to construct scalable MapReduce-based ETL flows with very few code lines. To achieve good performance and load balancing, a number of dimension and fact processing schemes are presented, including techniques for efficiently processing different types of dimensions. This chapter describes the integration of ETLMR with a MapReduce framework and evaluates its performance on large realistic data sets. The experimental results show that ETLMR achieves very good scalability and compares favourably with other MapReduce data warehousing tools.

4.1 Introduction

In data warehousing, ETL flows are responsible for collecting data from different data sources, transformation, and cleansing to comply with user-defined business rules and requirements. Traditional ETL technologies face new challenges as the growth of information explodes nowadays, e.g., it becomes common for an enterprise to collect hundreds of gigabytes of data for processing and analysis each day. The vast amount of data makes ETL extremely time-consuming, but the time window assigned for processing data typically remains short. Moreover, to adapt to rapidly changing business environments, users have an increasing demand of getting data as soon as possible. The use of parallelization is the key to achieve better performance and scalability for those challenges.

In recent years, a novel "cloud computing" technology, *MapReduce* [22], has been widely used for parallel computing in data-intensive areas. A MapReduce program implements the *map* and *reduce* functions, which process key/value pairs and are executed in many parallel instances. MapReduce provides programming flexibility, cost-effective scalability and capacity on commodity machines. A MapReduce framework provides off-the-shelf functionality for inter-process communication, fault-tolerance, load balancing and task scheduling to a parallel program. Now MapReduce is becoming increasingly popular and is establishing itself as the defacto standard for large-scale data-intensive processing. It is thus interesting to see how MapReduce can be applied to the field of ETL programming.

The data processing in ETL exhibits the *composable* property. For example, the processing of dimensions and facts can be split into smaller computation units and the partial results from these computation units can be merged to constitute the final results in a DW. This complies well with the MapReduce paradigm in term of *map* and *reduce*. Thus, MapReduce is a good foundation for the ETL parallelization. However, MapReduce is only a generic programming model. It lacks support for high-level DW/ETL specific constructs, such as the dimensional constructs of star schemas, snowflake schemas, and SCDs. An ETL program is inherently complex, which is due to the ETL-specific activities, such as transformation, cleansing, filtering, aggregating and loading. To implement a parallel ETL program is costy, error-prone, and thus leads to low programmer productivity.

In this chapter, we present a parallel dimensional ETL framework based on MapReduce, named *ETLMR*, which directly supports high-level ETL-specific dimensional constructs such as star schemas, snowflake schemas, and SCDs. This chapter makes several contributions: We leverage the functionality of MapReduce to the ETL parallelization and provide a scalable, fault-tolerable, and very lightweight ETL framework which hides the complexity of MapReduce. We present a number of novel methods which are used to process the dimensions of a star schema, snowflaked di-

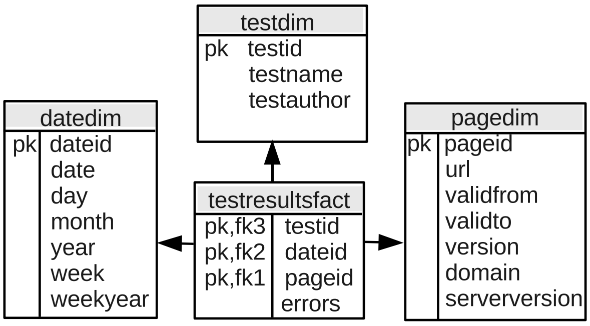


Figure 4.1: Star schema of the running example

mensions, SCDs and data-intensive dimensions. In addition, we introduce the offline dimension scheme which scales better than the online dimension scheme when handling massive workloads. The evaluations show that ETLMR achieves very good scalability and compares favourably with other MapReduce data warehousing tools.

The running example: To show the use of ETLMR, we use a running example throughout this chapter. This example is inspired by a project that evaluates the accessibility of web pages. We apply different tests to each of the web pages, and the tests output the number of detected errors to tab-separated files. These files serve as the data sources. We then use ETLMR to process the source data into a star schema in the DW shown in Figure 4.1 (through the necessary data transformations). This schema comprises a fact table and three dimension tables. Note that *pagedim* is a slowly changing dimension table. Later, we will consider a partly snowflaked (i.e., normalized) schema.

The remainder of this chapter is structured as follows: Section 4.2 gives a brief review of the MapReduce programming model. Section 4.3 gives an overview of ETLMR. Sections 4.4 and 4.5 present dimension processing and fact processing, respectively. Section 4.7 introduces the implementation of ETLMR in the Disco MapReduce framework and presents the performance study. Section 4.8 reviews related work. Finally, Section 4.9 concludes the chapter and provides ideas for future work. The appendix compares ETLMR to Hive and Pig by means of concrete ETL solutions.

4.2 MapReduce Programming Model

MapReduce [22] computations are expressed by means of two functions called *map* and *reduce*.

```
map: (k1, v1) -> list(k2, v2)
```

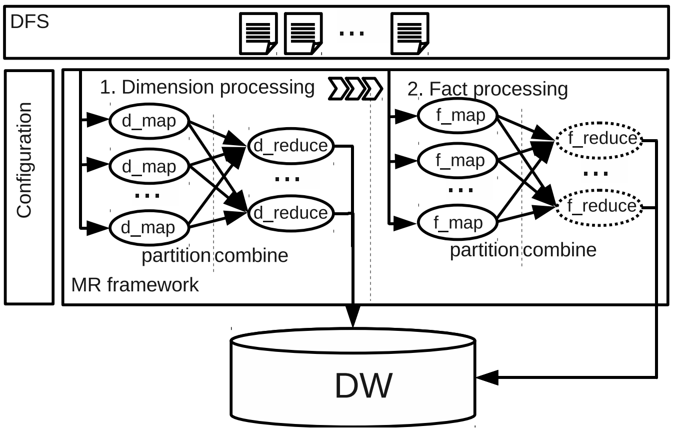


Figure 4.2: ETL Data flow on MapReduce framework

`reduce: (k2, list(v2)) -> list(v3)`

The *map* function, defined by users, takes as input a key/value pair $(k1, v1)$ and produces a list of intermediate key/value $(k2, v2)$ pairs. The MapReduce framework then groups all intermediate values with the same intermediate key $k2$ and passes them on to the *reduce* function. The *reduce* function, also defined by users, takes as input a pair consisting of a key $k2$ and a list of values. It merges or aggregates the values to form a possibly smaller (or even empty) list of values $list(v3)$.

Besides the `map` and `reduce` functions, there are five other functions offered by most MapReduce frameworks, including functions for input reading, data partitioning, combining map output, sorting, and output writing. Users can (but do not have to) specify these functions to fit to their specific requirements. A MapReduce framework achieves parallel computations by executing the implemented functions in parallel on clustered computers, each processing a chunk of the data sets.

4.3 Overview

In this section, we give an overview of ETLMR on a MapReduce framework and describe the data processing phases in ETLMR. Figure 4.2 illustrates the data flow using ETLMR on MapReduce. In ETLMR, the dimension processing is done at first in a MapReduce job, then the fact processing is done in another MapReduce job. A MapReduce job spawns a number of parallel map/reduce task (map/reduce task denotes map tasks and reduce tasks running separately) for processing dimension or fact data. Each task consists of several steps, including reading data from a distributed file system (DFS), executing the map function, partitioning, combining the map output, executing the reduce function and writing results. In dimension processing, the input data for a dimension table can be processed by different processing methods, e.g., the data can be processed by a single task or by all tasks. In fact processing, the data for a fact table is partitioned into a number of equal-sized data files which then are processed by parallel tasks. This includes looking up dimension keys and bulk loading the processed fact data into the DW. The processing of fact data in the reducers can be omitted (shown by dotted ellipses in Figure 4.2) if no aggregation of the fact data is done before it is loaded.

Algorithm 8 ETL process on MapReduce framework

```
1: Partition the input data sets;
2: Read the configuration parameters and initialize;
3: Read the input data and relay the data to the map function in the map readers;
4: Process dimension data and load it into online/offline dimension stores;
5: Synchronize the dimensions across the clustered computers, if applicable;
6: Prepare fact processing (connect to and cache dimensions);
7: Read the input data for fact processing and perform transformations in mappers;
8: Bulk-load fact data into the DW.
```

Algorithm 8 shows the details of the whole process of using ETLMR. The operations in lines 2-4 and 6-7 are the MapReduce steps which are responsible for initialization, invoking jobs for processing dimensions and facts, and returning processing information. Line 1 and 5 are the non-MapReduce steps which are used for preparing input data sets and synchronizing dimensions among nodes (if no DFS is installed). The data reader in ETLMR partitions the data while it reads lines from the input files. Specifically, it supports the following two partitioning schemes.

- *Round-robin partitioning*: This method distributes row number n to task number $n \ \% \ nr\_map$ where nr_map is the total number of tasks and $\%$ is the modulo operator. By this method, the data sets are equally divided and processed by the parallel tasks. Thus, it ensures the load balance.

- *Hash partitioning*: This method partitions data sets based on the values of one or several attributes of a row. It computes the hash value h for the attribute values and assigns the row to task number $h \ \% \ nr\_map$.

All parameters are defined in a configuration file, including declarations of dimension and fact tables, dimension processing methods, number of mappers and reducers, and others. The parameteres are summarized in Table 4.1. The use of parameters gives the user flexibility to configure the tasks to be more efficient. For example, if a user knows that a dimension is a data-intensive dimension, she can add it to the list $bigdims$ so that an appropriate processing method can be chosen to achieve better performance and load balancing. The configuration file is also used for specifications of user-defined functions (UDFs) for data processing.

Table 4.1: The configuration parameters

Parameters	Description
Dim_i	Dimension table definition, $i = 1, \ldots, n$
$Fact_i$	Fact table definition, $i = 1, \ldots, m$
Set_{bigdim}	data-intensive dimensions whose business keys are used for partitioning the data sets if applicable
$Dim_i(a_0, a_1, ..., a_n)$	Define the relevant attributes $a_0, a_1, ..., a_n$ of Dim_i in data source
$DimScheme$	Dimension scheme, online/offline (online is the default)
nr_reduce	Number of reducers
nr_map	Number of mappers

4.4 Dimension Processing

In ETLMR, each dimension table has a corresponding definition in the configuration file. For example, we define the object for the dimension table *testdim* of the running example by *testdim = CachedDimension(name='testdim', key='testid', defaultidvalue =-1, attributes=['testname', 'testauthor'], lookupatts=['testname',]).* It is declared as a cached dimension which means that its data will be temporarily kept in main memory during the processing. ETLMR also offers other dimension classes for declaring different dimension tables, including *SlowlyChangingDimension* and *SnowflakedDimension*, each of which is configured by means of a number of parameters for specifying the name of the dimension table, the dimension key, the attributes of dimension table, the lookup attributes (which identify a row uniquely), and others. Each class offers a number of functions for dimension operations such as *lookup*, *insert*, *ensure*, etc.

ETLMR employs MapReduce's *map*, *partition*, *combine*, and *reduce* to process data. This is, however, hidden from the user who only specifies transformations to apply to the data and declarations of dimension tables and fact tables. A map/reduce task reads data by iterating over lines from a partitioned data set. A line is first processed by *map*, then by *partition* which determines the target reducer, and then by *combine* which groups values having the same key. The data is then written to an intermediate file (there is one file for each reducer). In the reduce step, a reduce reader reads a list of key/values pairs from an intermediate file and invokes *reduce* to process the list. In the following, we present different approaches to process dimension data.

4.4.1 One Dimension One Task

In this approach, map tasks process data for all dimensions by applying user-defined transformations and by finding the relevant parts of the source data for each dimension. The data for a given dimension is then processed by a single reduce task. We name this method *one dimension one task* (*ODOT* for short).

The data unit moving around within ETLMR is a dictionary mapping attribute names to values. Here, we call it a *row*, e.g., *row*={*'url':'www.dom0.tl0/p0.htm','size': '12553','serverversion':'SomeServer/1.0', 'downloaddate':'2011-01-31','lastmoddate': '2011-01-01', 'test':'Test001', 'errors':'7'*}. ETLMR reads lines from the input files and passes them on as rows. A mapper does projection on rows to prune the unnecessary data through the defined UDFS for a target dimension table, and makes key/value pairs to be processed by reducers. If we define dim_i for a dimension table and its relevant attributes, $(a_0, a_1..., a_n)$, in the data source schema, the mapper will generate the map output, $(key, value) = (dim_i.name, \prod_{a_0,a_1,...,a_n}(row))$ where $name$ represents the name of dimension table. The MapReduce partitioner does hash partitioning the map output based on the key, i.e., $dim_i.name$, such that the data of dim_i will be shuffled to a single reducer (see Figure 4.3). Thus, each reducer will eventually load the data into a particular dimension table. To optimize, the values with identical keys (i.e., dimension table name) are combined in the combiner before they are sent to the reducers such that the network communication cost can be reduced. In a reducer, a row is first processed by UDFs to do data transformations, then the processed row is inserted into the dimension store, i.e., the dimension table in the DW or in an offline dimension store (described later). When ETLMR does this data insertion, it has the following *reduce* functionality: If the row does not exist in the dimension table, the row is inserted. If the row exists and its values are unchanged, nothing is done. If there are changes, the row in the table is updated accordingly. The ETLMR dimension classes provide this functionality in a single function, $dim_i.ensure(row)$. For an SCD, this function adds a new version if needed, and updates the values of the SCD attributes, e.g., the validto and version.

We have now introduced the most fundamental method for processing dimensions where only a limited number of reducers can be utilized. Therefore, its drawback is that it is not optimized for the case where some dimensions contain large amounts of data, namely data-intensive dimensions.

4.4.2 One Dimension All Tasks

We now describe another approach in which all reduce tasks process data for all dimensions. We name it *one dimension all tasks* (*ODAT* for short). In some cases, the data volume of a dimension is very large, e.g., the *pagedim* dimension in the running example. If we employ ODOT, the task of processing data for this dimension table

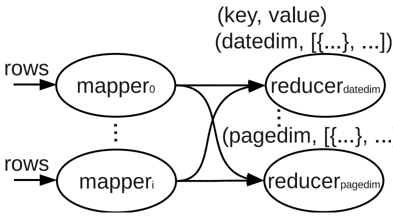


Figure 4.3: ODOT

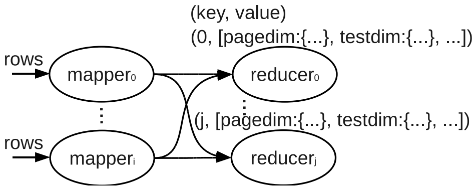

(key, value)
(0, [pagedim:{...}, testdim:{...}, ...])
rows
mapper$_0$
reducer$_0$
(j, [pagedim:{...}, testdim:{...}, ...])
rows
mapper$_i$
reducer$_j$

Figure 4.4: ODAT

will determine the overall performance (assume all tasks run on similar machines). We therefore refine the ODOT in two places, the map output partition and the reduce functions. With ODAT, ETLMR partitions the map output by round-robin fashion such that the reducers receive equally many rows (see Figure 4.4). In the reduce function, two issues are considered in order to process the dimension data properly by the parallel tasks:

The first issue is about the surrogate key assignment and to keep the uniqueness across the all the tasks, e.g., for the dimension values of a particular dimension table. We propose the following two approaches. The first one is to use a global ID generator (by making use of the sequential number generator of the underlying DW RDBMS), and use *post-fixing* (detailed in Section 4.4.4) to merge rows with identical values in the dimension *lookup* attributes (but different key values) into one row. The other approach is to use private ID generators and post-fixing. Each task has its own ID generator, and after the data is loaded into the dimension table, post-fixing is employed to fix the resulting duplicated surrogate key values. This requires the uniqueness constraint on the key of a dimension table to be disabled in the DW RDBMS before the data processing.

The second issue is how to handle concurrency when data manipulation language (DML) SQL, such as UPDATE, DELETE, etc., is issued by several tasks. Consider, for example, the type-2 SCD table *pagedim* for which INSERTs and UPDATEs are frequent (the SCD attributes *validfrom* and *validto* are updated). There are at least two ways to tackle this problem. The first one is row-based commit in which a COMMIT is issued after every row has been inserted so that the inserted row will not be locked. However, row-based commit is more expensive than transaction commit, thus, it is not very useful for a data-intensive dimension table. Another and better solution is to delay the UPDATE to the post-fixing which fixes all the problematic data when all the tasks have finished.

In the following section, we propose an alternative approach for processing snowflaked dimensions without requiring the post-fixing.

4.4.3 Snowflaked Dimension Processing

In a snowflake schema, dimensions are normalized meaning that there are foreign key references and hierarchies between dimension tables. If we consider the dependencies when processing dimensions, the post-fixing step can be avoided. We therefore propose two methods particularly for snowflaked dimensions: *level-wise processing* and *hierarchy-wise processing*.

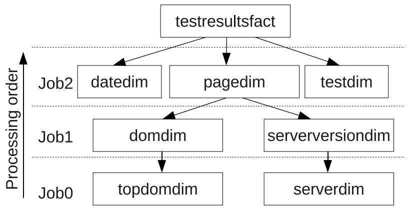


Figure 4.5: Level-wise processing

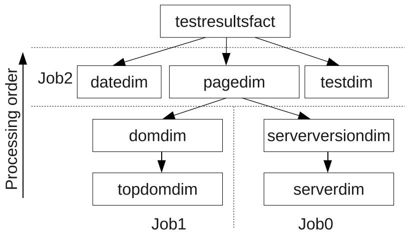


Figure 4.6: Hierarchy-wise processing

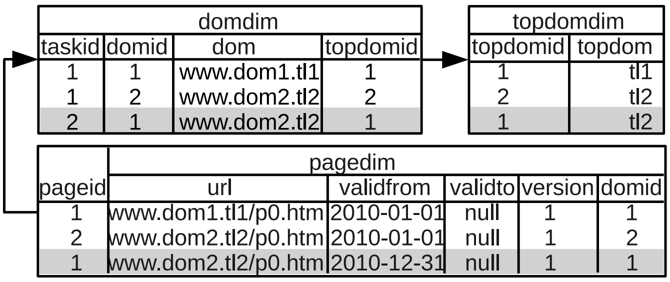
Level-wise processing This refers to processing snowflaked dimensions in an order from the leaves towards the root (the dimension table referred by the fact table is the root and a dimension table without a foreign key referencing other dimension tables is a leaf). The dimension tables with dependencies (i.e., with foreign key references) are processed in sequential jobs, e.g., *Job1* depends on *Job0*, and *Job2* depends on *Job1* in Figure 4.5. Each job processes independent dimension tables (without direct and indirect foreign key references) by parallel tasks, i.e., one dimension table is processed by one task. Therefore, in the level-wise processing of the running example, *Job0* first processes *topdomaindim* and *serverdim* in parallel, then *Job1* processes *domaindim* and *serverversiondim*, and finally *Job2* processes *pagedim*, *datedim* and *testdim*. It corresponds to the configuration *loadorder = [('topdomaindim', 'serverdim'), ('domaindim', 'serverversiondim'), ('pagedim', 'datedim', 'testdim')]*. With this order, a higher level dimension table (the referencing dimension table) is not processed until the lower level ones (the referenced dimension tables) have been processed and thus, the referential integrity can be ensured.

Hierarchy-wise processing This refers to processing a snowflaked dimension in a branch-wise fashion (see Figure 4.6). The root dimension, *pagedim*, derives two branches, each of which is defined as a separate snowflaked dimension, i.e., *domainsf = SnowflakedDimension([(domaindim, topdomaindim)])*, and *serverversionsf = SnowflakedDimension([(serverversiondim, serverdim)])*. They are processed by two parallel jobs, *Job0* and *Job1*, each of which processes in a sequential manner, i.e., *topdomaindim* followed by *domaindim* in *Job0* and *serverdim* followed by *serverversiondim* in *Job1*. The root dimension, *pagedim*, is not processed until the dimensions on its connected branches have been processed. It, together with *datedim* and *testdim*, is processed by the *Job2*.

4.4.4 Post-fixing

As discussed in Section 4.4.2, post-fixing is a remedy to fix problematic data in ODAT when all the tasks of the dimension processing have finished. Four situations require data post-fixing: 1) using a global ID generator which gives rise to duplicated values in the lookup attributes; 2) using private ID generators which produce duplicated key values; 3) processing snowflaked dimensions (and *not* using level-wise or hierarchy-wise processing) which leads to duplicated values in lookup and key attributes; and 4) processing slowly changing dimensions which results in SCD attributes taking improper values.

Example 14 (Post-fixing) *Consider two map/reduce tasks, task 1 and task 2, that process the* `page` *dimension which we here assume to be snowflaked. Each task uses a private ID generator. The root dimension,* `pagedim`*, is a type-2 SCD. Rows with the lookup attribute value url='www.dom2.tl2/p0.htm' are processed by both the tasks.*



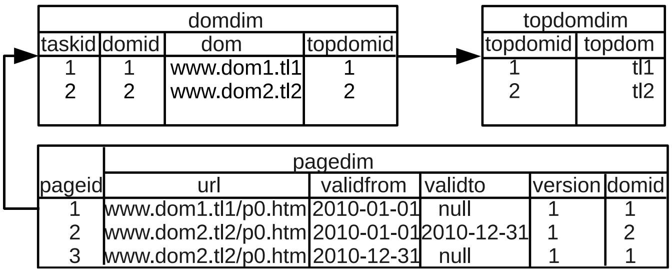
domdim			
taskid	domid	dom	topdomid
1	1	www.dom1.tl1	1
1	2	www.dom2.tl2	2
2	1	www.dom2.tl2	1

topdomdim	
topdomid	topdom
1	tl1
2	tl2
1	tl2

pageid	pagedim				
	url	validfrom	validto	version	domid
1	www.dom1.tl1/p0.htm	2010-01-01	null	1	1
2	www.dom2.tl2/p0.htm	2010-01-01	null	1	2
1	www.dom2.tl2/p0.htm	2010-12-31	null	1	1

Figure 4.7: Before post-fixing

Figure 4.7 depicts the resulting data in the dimension tables where the white rows were processed by task 1 and the grey rows were processed by task 2. Each row is labelled with the `taskid` *of the task that processed it. The problems include duplicated IDs in each dimension table and improper values in the SCD attributes,* `validfrom`*,* `validto`*, and* `version`*. The post-fixing program first fixes the* `topdomaindim` *such that rows with the same value for the lookup attribute (i.e.,* `url`*) are merged into one row with a single ID. Thus, the two rows with* `topdom` *= tl2 are merged into one row. The references to* `topdomaindim` *from* `domaindim` *are also updated to reference the correct (fixed) rows. In the same way,* `pagedim` *is updated to merge the two rows representing www.dom2.tl2. Finally,* `pagedim` *is*



domdim			
taskid	domid	dom	topdomid
1	1	www.dom1.tl1	1
2	2	www.dom2.tl2	2

topdomdim	
topdomid	topdom
1	tl1
2	tl2

	pagedim				
pageid	url	validfrom	validto	version	domid
1	www.dom1.tl1/p0.htm	2010-01-01	null	1	1
2	www.dom2.tl2/p0.htm	2010-01-01	2010-12-31	1	2
3	www.dom2.tl2/p0.htm	2010-12-31	null	1	1

Figure 4.8: After post-fixing

updated. Here, the post-fixing also has to fix the values for the SCD attributes. The result is shown in Figure 4.8.

Algorithm 9 post_fix(*dim*)

```
refdims ← The referenced dimensions of dim
for ref in refdims do
    itr ← post_fix(ref)
    for ((taskid, keyvalue), newkeyvalue) in itr do
        Update dim set dim.key = newkeyvalue where dim.taskid=taskid and dim.key=keyvalue
ret ← An empty list
Assign newkeyvalues to dim's keys and add ((taskid, keyvalue), newkeyvalue) to ret
if dim is not the root then
    Delete the duplicate rows, which have identical values in dim's lookup attributes
if dim is a type-2 SCD then
    Fix the values on SCD attributes, e.g., dates and version
return ret
```

The post-fixing invokes a recursive function (see Algorithm 9) to fix the problematic data in the order from the leaf dimension tables to the root dimension table. It comprises four steps: 1) assign new IDs to the rows with duplicate IDs; 2) update the foreign keys on the referencing dimension tables; 3) delete duplicated rows which have identical values in the business key attributes and foreign key attributes; and 4) fix the values in the SCD attributes if applicable. In most cases, it is not needed to fix something in each of the steps for a dimension with problematic data. For example, if a global ID generator is employed, all rows will have different IDs (such that step 1 is not needed) but they may have duplicate values in the lookup attributes (such that step 3 is needed). ETLMR's implementation uses an embedded SQLite database for data management during the post-fixing. Thus, the task IDs are not stored in the target DW, but only internally in ETLMR.

4.4.5 Offline Dimension

In ODOT and ODAT, the map/reduce tasks interact with the DW's ("*online*") dimensions directly through database connections at run-time and the performance is affected by the outside DW RDBMS and the database communication cost. To optimize this, the *offline dimension* scheme is proposed. In this scheme, the tasks do not interact with the DW directly, but with distributed offline dimensions residing physically in all nodes. The offline dimension scheme has the following characteristics and advantages. First, a dimension is partitioned into multiple smaller-sized subdimensions, and small-sized dimensions can benefit dimension *lookup*s, especially for a data-intensive dimension such as *pagedim*. Second, a high-performance storage system, such as an in-memory DBMS, can be employed to persist dimension data. Dimensions are thus configured to be fully or partially cached in main memory to speed up the *lookup*s. In addition, since offline dimension sheme does not require direct communication with the DW, the overhead (from the network and the DBMS) is greatly reduced. ETLMR has offline dimension implementations for one dimension one task (*ODOT (offline)* for short) and *hybrid*, which are described in the following.

4.4.5.1 ODOT (offline)

Figure 4.9 depicts the run-time architecture with two map/reduce tasks processing data. The data for each dimension table is saved locally in its own store in the node that processes it or in the DFS (shown in the center of the Figure 4.9). The data for a dimension table is processed by one and only one reduce task (as in the online ODOT), which does the following: 1) Select the values of the fields that are relevant to the dimension table in mappers (through the UDFs); 2) Partition the map output based on the names of dimension tables; 3) Process the data for dimension tables by using UDFs in the reducers (a reducer only processes the data for a single dimension table); 4) When all map/reduce tasks have finished, the data for all dimension tables are synchronized across all the nodes if no DFS is installed (Note that a reducer only processes the data for a single dimension table. The data files of the offline dimension stores are synchronized across all the nodes, thus, each of the nodes will eventually have the local copies of the data files of all dimension tables).

4.4.5.2 Hybrid

Hybrid combines the characteristics of ODOT and ODAT. In this approach, the dimensions are divided into two groups, the most data-intensive dimension and the other dimensions. The input data for the most data-intensive dimension table is partitioned based on the business keys, e.g., on the `url` of `pagedim`, and processed by all the map tasks (this is similar to ODAT), while for the other dimension tables,

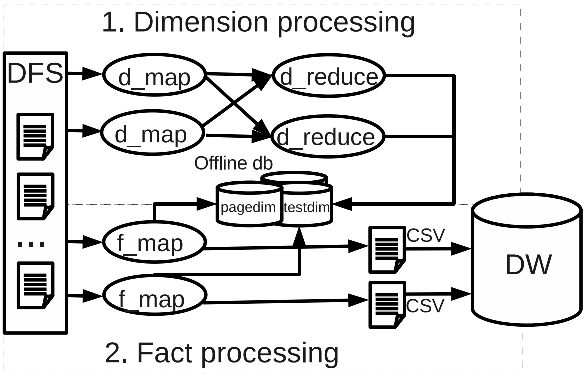


Figure 4.9: ODOT (offline)

their data is processed in reducers, a reducer exclusively processing the data for one dimension table (this is similar to ODOT).

This corresponds to the following steps: 1) Choose the most data-intensive dimension and partition the input data sets, for example, on the business key values; 2) Process the chosen data-intensive dimension and select the required data for each of the other dimensions in the mappers; 3) Round-robin partition the map output; 3) Process the dimensions in the reducers (each is processed by one reducer); 4) When all the tasks have finished, synchronize all the processed dimensions across the nodes if no DFS is installed, but keep the partitioned data-intensive dimension in all nodes.

Example 15 (Hybrid) *Consider using two parallel tasks to process the dimension tables of the running example (see the upper part in Figure 4.10). The data for the most data-intensive dimension table,* `pagedim`*, is partitioned into a number of chunks based on the business key* `url`*. The chunks are processed by two mappers, each processing a chunk. This results in two offline stores for* `pagedim`*,* `pagedim-0` *and* `pagedim-1`*. However, the data for the other dimension tables is processed similarly to the offline ODOT, which results in the other two offline dimension stores,* `datedim` *and* `testdim`*, respectively.*

In the offline dimension scheme, the dimension data in the offline stores is expected to reside in the nodes permanently and will not be loaded into the DW until this is explicitly requested.

4.5 Fact Processing

Fact processing is the second phase in ETLMR. In this phase, ETLMR looks up dimension keys for the facts, does aggregation of measures (if applicable), and loads the

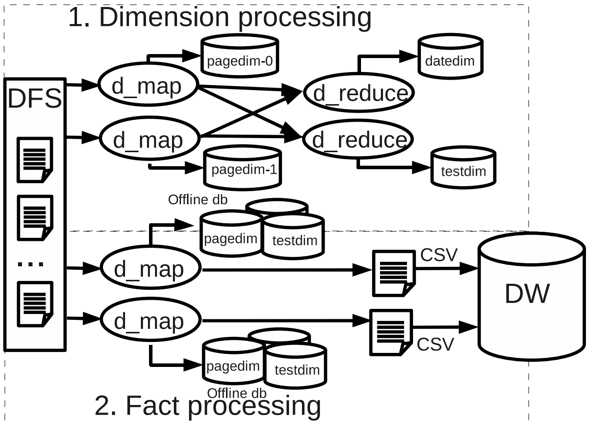


Figure 4.10: Hybrid

processed facts into the DW. Similarly to the dimension processing, the definitions and settings of fact tables are also declared in the configuration file. ETLMR provides the *BulkFactTable* class which supports bulk loading of facts to the DW. For example, the fact table of the running example is defined as *testresultsfact=BulkFactTable(name ='testresultsfact', keyrefs=['pageid', 'testid', 'dateid'], measures=['errors'], bulkloader= UDF_pgcopy, bulksize=5000000).* The parameters are the fact table name, a list of the keys referencing dimension tables, a list of measures, the bulk loader function, and the size of the bulks to load. The bulk loader is a UDF such that ETLMR can be used with different types of DBMSs.

Algorithm 10 shows the pseudo code for processing facts.

Algorithm 10 $process\_fact(row)$

```
Input: A row from the input data and the config
facttbls ← the fact tables defined in config
for facttbl in facttbls do
    dims ← the dimensions referenced by facttbl
    for dim in dims do
        row[dim.key] ← dim.lookup(row)
    rowhandlers ← facttbl.rowhandlers
    for handler in rowhandlers do
        handler(row)
    facttbl.insert(row)
```

The function can be used as the map function or as the reduce function. If no aggregations (such as *sum*, *average*, or *count*) are required, the function is configured to be the map function and the reduce step is omitted for better performance. If aggregations are required, the function is configured to be the reduce function since the aggregations must be computed from all the data. This approach is flexible and good for performance. Line 1 retrieves the fact table definitions in the configuration file and they are then processed sequentially in line 2–8. The processing consists of two major operations: 1) look up the keys from the referenced dimension tables (line 3–5), and 2) process the fact data by the *rowhandlers*, which are user-defined transformation functions used for data type conversions, calculating measures, etc. (line 6–8). Line 9 invokes the insert function to insert the fact data into the DW. The processed fact data is not inserted into the fact table directly, but instead added into a configurably-sized buffer where it is kept temporarily. When a buffer becomes full, its data is bulk loaded into the DW. Each map/reduce task has a separate buffer and bulk loader such that tasks can do bulk loading in parallel.

4.6 Implementation and Optimization

ETLMR is designed to achieve plug-in like functionality to facilitate the integration with Python-supporting MapReduce frameworks. In this section, we introduce the ETL programming framework *pygrametl* which is used to implement ETLMR. Further, we give an overview of MapReduce frameworks with a special focus on Disco [24] which is our chosen MapReduce platform. The integration and optimization techniques employed are also described.

4.6.1 pygrametl

`pygrametl` was implemented in previous work [84] and has a number of characteristics. It is a code-based ETL framework which enables efficient development due to its simplicity and use of Python. `pygrametl` supports processing of dimensions in both star schema and snowflake schema and it provides direct support for slowly changing dimensions. Regardless of the dimension types, the ETL implementation is very concise and convenient. It uses an object to represent a dimension. Only a single method call such as `dimobj.insert(row)` is needed to insert new data. In this method call, all details, such as key assignment and SQL generation, are transparent to users. `pygrametl` supports the most commonly used operations on a dimension, such as `lookup`, `insert`, `ensure`. In the implementation of ETLMR, most functionality offered by `pygrametl` can be re-used, but some parts have been extended or modied to support the MapReduce operations.

4.6.2 MapReduce Frameworks

There are many open source and commercial MapReduce frameworks available. The most popular one is Apache Hadoop [78], which is implemented in Java and includes a distributed file system (HDFS). Hadoop is embraced by a variety of academic and industrial users, including Amazon, Yahoo!, Facebook, and many others [8]. Google's MapReduce implementation is extensively used inside the company, but is not publicly available. Apart from them, many companies and research units develop their own MapReduce frameworks for their particular needs, such as [45, 66, 89].

For ETLMR, we choose the open source framework Disco [24] as the MapReduce platform. Disco is developed by Nokia using the Erlang and Python programming languages. Disco is chosen for the following reasons. First, Disco's use of Python facilitates rapid scripting for distributed data processing programs, e.g., a complicated program or algorithm can be expressed in tens of lines of code. This feature is consistent with our aim of providing users a simple and easy means of implementing a parallel ETL. Second, Disco achieves a high degree of flexibility by providing many customizable MapReduce programming interfaces. These make the integration of ETLMR very convenient by having a plug-in-like effect. Third, unlike the alternatives, it provides direct support for the distributed programs written in Python. Some MapReduce frameworks implemented in other programming languages also claim to support Python programs, e.g., Hadoop, but they require bridging middleware and are, thus, not implementation-friendly.

4.6.3 Integration with Disco

Disco's architecture is similar to the Google and Hadoop MapReduce architectures in which intermediate results are stored as local files and accessed by appropriate reduce tasks. However, the used version (0.2.4) does not include a built-in distributed file system (DFS), but supports any POSIX-compatible DFS such as GlusterFS. If no DFS is installed, the input files are required to be distributed initially to each node so that they can be read locally [24]. To use Disco, the functions to use as map and reduce are passed on as arguments to Disco.

We present the runtime architecture of ETLMR on Disco in Figure 4.11, where all ETLMR instances are running in parallel on many processors of clustered computers (or nodes). This architecture is capable of shortening the time of an ETL job by scaling up to the number of nodes in the cluster. It is an one-master-many-worker architecture. The master is responsible for scheduling the components (tasks) of the jobs to run on the workers, assigning partitioned data sets to the workers, tracking the status, and monitoring the status of the workers. When the master receives ETL jobs, it puts them into a queue and distributes them to the available workers. In each node, there is a worker supervisor started by the master which is responsible for spawning and monitoring all tasks on that particular node. When a worker receives a task, it runs this task exclusively in a processor of this node, processes the input data, and saves the processed data to the DW.

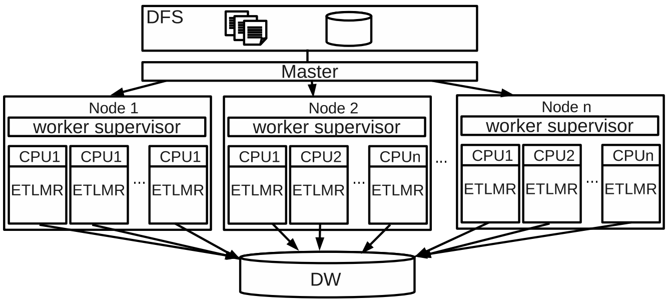


Figure 4.11: Parallel ETL overview

The master-workers architecture has a highly fault-tolerant mechanism. It achieves reliability by distributing the input files to the nodes in the cluster. Each worker reports to the master periodically with the completed tasks and their status. If a worker falls silent for longer than an interval, the master will record this worker as dead and send the node's assigned tasks to other workers.

4.6.4 Optimizations

In ETLMR, the data sets from different storage systems are prepared into data files. Each file gets a unique address for map readers, such as a URL, file path, or distributed file path. In many cases, however, a data source may be a database or an indexed file structure such that we can make use of its indices for efficient filtering, i.e., instead of returning all the data, only the needed columns or subsets of the data are selected. If the data sets are pre-split, such as several data files from heterogeneous systems, the split parts are directly processed and combined in MapReduce. Different map readers are implemented in ETLMR for reading data from different storage systems, such as the DBMS reader supporting user-defined SQL statements and text file reader. If the data sets are not split, such as a big data file, Dean and Ghemwat [21] suggest utilizing many MapReduce processes to run complete passes over the data sets, and process the subsets of the data. Accordingly, we implement such a map reader (see Program 2). It supports reading data from a single data source, but does not require the data sets to be split before being processed. In addition, we implement the offline dimension scheme by using the Python `shelve` package [70] in which main-memory database systems, such as `bsdbdb`, can be configured to persist dimension data. At run-time, the dimension data is fully or partially kept in main memory such that the *lookup* operation can be done efficiently.

Program 2 Map reader function

```
def map_reader(content, bkey, thispartition=this_partition()):
  while True:
    line = content.next()
    if not line:
        break
    if (hash(line[bkey])%Task.num_partitions)==thispartition:
        yield line
```

4.7 Performance Study

In this section, we present the performance improvements achieved by the proposed methods. Further, we evaluate the system scalability on various sizes of tasks and data sets and compare with other business intelligence tools using MapReduce.

4.7.1 Experimental Setup

All experiments are conducted on a cluster of 6 nodes connected through a gigabit switch and each having an Intel(R) Xeon(R) CPU X3220 2.4GHz with 4 cores, 4 GB RAM, and a SATA hard disk (350 GB, 3 GB/s, 16 MB Cache and 7200 RPM). All nodes are running the Linux 2.6.32 kernel with Disco 0.2.4, Python 2.6, and ETLMR installed. The GlusterFS DFS is set up for the cluster. PostgreSQL 8.3 is used for the DW DBMS and is installed on one of the nodes. One node serves as the master and the others as the workers. Each worker runs 4 parallel map/reduce tasks, i.e., in total 20 parallel tasks run. The time for bulk loading is not measured as the way data is bulk loaded into a database is an implementation choice which is independent of and outside the control of the ETL framework. To include the time for bulk loading would thus clutter the results. We note that bulk loading can be parallelized using off-the-shelf functionality.

4.7.2 Test Data

We continue to use the running example. We use a data generator to generate the test data for each experiment. In line with Jean and Ghemawat's assumption that MapReduce usually operates on numerous small files rather than a single, large, merged file [21], the test data sets are pre-partitioned and saved into a set of files in DFS. These files provide the input for the dimension and fact processing phases. We generate two data sets, *bigdim* and *smalldim* which differ in the size of the *page* dimension. In particular, 80 GB *bigdim* data results in 10.6 GB fact data (193,961,068 rows) and 6.2 GB *page* dimension data (13,918,502 rows) in the DW while 80 GB *smalldim* data results in 12.2 GB (222,253,124 rows) fact data and 54 MB *page* dimension data (193,460 rows) in the DW. Both data sets produce 32 KB *test* (1,000 rows) and 16 KB *date* dimension data (1,254 rows).

4.7.3 Scalability of the Proposed Processing Methods

In this experiment, we compare the scalability and performance of the different ETLMR processing methods. We use a fixed-size *bigdim* data set (20 GB), scale the number of parallel tasks from 4 to 20, and measure the total elapsed time from start to finish. The results for a snowflake schema and a star schema are shown in Figure 4.12 and Figure 4.13, respectively. The graphs show the *speedup*, computed by $T_{4,odot,snowflake}/T_n$ where $T_{4,odot,snowflake}$ is the processing time for *ODOT* using 4 tasks in a snowflake schema and T_n is the processing time when using n tasks for the given processing method.

We see that the overall time used for the star schema is less than for the snowflake schema. This is because the snowflake schema has dimension dependencies and hierarchies which require more (level-wise) processing. We also see that the offline hybrid scales the best and achieves almost linear speedup. The ODAT in Figure 4.13 behaves similarly. This is because the dimensions and facts in offline hybrid and ODAT are processed by all tasks which results in good balancing and scalability. In comparison, ODOT, offline ODOT, level-wise, and hierarchy-wise do not scale as well as ODAT and hybrid since only a limited number of tasks are utilized to process dimensions (a dimension is only processed in a single task). The offline dimension scheme variants outperform the corresponding online ones, e.g., offline ODOT vs. ODOT. This is caused by 1) using a high performance storage system to save dimensions on all nodes and provide in-memory lookup; 2) The data-intensive dimension, *pagedim*, is partitioned into smaller chunks which also benefits the lookups; 3) Unlike the online dimension scheme, the offline dimension scheme does not communicate directly with the DW and this reduces the communication cost considerably. Finally, the results show the relative efficiency for the optimized methods which are much faster than the baseline ODOT.

4.7.4 System Scalability

In this experiment, we evaluate the scalability of ETLMR by varying the number of tasks and the size of the data sets. We select the hybrid processing method, use the offline dimension scheme, and conduct the testing on a star schema, as this method not only can process data among all the tasks (unlike ODOT in which only a limited number of tasks are used), but also showed the best scalability in the previous experiment. In the dimension processing phase, the mappers are responsible for processing the data-intensive dimension *pagedim* while the reducers are responsible for the other

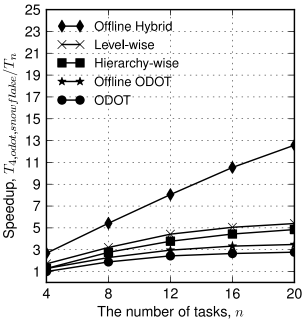


Figure 4.12: Load for snowflake schema, 20 GB

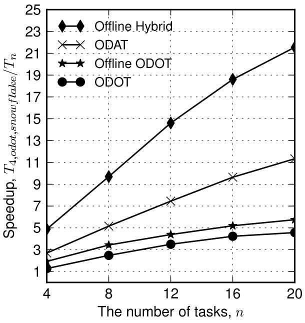


Figure 4.13: Load for star schema, 20 GB

two dimensions, *datedim* and *testdim*, each using only a single reducer. In the fact processing phase, no reducer is used as no aggregation operations are required.

We first do two tests to get comparison baselines by using one task (named *1-task ETLMR*) and (plain, non-MapReduce) *pygrametl*, respectively. Here, pygrametl also employs 2-phase processing, i.e., the dimension processing is done before the fact processing. The tests are done on the same machine with a single CPU (all cores but one are disabled). The tests process 80 GB *bigdim* data. We compute the speedups by using T_1/T_n where T_1 represents the elapsed time for 1-task ETLMR or for pygrametl, and T_n the time for ETLMR using n tasks. Figure 4.14 shows that ETLMR achieves a nearly linear speedup in the number of tasks when compared to 1-task ETLMR (the line on the top). When compared to pygrametl, ETLMR has a nearly linear speedup (the lower line) as well, but the speedup is a little lower. This is because the baseline, 1-task ETLMR, has a greater value due to the overhead from the MapReduce framework.

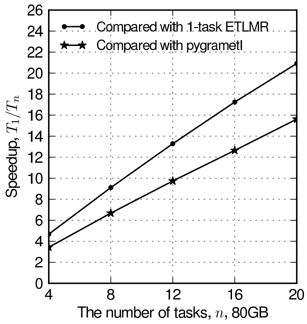


Figure 4.14: Speedup with increasing tasks, 80 GB

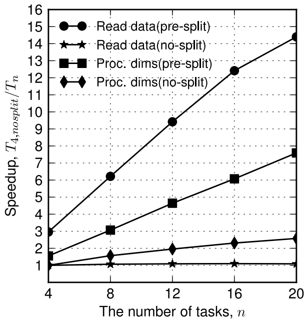


Figure 4.15: Speedup of pre-split and no split, 20 GB

Table 4.2: Execution time distribution, 80 GB (min.)

Testing data	Phase	Task Num	Part. Input	Map func.	Part.	Comb.	Red. func.	Others	Total
bigdim data (results in 10.6GB facts)	dims	4	47.43	178.97	8.56	24.57	1.32	0.1	260.95
		8	25.58	90.98	4.84	12.97	1.18	0.1	135.65
		12	17.21	60.86	3.24	8.57	1.41	0.1	91.39
		16	12.65	47.38	2.50	6.54	1.56	0.1	70.73
		20	10.19	36.41	1.99	5.21	1.32	0.1	55.22
	facts	4	47.20	183.24	0.0	0.0	0.0	0.1	230.44
		8	24.32	92.48	0.0	0.0	0.0	0.1	116.80
		12	16.13	65.50	0.0	0.0	0.0	0.1	81.63
		16	12.12	51.40	0.0	0.0	0.0	0.1	63.52
		20	9.74	40.92	0.0	0.0	0.0	0.1	50.66
smalldim data (results in 12.2GB facts)	facts	4	49.85	211.20	0.0	0.0	0.0	0.1	261.15
		8	25.23	106.20	0.0	0.0	0.0	0.1	131.53
		12	17.05	71.21	0.0	0.0	0.0	0.1	88.36
		16	12.70	53.23	0.0	0.0	0.0	0.1	66.03
		20	10.04	42.44	0.0	0.0	0.0	0.1	52.58

To learn more about the details of the speedup, we break down the execution time of the slowest task by reference to the MapReduce steps when using the two data sets (see Table 4.2). As the time for dimension processing is very small for *smalldim* data, e.g., 1.5 min for 4 tasks and less than 1 min for the others, only its fact processing time is shown. When the *bigdim* data is used, we can see that partitioning input data, map, partitioning map output (dims), and combination (dims) dominate the execution. More specifically, partitioning input data and map (see the *Part.Input* and *Map func.* columns) achieve a nearly linear speedup in the two phases. In the dimension processing, the map output is partitioned and combined for the two dimensions, *datedim* and *testdim*. Also here, we see a nearly linear speedup (see the *Part.* and *Comb.* columns). As the combined data of each is only processed by a single reducer, the time spent on reducing is proportional to the size of data.

However, the time becomes very small since the data has been merged in combiners (see *Red. func.* column). The cost of post-fixing after dimension processing is not listed in the table since it is not required in this case where a global key generator is employed to create dimension IDs and the input data is partitioned by the business key of the SCD *pagedim* (see section 4.4.4).

In the fact processing, the reduce function needs no execution time as there is no reducer. The time for all the other parts, including map and reduce initialization, map output partitioning, writing and reading intermediate files, and network traffic, is relatively small, but it does not necessarily decrease linearly when more tasks are added (*Others* column). To summarize (see *Total* column), ETLMR achieves a nearly linear speedup when the parallelism is scaled up, i.e., the execution time of 8 tasks is nearly half that of 4 tasks, and the execution time of 16 tasks is nearly half that of 8 tasks.

Table 4.2 shows some overhead in the data reading (see *Part. input* column) which uses nearly 20% of the total time. We now compare the two approaches for reading data, namely when it is split into several smaller files ("pre-split") and when it is available from a single big file ("no-split"). We use *bigdim* data set and only process dimensions. We express the performance improvement by the speedup, $T_{4,\textit{no-split}}/T_n$, where $T_{4,\textit{no-split}}$ is the time taken to read data and process dimensions on the no-split data using 4 tasks (on 1 node), and T_n is the time when using n tasks. As illustrated in Figure 4.15, for no-split, the time taken to read data remains constant eventhough there are more tasks as all read the same data sets. When the data is pre-split, the time taken to read data and process dimensions scales much better than for no-split since a smaller sized data set is processed by each task. In addition, pre-split is inherently faster – by a factor of 3 – than no-split. The slight sub-linear scaling is seen because the system management overhead (e.g., the time spent on communication, adjusting, and maintaining the overall system) increases with the growing number of tasks. However, we can conclude that pre-split is by far the best option.

We now proceed to another experiment where we for a given number of tasks size up the data sets from 20 to 80 GB and measure the elapsed processing time. Figure 4.16 and Figure 4.17 show the results for the *bigdim* and *smalldim* data sets, respectively. It can be seen that ETLMR scales linearly in the size of the data sets.

4.7.5 Comparison with Other Data Warehousing Tools

There are some MapReduce data warehousing tools available, including Hive [86, 87], Pig [57] and Pentaho Data Integration (PDI) [62]. We compare ETLMR with each of them in the following.

Pig and Hive: They both offer data storage on the Hadoop distributed file system (HDFS) and the scripting languages which have some limited ETL abilities. Unlike Hive and Pig, ETLMR does not have its own data storage (note that the offline di-

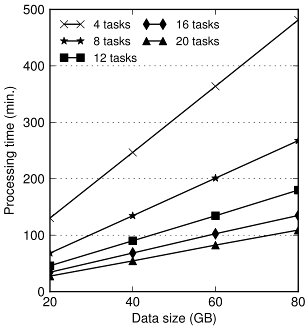


Figure 4.16: Scale up the size of bigdim data

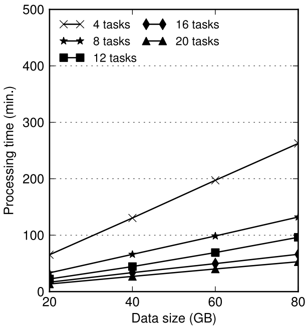


Figure 4.17: Scale up the size of smalldim data

mension store is only for speedup purpose), but is an ETL tool suitable for processing large scale data in parallel. Hive and Pig share large similarity, such as using Hadoop MapReduce, using HDFS as their data storage, integrating a command line user interface, having a scripting language, being able to do some ETL data analysis, and others. Table 4.3 summarizes the comparison of the ETL features for all.

First, each system has a user interface. Hive provides an SQL-like language HiveQL and a shell, Pig provides a scripting language Pig Latin and a shell, and ETLMR provides a configuration file to declare dimensions, facts, UDFs, and other run-time parameters. Unlike Hive and Pig which require users to write data processing scripts explicitly, ETLMR is intrinsically an ETL tool which implements ETL process within the framework. The advantage is that users do not have to learn the details of each ETL step, and are able to craft a parallel ETL program even without much ETL knowledge. Second, each system supports UDFs. In Hive and Pig, an external function or user customized code for a specific task can be implemented as a UDF, and integrated into their own language, e.g., functions for serialization/deserialization data. In ETLMR, UDFs are a number of *rowhandler*s (see Section 4.4 and B.4) integrated into *map* and *reduce* functions. These UDFs are defined for data filtering, transformation, extraction, name mapping. ETLMR also provides other ETL primitive constructs, such as hash join or merge join between data sources, and the aggregation of facts by plugable function, i.e., used as the reduce function (see Section 4.5). In contrast, Hive and Pig achieve the functionality of ETL constructs through a sequence of user-written statements, which are later translated into execution plans, and executed on Hadoop. Third, as ETLMR is a specialized tool developed for fast implementation of parallel ETLs, it explicitly supports the ETLs for processing different schemas, including star schema, snowflake schema and SCDs, and very large dimensions. Therefore, the implementation of a parallel ETL program

Table 4.3: The comparison of ETL features

Feature	ETLMR	HIVE	PIG
User Interface	Configuration file	Shell/HiveQL/Web JDBC/ODBC	Shell/Pig Latin
Pre-knowledge of ETL	Low	High	High
User Defined Functions (UDF)	Yes	Yes	Yes
Filter/Aggregation /Join	Yes	Yes	Yes
Star Schema	Yes (explicit)	By handcode (implicit)	By handcode (implicit)
Snowflake Schema	Yes (explicit)	By handcode (implicit)	By handcode (implicit)
Slowly Changing Dimension (SCD)	Yes (explicit)	No	No
ETL details exposed to users	Transparent	Fine-level	Fine-level

is very concise and intuitive for these schemas, i.e., only fact tables, their referenced dimensions and *rowhandlers* if necessary must be declared in the configuration file. Although Hive and Pig both are able to process star and snowflake schemas technically, implementing an ETL, even the most simple star schema, is not a trivial task as users have to dissect the ETL, write the processing statements for each ETL step, implement UDFs, and do numerous testing to make them correct. Moreover, Hive and Pig do not support the UPDATE operation, which is required for processing SCDs, i.e., update the valid date or/and version of an SCD. Fourth, ETLMR is an alternative to traditional ETL tools but offer better scalability. In contrast, Hive and Pig are obviously not optimal for the situation, where an external DW is used.

In order to make the comparison more intuitive, we create the ETL programs of processing the snowflake schema for the running example (see Figure 4.5) using each of the tools. The scripts for ETLMR, HiveQL and Pig-Latin are shown in Program 3, 4 and 5, respectively. UDFs are not shown in the scripts, but indicated by self-explaining names (starting with *UDF_*).

As shown, the implementation of ETLMR is very concise, which only contains 14 statements for declaring dimension tables, fact tables, data sources and other parameters (a statement may span several lines by "\" in Python). In contrast, the scripts of Hive and Pig are much cumbersome and not intuitive, i.e., containing the finest-level details of ETL. The numbers of the statements are 23 and 40 for Hive and Pig,

Program 3 ETL program for snowflake schema using ETLMR

```
# The configuration file, config.py
# Declare all the dimensions:
datedim = Dimension(name=’date’,key=’dateid’,attributes=[’date’,’day’,’month’,\
          ’year’,’week’,’weekyear’],lookupatts=[’date’])
testdim = Dimension(name=’test’,key=’testid’,defaultidvalue=-1,\
          attributes=[’testname’],lookupatts=[’testname’])
pagedim = SlowlyChangingDimension(name=’page’,key=’pageid’,lookupatts=[’url’],
          attributes=[’url’, ’size’, ’validfrom’,’validto’,’version’,’domain’,\
          ’serverversion’],versionatt=’version’,srcdateatt=’lastmoddate’,\
          fromatt=’validfrom’,toatt=’validto’,srcdateatt=’lastmoddate’)
topdomaindim = Dimension(name=’topdomain’,key=’topdomainid’,\
          attributes=[’topdomain’],lookupatts=[’topdomain’])
domaindim = Dimension( name=’domain’,key=’domainid’, attributes=[’domain’, \
          ’topdomainid’], lookupatts=[’domain’])
serverdim = Dimension(name=’server’,key=’serverid’,attributes=[’server’],\
          lookupatts=[’server’])
serverversiondim = Dimension(name=’serverversion’,key=’serverversionid’,\
          attributes = [’serverversion’,’serverid’], lookupatts = \
          [’serverversion’],refdims=[serverdim])

# Define the snowflaked referencing-ship:
pagesf = [(pagedim, [serverversiondim, domaindim]),(serverversiondim, serverdim),\
         (domaindim, topdomaindim)]

# Declare the facts:
testresultsfact = BulkFactTable(name=’testresults’,keyrefs=[’pageid’,’testid’,\
          ’dateid’], measures=[’errors’], bulkloader=UDF_pgcopy,bulksize=5000000)

# Define the settings of dimensions, including  data source schema, UDFs,
# dimension load order, and the referenced dimensions of fact:
dims ={pagedim: {’srcfields’:(’url’,’serverversion’,’domain’,’size’,\
      ’lastmoddate’),’rowhandlers’:(UDF_extractdomain, UDF_extractserver)},\
      datedim: {’srcfields’:’downloaddate’,),’rowhandlers’:(UDF_explodedate,) },\
      testdim:{’srcfields’:(’test’,), ’rowhandlers’:(, )},}

# Define the processing order of snowflaked dimesions:
loadorder = [(’topdomaindim’, ’serverdim’),(’domaindim’, ’serverversiondim’),\
      (’pagedim’, ’datedim’, ’testdim’)]

# Define the settings of facts:
facts = {testresultsfact:{’refdims’:(pagedim, datedim, testdim),’rowhandlers’:(, )},}

# Define the input data:
inputdata = [’dfs://localhost/TestResults0.csv’, ’dfs://localhost/TestResults1.csv’]

# The main ETLMR program: paralleletl.py
# Start the ETLMR program:
ETLMR.load(’localhost’,inputdata,required_modules=[(’config’,’config.py’),],\
          nr_maps=4,nr_reduces=4)
```

Program 4 ETL program for snowflake schema using HiveQL

```
-- Copy the data sources from local file system to HDFS:
hadoop fs -copyFromLocal /tmp/DownloadLog.csv /user/test;
hadoop fs -copyFromLocal /tmp/TestResults.csv /user/test;

-- Create staging tables for the data sources:
CREATE EXTERNAL TABLE downloadlog(localfile STRING, url STRING, serverversion
STRING, size INT, downloaddate STRING,lastmoddate STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/downloadlog';

CREATE EXTERNAL TABLE testresults(localfile STRING, test STRING, errors INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION
'/user/test/testresults';

-- Load the data into the staging tables:
LOAD DATA INPATH /user/test/input/DownloadLog.csv INTO TABLE downloadlog;
LOAD DATA INPATH /user/test/input/TestResults.csv INTO TABLE testresults;

-- Create all the dimension tables and fact tables:
CREATE EXTERNAL TABLE datedim(dateid INT, downloaddate STRING, day STRING,
month STRING, year STRING, week STRING, weekyear STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/datedim';

CREATE EXTERNAL TABLE testdim(testid INT, testname STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/testdim';

CREATE EXTERNAL TABLE topdomaindim(topdomainid INT, topdomain STRING) ROW FORMAT
DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION
'/user/test/topdomaindim';

CREATE EXTERNAL TABLE domaindim(domainid INT, domain STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/domaindim';

CREATE EXTERNAL TABLE serverdim(serverid INT, server STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/serverdim';

CREATE EXTERNAL TABLE serverversiondim(serverversionid INT, serverversion STRING,
serverid INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE
LOCATION '/user/test/serverversiondim';

CREATE EXTERNAL TABLE pagedim(pageid INT, url STRING, size INT, validfrom STRING,
validto STRING, version INT, domainid INT,serverversionid INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION
'/user/test/pagedim';

CREATE EXTERNAL TABLE testresultsfact(pageid INT, testid INT, dateid INT,
error INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE
LOCATION '/user/test/testresultsfact';

-- Load data into the non-snowflaked dimension tables, testdim and datedim:
INSERT OVERWRITE TABLE  datedim SELECT UDF_getglobalid() AS dateid, downloaddate,
UDF_extractday(downloaddate), UDF_extractmonth(downloaddate),
UDF_extractyear(downloaddate), UDF_extractweek(downloaddate),
UDF_extractweekyear(downloaddate) from downloadlog;

INSERT OVERWRITE TABLE testdim SELECT UDF_getglobalid() AS testid, A.testname FROM
(SELECT DISTINCT test AS testname FROM testresults) A;

-- Load data into the snowflaked dimension tables from leaves to the root:
INSERT OVERWRITE TABLE topdomaindim SELECT UDF_getglobalid() AS topdomainid,
A.topdomain FROM (SELECT DISTINCT UDF_extracttopdomain(url) FROM downloadlog) A;
```

```
INSERT OVERWRITE TABLE domaindim SELECT UDF_getglobalid() AS domainid, A.domain,
B.topdomainid FROM (SELECT DISTINCT UDF_extractdomain(url) AS domain,
UDF_extracttopdomain(url) AS topdomain FROM downloadlog) A JOIN topdomaindim B
ON (A.topdomain=B.topdomain);

INSERT OVERWRITE TABLE serverdim SELECT UDF_getglobalid() AS serverid, A.server
FROM (SELECT DISTINCT UDF_extractserver(serverversion) AS server FROM downloadlog) A;

INSERT OVERWRITE TABLE serverversiondim SELECT UDF_getglobalid() AS serverversionid,
A. serverversion, B.serverid FROM (SELECT DISTINCT  serverversion,
UDF_extractserver(serverversion) as server FROM downloadlog) A JOIN serverdim B
ON (A.server=B.server);

INSERT OVERWRITE TABLE pagedim SELECT UDF_getglobalid() AS pageid, A.url, A.size,
A.lastmoddate, B.domainid, C.serverversionid FROM (SELECT url, size, lastmoddate,
UDF_extractdomain(uri) AS domain, serverversion FROM downloadlog) A JOIN domaindim B
ON (A.domain=B.domain) JOIN serverversiondim C JOIN (A.serverversion=C.serverversion);

CREATE EXTERNAL TABLE pagedim_tmp(pageid INT, url STRING, size INT, lastmoddate
STRING, domainid INT, serverversionid INT) ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/pagedim_tmp';

-- Load data into the fact table, testresultstact:
INSERT OVERWRITE TABLE testresultsfact SELECT C.pageid, E.testid, D.dateid,
B.errors FROM downloadlog A JOIN testresults B ON (A.localfile=B.localfile)
JOIN pagedim C ON (A.url=C.url) JOIN datedim D ON
(A.downloaddate=D.downloaddate) JOIN testdim E ON (B.test=E.testname);
```

Program 5 ETL program for snowflake schema using Pig-Latin

```
-- Copy the data from local file system to HDFS:
hadoop fs -copyFromLocal /tmp/DownloadLog.csv /user/test;
hadoop fs -copyFromLocal /tmp/TestResults.csv /user/test;

-- Load the data into in PIG:
downloadlog = LOAD 'DownloadLog.csv' USING PigStorage('\t')
              AS (localfile, url, serverversion,size,downloaddate,lastmoddate);
testresults = LOAD 'TestResults.csv' USING PigStorage('\t')
              AS (localfile, test, errors);

-- Load the dimension table, testdim:
testers = FOREACH testresults GENERATE test AS testname;
distincttestname = DISTINCT testers;
testdim = FOREACH distincttestname GENERATE UDF_getglobalid()
          AS testid, testname;
STORE testdim INTO '/tmp/testdim' USING PigStorage();

-- Load the dimension table, datedim:
downloadates = FOREACH downloadlog GENERATE downloaddate;
distinctdownloadates = DISTINCT downloadates;

datedim = FOREACH distinctdownloadates GENERATE UDF_getglobalid()
AS dateid, downloaddate, UDF_extractday(downloaddate) AS day,
UDF_extractmonth(downloaddate) AS month, UDF_extractyear(downloaddate) AS year,
UDF_extractweek(downloaddate) AS week, UDF_extractweekyear(downloaddate)
AS weekyear;
```

```
STORE datedim INTO ’/tmp/datedim’ USING PigStorage();

-- Load the sknowflaked dimension tables:
urls = FOREACH downloadlog GENERATE url;

serverversions = FOREACH downloadlog GENERATE serverversion;

domains =  FOREACH downloadlog GENERATE UDF_extractdomain(url) AS domain;

distinctdomains = DISTINCT domains;

topdomains = FOREACH distinctdomains GENERATE UDF_extracttopdomain(domain)
             AS topdomain;

distincttopdomains = DISTINCT topdomains;

topdomaindim = FOREACH distincttopdomains GENERATE UDF_getglobalid()
               AS topdomainid, topdomain;
STORE topdomaindim INTO ’/tmp/topdomaindim’ USING PigStorage();

ndomains = FOREACH distinctdomains GENERATE domain AS domain,
           UDF_extracttopdomain(domain) AS topdomain;

ndomainjoin = JOIN ndomains BY topdomain, topdomaindim BY topdomain;

domaindim = FOREACH ndomainjoin GENERATE UDF_getglobalid()
            AS domainid, domain, topdomainid;
STORE domaindim INTO ’/tmp/domaindim’ USING PigStorage();

distinctserverversions = DISTINCT serverversions;

nserverversions = FOREACH distinctserverversions GENERATE serverversion
                 AS serverversion, UDF_extractserver(serverversion) AS server;

servers = FOREACH nserverversions GENERATE server AS server;
distinctservers = DISTINCT servers;

serverdim = FOREACH distinctservers GENERATE UDF_getglobalid() AS serverid, server;
STORE serverdim INTO ’/tmp/serverdim’ USING PigStorage();

nserverversionjoin = JOIN nserverversions BY server, serverdim BY server;

serverversiondim = FOREACH nserverversionjoin GENERATE UDF_getglobalid()
                   AS serverversionid, serverversion, serverid;

STORE serverversiondim INTO ’/tmp/serverversiondim’ USING PigStorage();

joindomservversion = JOIN (JOIN downloadlog BY UDF_extractdomain(url),
                     domaindim by domain) BY serverversion, serverversiondim
                     BY serverversion;

pagedim = FOREACH joindomservversion GENERATE UDF_getglobalid() AS pageid,
          url, size, lastmoddate, serverversionid, domainid;

STORE pagedim INTO ’/tmp/pagedim’ USING PigStorage();
```

```
-- Load the fact tables:
testresults = JOIN downloadlog BY localfile, testresults BY localfile;

joinpagedim = JOIN testresults BY url, pagedim BY url;

joindatedim = JOIN joinpagedim BY downloaddate, datedim BY downloaddate;

jointestdim = JOIN joindatedim BY test, testdim BY testname;

testresultsfact = FOREACH jointestdim GENERATE dateid, pageid, testid, errors;

STORE testresultsfact INTO '/tmp/testresultsfact' USING PigStorage();
```

respectively (each statement ends with ";"). In addition, during the implementation although we have a clear picture of the ETL processing for this schema, we still spent several hours to write the scripts (the time of implementing UDFs is not included), and test each step, however, it is very efficient to script using ETLMR.

Pentaho Data Integration (PDI): PDI is an ETL tool and provides Hadoop support in its 4.1 GA version. However, there are still many limitations with this version. For example, it only allows to set a limited number of parameters in the job executor, customized combiner and mapper-only jobs are not supported, and the transformation components are not fully supported in Hadoop. We only succeeded in making an ETL flow for the simplest star schema, but still with some compromises. For example, a workaround is employed to load the processed dimension data into the DW as PDI's *table output* component repeatedly opens and closes database connections in Hadoop such that the performance suffers.

In the following, we compare how PDI and ETLMR perform when they process the star schema (with *page* as a normal dimension, not an SCD) of the running example. To make the comparison neutral, the time for loading the data into the DW or the HDFS is not measured, and the dimension lookup cache is enabled in PDI to achieve a similar effect of ETLMR using offline dimensions. Hadoop is configured to run 4 parallel task trackers in maximum on each node, and scaled by adding nodes horizontally. The task tracker JVM option is set to be -Xmx256M while the other settings are left to the default.

Table 4.4 shows the time spent on processing 80 GB *smalldim* data when scaling up the number of tasks. As shown, ETLMR is significantly faster than PDI for Hadoop in processing the data. Several reasons are found for the differences. First, compared with ETLMR, the PDI job has one more step (the reducer) in the fact processing as its job executor does not support a mapper-only job. Second, by default the data in Hadoop is split which results in many tasks, i.e., 1192 tasks for the fact data. Thus, longer initialization time is observed. Further, some transformation components are observed to run with low efficiency in Hadoop, e.g., the components to remove duplicate rows and to apply JavaScript.

Table 4.4: Time for processing star schema (no SCD), 80 GB *smalldim* data set, (min.)

Tasks	4	8	12	16	20
ETLMR	246.7	124.4	83.1	63.8	46.6
PDI	975.2	469.7	317.8	232.5	199.7

4.8 Related Work

Besides MapReduce, other parallel programming models also exist. Multi-threading is an alternative and it is possible for an ETL framework to provide abstractions that make it relatively easy for the ETL developer to use several threads. Multi-threading is, however, only useful when a solution is scaled up on a SMP machine and not when scaled out to several machines in a cluster. For very large data sets it is necessary to scale out and the MapReduce framework does this elegantly and provides support for the "tedious" details. Another, alternative is Message Passing Interface (MPI), but again the MapReduce framework makes the developer's work easier by its abstractions and automatic handling of crashes, stragglers working unusually slowly, etc. In recent years, other massively parallel data processing systems have also been proposed. These include Clustera [23], Dryad [38], and Percolator [61]. Clustera and Dryad are systems that support many kinds of tasks ranging from general cluster computations to data management with parallel SQL queries. Their different approaches to this are interesting and give promising results, but the tools are still only available as academic prototypes and are not available to the general public for commercial use. In comparison, MapReduce so far has a far more wide-spread use. It is supported by many different frameworks and is also available as a service, e.g., from Amazon. Percolator was developed by Google motivated by the fact that web indexing with MapReduce requires a re-run on the entire data set to handle updates. Unlike this situation, Percolator supports incremental updates. It does so by letting the user define "observers" (code) that execute when certain data in Google's BigTable is updated. Further, Percolator adds transnational support. It is, however, closely related to Google's BigTable which makes it less general for ETL purposes. Further, it is not publicly available. MapReduce, on the other hand, can be used with several types of input data and many free implementations exist.

Though MapReduce is a framework well suited for large-scale data processing on clustered computers, it has been criticized for being too low-level, rigid, hard to maintain and reuse [57, 86]. In recent years, an increasing number of parallel data processing systems and languages built on the top of MapReduce have appeared. For example, besides Hive and Pig (discussed in Section 4.7.5), Chaiken et al. present the SQL-like language SCOPE [19] on top of Microsoft's Cosmos MapReduce and distributed file system. Friedman et al. introduce SQL/MapReduce [31], a user-defined function (UDF) framework for parallel computation of procedural functions on massively-parallel RDBMSs. These systems or languages vary with respect to how they are implemented and what functionality they provide, but overall they give good improvements to MapReduce such as high-level languages, user interfaces, schemas, and catalogs. They process data by using query languages, or UDFs embedded in the query languages, and execute them on MapReduce. However, they do not offer direct constructs for processing star schemas, snowflaked dimensions, and slowly changing dimensions. In contrast, ETLMR runs separate ETL processes on a MapReduce framework to achieve parallelization and ETLMR directly supports common ETL operations for these schemas.

Another well-known distributed computing system is the parallel DBMS which first appeared two decades ago. Today, there are many parallel DBMSs, e.g., Teradata, DB2, Objectivity/DB, Vertica, etc. The principal difference between parallel DBMSs and MapReduce is that parallel DBMSs run long pipe-lined queries instead of small independent tasks as in MapReduce. The database research community has recently compared the two classes of systems. Pavlo et al. [60], and Stonebraker et al. [76] conduct benchmarks and compare the open source MapReduce implementation Hadoop with two parallel DBMSs (a row-based and a column-based) in large-scale data analysis. The results demonstrate that parallel DBMSs are significantly faster than Hadoop, but they diverge in the effort needed to tune the two classes of systems. Dean et al. [21] argue that there are mistaken assumptions about MapReduce in the comparison papers and claim that MapReduce is highly effective and efficient for large-scale fault-tolerance data analysis. They agree that MapReduce excels at complex data analysis, while parallel DBMSs excel at efficient queries on large data sets [76].

In recent years, ETL technologies have started to support parallel processing. Informatica PowerCenter provides a thread-based architecture to execute parallel ETL sessions. Informatica has also released PowerCenter Cloud Edition (PCE) in 2009 which, however, only runs on a specific platform and DBMS. Oracle Warehouse Builder (OWB) supports pipeline processing and multiple processes running in parallel. Microsoft SQL Server Integration Services (SSIS) achieves parallelization by running multiple threads, multiple tasks, or multiple instances of a SSIS package. IBM InfoSphere DataStage offers a process-based parallel architecture. In the thread-based approach, the threads are derived from a single program, and run on a single (expensive) SMP server, while in the process-based approach, ETL processes are replicated to run on clustered MPP or NUMA servers. ETLMR differs from the above by being based on MapReduce with the inherent advantages of multi-platform support, scalability on commodity clustered computers, light-weight operation, fault tolerance, etc. ETLMR is also unique in being able to scale automatically to more nodes (with no changes to the ETL flow itself, only to a configuration parameter) while at the same time providing automatic data synchronization across nodes even for complex structures like snowflaked dimensions and SCDs. We note that the licenses of the commercial ETL packages prevent us from presenting comparative experimental results.

4.9 Conclusion and Future Work

As business intelligence deals with continuously increasing amounts of data, there is an increasing need for ever-faster ETL processing. In this chapter, we have presented ETLMR which builds on MapReduce to parallelize ETL processes on commodity computers. ETLMR contains a number of novel contributions. It supports high-level ETL-specific dimensional constructs for processing both star schemas and snowflake schemas, SCDs, and data-intensive dimensions. Due to its use of MapReduce, it can automatically scale to more nodes (without modifications to the ETL flow) while it at the same time provides automatic data synchronization across nodes (even for complex dimension structures like snowflakes and SCDs). Apart from scalability, MapReduce also gives ETLMR a high fault-tolerance. Further, ETLMR is open source, light-weight, and easy to use with a single configuration file setting all run-time parameters. The results of extensive experiments show that ETLMR has good scalability and compares favourably with other MapReduce data warehousing tools.

ETLMR comprises two data processing phases, dimension and fact processing. For dimension processing, this chapter proposed a number of dimension management schemes and processing methods in order to achieve good and load balancing. The online dimension scheme directly interacts with the target DW and employs several dimension specific methods to process data, including *ODOT*, *ODAT*, and *level-wise* and *hierarchy-wise* processing for snowflaked dimensions. The offline dimension scheme employs high-performance storage systems to store dimensions distributedly on each node. The methods, *ODOT* and *hybrid* allow better scalability and performance. In the fact processing phase, bulk-load is used to improve the loading performance.

Currently, we have integrated ETLMR with the MapReduce framework Disco. In the future, we intend to port ETLMR to Hadoop and explore a wider variety of data storage options. In addition, we intend to implement dynamic partitioning which automatically adjusts the parallel execution in response to additions/removals of nodes from the cluster, and automatic load balancing which dynamically distributes jobs across available nodes based on CPU usage, memory, capacity and job size through automatic node detection and algorithm resource allocation.

Chapter 5

CloudETL: Scalable Dimensional ETL for Hadoop and Hive

Extract-Transform-Load (ETL) programs process data from sources into data warehouses (DWs). Due to the rapid growth of data volumes, there is an increasing demand for systems that can scale on demand. Recently, much attention has been given to *MapReduce* which is a framework for highly parallel handling of massive data sets in cloud environments. The MapReduce-based *Hive* has been proposed as a DBMS-like system for DWs and provides good and scalable analytical features. It is, however, still challenging to do proper dimensional ETL processing with Hive; for example, UPDATEs are not supported which makes handling of slowly changing dimensions (SCDs) very difficult. To remedy this, we here present the cloud-enabled ETL framework *CloudETL*. CloudETL uses the open-source MapReduce implementation *Hadoop* to parallelize the ETL execution and to process data into Hive. The user defines the ETL process by means of high-level constructs and transformations and does not have to worry about the technical details of MapReduce. CloudETL provides built-in support for different dimensional concepts, including star schemas, snowflake schemas, and SCDs. In the chapter, we present how CloudETL works. We present different performance optimizations including a purpose-specific data placement policy for Hadoop to *co-locate* data. Further, we present a performance study using realistic data amounts and compare with other cloud-enabled systems. The results show that CloudETL has good scalability and outperforms the dimensional ETL capabilities of Hive both with respect to performance and programmer productivity.

5.1 Introduction

In data warehousing, data from different source systems is processed into a central DW by an Extract–Transform–Load (ETL) process in a periodic manner. Traditionally, the DW is implemented in a relational database where the data is stored in fact tables and dimension tables which form a star schema or snowflake schema [43]. Many enterprises now collect and analyze tens or hundreds of gigabytes data each day. The data warehousing technologies are thus faced with the challenge of handling the growing data volumes in little time. However, many existing data warehousing systems already take hours, or even days to finish loading the daily data and will be to slow with larger data volumes.

There is thus an increasing need for a new data warehousing architecture that can achieve better scalability and efficiency. In recent years, with the emergence of the cloud computing technologies, such as the MapReduce paradigm [22], many enterprises have shifted away from deploying their analytical systems on high-end proprietary machines and instead moved towards cheaper, commodity machines [2]. The MapReduce paradigm provides cost-effective scalability for large-scale data sets and handles fault tolerance very well even on hundreds or thousands of machines. The recent system Hive [86] uses the open-source MapReduce implementation Hadoop [78] and can be used for scalable data warehousing. Hive stores data in the Hadoop Distributed File System (HDFS), and presents the data by logical *tables*. The data in the tables is queried by user-written (SQL-like) *HiveQL* scripts which are translated into MapReduce jobs to process the data. However, Hive only has limited dimensional ETL capabilities and it is not straightforward to use in an ETL process. It is more like a DBMS and less like an ETL tool. For example, Hive lacks support for high-level ETL-specific constructs including those for looking up a dimension member or, if not found, updating the dimension table (or even more dimension tables in a snowflake schema). There is also no specialized handling for SCDs. Writing HiveQL scripts for such processing is cumbersome and requires a lot of programming efforts [50]. In addition, Hive also lacks support for UPDATEs which makes handling of SCDs even more complicated when time-valued attributes are used to track the changes of dimension values.

In this chapter, we present *CloudETL* which is a scalable dimensional ETL framework for Hive running on Hadoop. CloudETL supports the aforementioned ETL constructs, among others, for different DW schemas. CloudETL coexists with Hive for scalable data warehousing and aims at making it easier and faster to create scalable and efficient ETL processes that load DWs in Hive. CloudETL allows ETL programmers to easily translate a high-level ETL design into actual MapReduce jobs on Hadoop by only using high-level constructs in a Java program and without knowing MapReduce concepts. We provide a library of commonly used ETL constructs as building blocks. All the complexity associated with parallel programming is transparent to user, and the programmer only needs to think about how to apply the constructs to a given DW schema leading to high programmer productivity.

The contributions of this chapter are listed in the following: First, we present a novel and scalable dimensional ETL framework which provides direct support for high-level ETL constructs, including handling of star schemas, snowflake schemas, and SCDs. Second, we present a method for processing SCDs enabling update capabilities in a cloud environment. Third, we present how to process big dimensions efficiently through purpose-specific *co-location* of files on the distributed file system, and we present *in-map updates* to optimize dimension processing. Fourth, we present lookup indices and multi-way lookups for processing fact data efficiently in parallel. Fifth, we provide a set of high-level transformation operators to simplify the implementation of a parallel, dimensional ETL program for a MapReduce environment.

The rest of the chapter is structured as follows. In Section 5.2, we give an overview of CloudETL and its components. In Section 5.3, we introduce a running example. In Section 5.4, we detail dimension processing including the parallelization for multiple dimension tables, co-locating data and the updates for SCDs. In Section 5.5, we present the approach for processing facts. In Section 5.6, we describe the implementation of CloudETL. In Section 5.7, we study the performance CloudETL and compare with other similar works. In Section 5.8, we present the related work. Finally, in Section 5.9, we summarize the chapter and discuss the future research directions.

5.2 System Overview

CloudETL employs Hadoop as the ETL execution platform and Hive as the warehouse system (see Figure 5.1). CloudETL has a number of components, including the application programming interfaces (APIs) used by ETL programs, a sequence service, a metastore, ETL transformers, and a job planner. The sequence service is used by the distributed jobs to generate unique, sequential numbers for dimension keys. The ETL transformers define the transformations for reading source data, cleansing data, and writing the data into the DW. The job planner is used to plan the execution order of the jobs that will be submitted to Hadoop. The metastore is the system catalog which contains the meta-data about the dimension and fact data stored in the Hive, and the next values of the sequential number generator.

The ETL workflow in CloudETL consists of two sequential steps: dimension processing and fact processing. The source data is assumed to be present in HDFS when the MapReduce (MR) jobs are started (see the left of Figure 5.1). CloudETL allows processing of data into multiple tables within a job. The source data is processed into dimension values or facts by the user-defined transformers which are executed

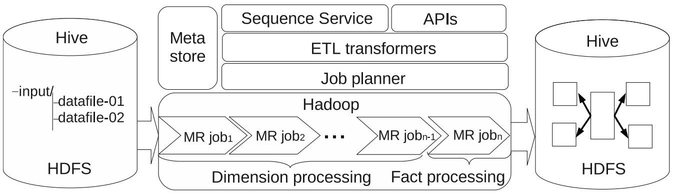


Figure 5.1: CloudETL Architecture

by mappers and reducers. A transformer integrates a number of transformation operators used for processing data, such as filters, data type converters, splitters, lookups (for getting dimension key values), and others. According to the dependencies of the tables to be processed, the CloudETL job planner can automatically plan the jobs and submit the jobs to the Hadoop JobTracker in a sequential order. The jobs for dimension processing are planned to run before the jobs for fact processing (see the middle of Figure 5.1) as processing facts requires looking up referenced primary key values from the dimension tables. For normalized ("snowflaked") dimensions with many dimension tables, there also exists foreign key dependencies. The MR jobs handling the dimension tables run partly sequentially (see Section 5.4 for details). Take the snowflaked dimension tables $T_1, \ldots, T_n$ as an example. Suppose that the tables have foreign key references such that T_1 is referenced by T_2 which is referenced by T_3, etc. Then the execution order of the jobs becomes *MR job*$_1$ followed by *MR job*$_2$, *MR job*$_2$ followed by *MR job*$_3$ and so forth (see the middle of Figure 5.1).

Hive employs HDFS for physical data storage but presents the data in the HDFS directories and files as logical tables. Therefore, when Hive is used, we can easily write data directly into files which then can be used by Hive. The right-most part of Figure 5.1 shows a star schema in Hive which consists of five files in HDFS.

5.3 Running Example

In the following, we will use a running example to show how CloudETL processes data into dimension tables and fact tables. This example is inspired by our work in a previous project [82]. This example considers a DW with data about tests of web pages. The star schema shown in Figure 5.2 consists of a fact table `testresults-fact` with the single measure `errors` telling how many errors were detected on a given version of a web page on a given date. There are three dimension tables, `testdim`, `pagedim`, and `datedim`, which represent tests, web pages, and dates, respectively. The `pagedim` can be normalized to get a partial snowflake schema (we show the snowflake schema in Section 5.4.6). Note that `pagedim` is a "type-2" SCD which tracks changes by having multiple versions of its dimension values.

pagedim is also a data-intensive dimension table which contains many more rows than the other two dimension tables. We use this example, instead of a more common example such as TPC-H [88], because it has an SCD, a data-intensive dimension, and can be represented by both star and snowflake schemas. The chosen example thus allows us to illustrate and test our system more comprehensively.

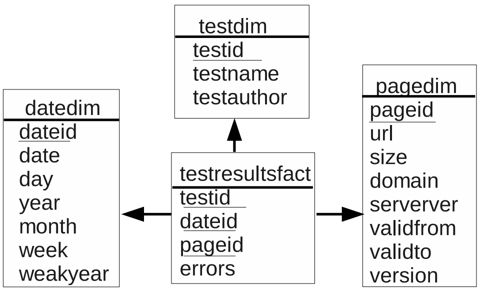


Figure 5.2: Star schema

5.4 Dimension Processing

5.4.1 Challenges and Designs

In conventional ETL processing, many transformation capabilities rely on the underlying DW DBMS to, e.g., generate sequential key values, perform SCD updates, and insert rows into dimension tables and fact tables. However, support for this is not available in Hive and Hadoop and, as mentioned, Hive and other MapReduce-based programs for data analysis do not support the UPDATE operation known from SQL. In addition, a number of limitations make the ETL on Hadoop more challenging. For example, the nodes running the MR jobs share no global state, which makes certain tasks (such as acquiring unique values for a dimension key) more difficult for an ETL process, and Hadoop does not hold schema information for the data to process, which makes data cleansing difficult.

We address these limitations with the following design which is explained further in the following sections: First, a line read from the data source is made into a *record* which migrates from mappers to reducers. A record contains schema information about the line, i.e., the names of attributes and the data types. For example, a record with pagedim data has the schema ⟨url string, size int, moddate date⟩. Example attribute values are ⟨www.dom.com/p0.htm, 10, 2012-02-01⟩. Second, to currectly maintain an SCD, we union incremental data with the existing dimension data that has already been processed into a dimension table, and replace the attribute values for both when necessary. Third, we create a global sequential number generation service and add a metastore to maintain the global state of the ETL on Hadoop.

5.4.2 Execution Flow

Figure 5.3 shows the execution flow for dimension processing. For simplicity, we only show the ith reducer which processes a part of the intermediate data output by all mappers. The dimension source data in HDFS is split (by Hadoop) and assigned to the map tasks. The records from a file split are processed by user-defined transformation operators in mappers and reducers. Here, we categorize the transformation operators into two classes: *non-aggregating* and *aggregating*. The non-aggregating operators are those that can be performed by only seeing a single record (e.g., field projection, filtering, and splitting). They are executed in the mappers. The aggregating operators, on the other hand, operate on many records (e.g., a grouping). They are executed in the reducers. Note that a mapper can process a source with data that will go to different dimensions. In the shuffling, the data is sent to different reducers. The reducer output (corresponding to a dimension table) is written to the HDFS.

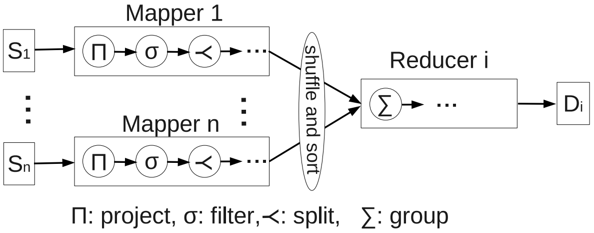


Figure 5.3: Execution flow of dimension processing

We now describe how to process changes by using the `pagedim` dimension as an example. All dimensions are considered to be SCDs, but if there are no changes in the source data, the dimensions will of course not be updated. The main problem of processing SCDs is how to update the special SCD type-2 attribute values (with validity dates and version numbers) in a MapReduce environment. For type-2 SCDs, the validity dates and version numbers are updated by following the original change order in the source data. For example, the end date of a given version is set to the start date of its successor. For type-1 SCDs, the attribute values of a dimension member are overwritten by new values. When doing incremental loading, we also need to update the versions that have already been loaded into Hive. The idea here is to collect different versions of dimension values from both the incremental data and the existing dimension data, and to perform the updates in reducers. Therefore, the execution flow is as follows: First, we do the non-aggregating transformations on the incremental data in mappers, then hash partitioning on the *business keys* (a dimension must have a key that distinguishes its members) of the map output. For example, we partition the map output for pagedim based on the values of the business key, *url*. Therefore, the rows with identical key values are shuffled to the same reducer. To acquire the original change order, we sort the intermediate data by the modification date. For example, moddate of the pagedim source data tells when a given page was changed. If the source data does not include a date, we assume that the line numbers show the changing order (line numbers should be explicitly given in input data if the input is made up of several files). Alternatively, the user can choose that another attribute should be used to get the change order.

In any case, we include the sorting attribute(s) in the key of the map output since Hadoop only supports sorting on keys, but not on values. To decrease the network I/O, we include both the business key and the SCD date (or another sorting attribute) in the key of the map output (this is an application of the *value-to-key* MapReduce design pattern [48]). In addition, we make a task support processing of multiple dimension tables by tagging the key-value pairs with the name of the dimension table to which the data will be written. Therefore, the map output has the format (⟨ *the name of a dimension table, business key, SCD date/line no.* ⟩, ⟨ *the rest of dimension values*⟩) where the key is composite of three attributes.

To make the above more concrete, we show the input and output of map and reduce when processing pagedim (see Table 5.1). We assume that a single dimension value has already been loaded into Hive, and now we perform incremental loading with two new values. The map input consists of both the existing and the new data (see Map input). We discuss an optimization in Section 5.4.5. A new record has the attributes url, size, and moddate which indicate the web page address, the size (may change when the page is modified) and the modification date of a page, respectively. The existing dimension record contains the three additional SCD attributes version, validfrom, and validto. In map, we transform the raw incremental data into dimension values by adding the three additional SCD attributes and the surrogate key id. Then, the mapper writes the records in the described output format. For the existing dimension record, no transformation is needed, but the record is re-structured in line with the map output format. When the map output is shuffled to the reducers, the key-value pairs are grouped by the composite key values and sorted by the validity SCD date in ascending order (see reduce input). In the reducer, unique numbers (acquired from the sequence generation service) are assigned to the key attribute, i.e., id, of the two new dimension values and the values of the SCD attributes are updated, i.e., validto of a version is updated to the starting valid date of the following version (see the validfrom and validto of the different versions for url=www.dom.com/p0.htm). The version number is also updated accordingly. The three records are finally written to HDFS (nil in the `validto` attribute represents that a dimension record is valid till now) (see `reduce output` in Table 5.1).

	Incremental data (key, value)	Existing dimension data (key, value)
Map input	(lineno, < url, size, moddate >) (0, <www.dom.com/p0.htm, 20, 2011-02-01>) (1, <www.dom.com/p1.htm, 10, 2011-01-01>)	(lineno, <id, url, size, version, validfrom, validto>) (0, <0, www.dom.com/p0.htm, 10, 1, 2011-01-01, nil >)
Map output	(<name, url, scddate >, <id, size, version, validfrom, validto>) (<pagedim, www.dom.com/p0.htm, 2011-02-01>, <nil, 20, nil, nil, nil >) (<pagedim, www.dom.com/p1.htm, 2011-01-01>, <nil, 10, nil, nil, nil >) (<pagedim, www.dom.com/p0.htm, 2011-01-01>, < 1, 10, 1, 2011-01-01, nil >)	
Reduce input	(<name, url, scddate >, <id, size, version, validfrom, validto>) (<pagedim, www.dom.com/p0.htm, 2011-01-01>, <1, 10, 1, 2011-01-01, nil >) (<pagedim, www.dom.com/p0.htm, 2011-02-01>, <nil, 20, nil, nil, nil >) (<pagedim, www.dom.com/p1.htm, 2011-01-01>, <nil , 10, nil, nil, nil >)	
Reduce output	(nil, <id, url, size, version, validfrom, validto >) (nil, <1, www.dom.com/p0.htm, 10, 1, 2011-01-01, 2011-02-01>) (nil, <2, www.dom.com/p0.htm, 20, 2, 2011-02-01, nil >) (nil, <3, www.dom.com/p1.htm, 10, 1, 2011-01-01, nil >)	

Table 5.1: MapReduce input and output of type-2 *page* SCDs (nil represents the value is empty)

5.4.3 Algorithm Design

Algorithm 11 shows the map algorithm and Figure 5.4 shows the ETL components in a mapper. In the algorithm, Γ is the set of *transformers* defined by the user. A transformer is an ETL flow for processing data from a source to a dimension table. Formally, a transformer is a pair $\langle \overrightarrow{P}, d\rangle$ where $\overrightarrow{P}$ is an n-tuple $\overrightarrow{P} = \langle P_1, P_2, \ldots, P_n\rangle$ representing a number of *transform pipes* connected one after another. The connected transform pipes process data from a data source into a final dimension table d. The transformers are deserialized when a mapper is initialized (see line 3). The algorithm supports processing of several dimension tables within a mapper; each of them is then processed by a different transformer (see line 5–12). A transform pipe, P_i with $(1 \leq i \leq n)$ is the ETL component integrating a variety of operators for transforming data. It is defined as $P_i = \langle t_{i1}, t_{i2}, \ldots, t_{im}\rangle$ where t_{ij} with $(1 \leq j \leq m)$ represents an ETL transformation operator. The root transform pipe, P_1, defines the schematic information of the data source such as the names of attributes, the data types, and the attributes for sorting of versions (such as a date or line no.). When a record is processed in a mapper, the record goes through the connected transform pipes from the root and is processed by the transformation operators consecutively within each of the transform pipes (see line 6–8). In the algorithm, the transformer for processing existing dimension data only consists of one transform pipe, i.e., Γ only consists of the root. The reason is that the existing dimension data is only read back be able to process SCD attribute values and no other data transformation is required.

Algorithm 12 shows the reduce algorithm (in Section 5.4.5, we present a specialized map-only method for big dimensions). The input is grouped by the composite key values and sorted (by the validity date or line no.) before it is fed to the REDUCE

Algorithm 11 Map

```
class Mapper
    function INITIALIZE( )
        $\Gamma \leftarrow$ DESERIALIZETRANSFORMERS()
    function MAP(offset $o$, Record $r$)
        for all $\langle \overrightarrow{P}, d\rangle \in \Gamma$ do
            for all $P \in \overrightarrow{P}$ do                    ▷ $\overrightarrow{P} = \langle P_1, P_2, ..., P_n\rangle$
                for all $t \in P$ do                                 ▷ $P = \langle t_1, t_2, ..., t_m\rangle$
                    $r \leftarrow t$.PROCESSRECORD$(r)$   ▷ This function might return the special value $\perp$ which
represents that $r$ is filtered out or deleted.
            if $r \neq \perp$ then
                $key \leftarrow$ CREATECOMPOSITEKEY($r$, $d$)
                $value \leftarrow$ CREATEVALUE($r$, $d$)
                EMIT$(key, value)$

```

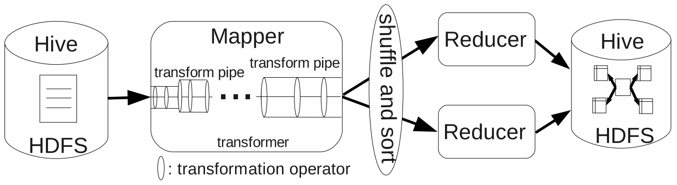


Figure 5.4: Transformers, transform pipes and transformation operators

method. The REDUCE method reads all the dimension data with a particular composite key value (and thus the data also has the same business key value). For type-2 SCDs (see lines 4–13), we keep the dimension values temporarily in a buffer (lines 5–10), assign a sequential number to the key of a new dimension record (line 9), and update the SCD attribute values (line 11), including the validity dates and the version number. The function MAKEDIMENSIONRECORD extracts the dimension's business key from the composite key given to the mapper and combines it with the remaining values. Finally, we write the reduce output with the name of the dimension table as the key, and the dimension data as the value (see line 12). The reduce output format is customized so that the value is written into the directory named by the output key value. For type-1 SCDs (see line 14–21), we overwrite the old values. That is, we only keep the latest version with a given business key value. An important point here is that if r_0 has a primary key value set (here denoted $r_0[id]$), the data in r_0 comes from the existing table and the primary key value should be reused (line 18). If the primary key value is not set in r_0, we are handling an entirely new member and should get a new primary key value.

Algorithm 12 Reduce

```
class Reducer
    function REDUCE(CompositeKey $key$, $values[0...n-1]$)
        $name \leftarrow$ GETNAMEOFDIMENSIONTABLE($key$)
        if DIMTYPE($name$) $= type2SCD$ then
            $L \leftarrow$ new LIST()
            for $i \leftarrow 0, n-1$ do
                $r \leftarrow$ MAKEDIMENSIONRECORD($key, values[i]$)
                if $r[id] = \perp$ then                    ▷ $id$ is the dimension key
                    $r[id] \leftarrow$ GETDIMENSIONID($name$)
                $L$.ADD($r$)
            UPDATESCDATTRIBUTEVALUES($L$)
            for all $r \in L$ do
                EMIT($name, r$)
        else    ▷ It is a type-1 SCD: keep the key value of the first record, but the dimension values of the last record
            $r_0 \leftarrow$ MAKEDIMENSIONRECORD($key, values[0]$)
            $r_{n-1} \leftarrow$ MAKEDIMENSIONRECORD($key, values[n-1]$)
            if $r_0[id] \neq \perp$ then
                $r_{n-1}[id] \leftarrow r_0[id]$
            else
                $r_{n-1}[id] \leftarrow$ GETDIMENSIONID($name$)
            EMIT($name, r_{n-1}$)
```

5.4.4 Pre-update in Mappers

On Hadoop, it is relatively time-consuming to write map output to disk and transfer the intermediate data from mappers to reducers. For type-1 SCDs, we thus do *pre-updates* in mappers to improve the efficiency by shrinking the size of the intermediate data shuffled to the reducers. This is done by only transferring the resulting dimension member (which may have been updated several times) from the mapper to the reducer. On the reduce side, we then do *post-updates* to update the dimension to represent the dimension member correctly.

Algorithm 13 shows how pre-updates for type-1 SCDs are handled. For simplicity, we no longer show the ETL transformation operations in this algorithm. In the map initialization (see line 3), we create a hash map M to cache the mappings of a business key value to the corresponding dimension values. Since the state of M is preserved during the multiple calls to the MAP method, we can use M until the entire map task finishes. In the mapper, the dimension attribute values are always updated to the latest version's if there are any changes (see line 5–9). Here, we should preserve the key value (i.e., the value of id) if the member is already represented, but update the other attribute values (see line 8 and 9). The construction and emission of the composite key-value pairs are deferred to the CLOSE method which is called when the mapper has finished processing a file split.

Algorithm 13 Pre-update in mappers for type-1 SCDs

```
class Mapper
    function INITIALIZE( )
        $M \leftarrow$ new HASHMAP()
    function MAP(offset $o$, Record $r$)
        $k \leftarrow \text{GETBUSINESSKEY}(r)$
        $pre \leftarrow M[k]$
        if $pre \neq \varnothing$and $(pre[scddate] < r[scddate])$ then
            $r[id] \leftarrow pre[id]$                ▷ Preserve $id$ of the existing dimension
        $M[k] \leftarrow r$                           ▷ Update the old attribute values
    function CLOSE( )
        for all $m \in M$ do
            $key \leftarrow \text{CREATECOMPOSITEKEY}(m)$
            $value \leftarrow \text{CREATEVALUE}(m)$
            $\text{EMIT}(key, value)$
```

With the pre-updates, we can decrease the size of the intermediate data transferred over the network, which is particularly useful for data with frequent changes. Of course, we can also do the updates in a combiner. However, doing pre-updates in a combiner would typically be more expensive since we have to transfer the intermediate data from the mapper to the combiner, which involves object creation and destruction, and object serialization and deserialization if the in-memory buffer is not big enough to hold the intermediate data.

5.4.5 Process Big Dimensions

Typically, the size of dimension data is relatively small compared to the fact data and can be efficiently processed by the method we discussed above. This is the case for `datedim` and `testdim` of the running example. However, some dimensions – such as `pagedim` – are very big and have much more data than typical dimensions. In this case, shuffling a large amount of data from mappers to reducers is not efficient. We now present a method that processes data for a big dimension in a map-only job. The method makes use of data locality in HDFS and is based on the general ideas of CoHadoop [29], but is in CloudETL automated and made purpose-specific for dimensional data processing. We illustrate it by the example in Figure 5.5. Consider an incremental load of the `pagedim` dimension and assume that the previous load resulted in three dimension data files, D_1, D_2, and D_3, each of which is the output of a task. The files reside in the three data nodes `node1`, `node2`, and `node3`, respectively (see the left of Figure 5.5). For the incremental load, we assume the incremental data is partitioned on the business key values using the same partitioning function as in the previous load. Suppose that the partitioning has resulted in two partitioned files, S_1 and S_3. D_1 and S_1 have the same hash value on the business key values, and so do D_3 and S_3. When S_1 and S_3 are created in HDFS, *data co-location* is applied to them, i.e., D_1 and S_1 are placed together on a data node and so are D_3 and S_3 (see the right of Figure 5.5). Then, a map-only job is run to process the co-located data on each node locally. In this example, note that no incremental data is co-located with D_2. The existing dimension data in D_2 is not read or updated during the incremental load.

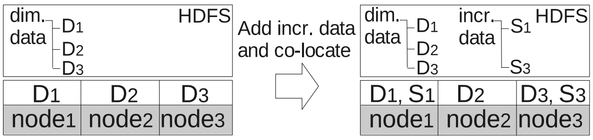


Figure 5.5: Co-locate files on HDFS

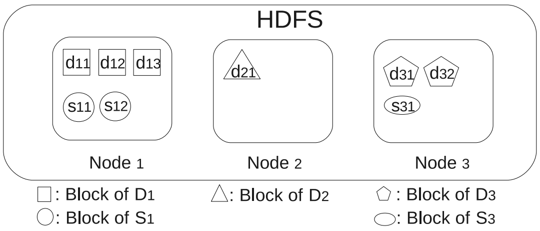


Figure 5.6: Data blocks of co-location (replication factor $= 1$)

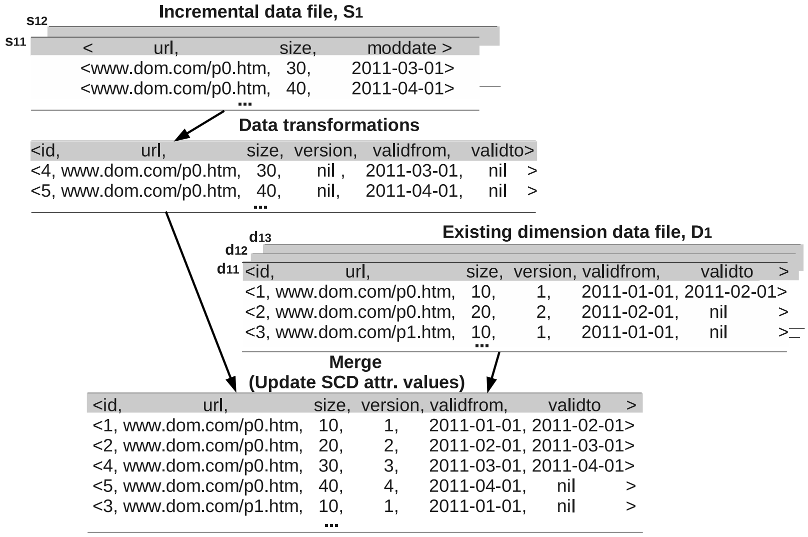
We now present how the *blocks* of the co-located files are handled (see Figure 5.6). For simplicity, we show the blocks when the replication factor is set to 1. As shown, blocks of co-located files are placed on the same data node. For example, the blocks of D_1 (d_{11}, d_{12}, and d_{13}) and the blocks of S_1 (s_{11} and s_{12}) are on `node1`. If the replication factor is different from 1, the block replicas of the co-located files are also co-located on other nodes.

Since Hadoop 0.21.0, users can use their own *block placement policy* by setting the configuration property “dfs.block.replicator .classname”. This does not require re-compiling Hadoop or HDFS. The block placement policy used by CloudETL is shown in Algorithm 14. In CloudETL, we (unlike CoHadoop) co-locate the files based on their names, i.e., the files whose names match the same regular expression are co-located. The regular expressions are defined in the name node configuration file, `core-site.xml`. For example, if we define the regular expression `(.*\.page1)`, the data blocks of the existing dimension data files and the partitioned incremental files with the name extension `.page1` are placed together. When the name node starts up, a hash dictionary, M, is created to hold mappings from a data node to information about the blocks on the node, i.e., the total number of blocks of the files whose names match a regular expression (see line 2–12). When an HDFS

Algorithm 14 Choosing targets in the block placement policy

```
class BlockPlacementPolicy
    function INITIALIZE( )
        $M \leftarrow$ new HASHMAP()
        $regex \leftarrow$ GETFILENAMEREGEXFROMCONFIG()
        $n \leftarrow$ GETREPLICANUMFROMCONFIG()
        $F \leftarrow$ GETFILESFROMNAMESYSTEM($regex$)          ▷ Get the files whose names match $regex$
        for all $f \in F$ do
            $B \leftarrow$ GETBLOCKSFROMBLOCKMANAGER($f$)
            for all $b \in B$ do
                $D \leftarrow$GETDATANODES($b$)
                for all $d \in D$ do
                    $M[d] \leftarrow M[d] + 1$
    function CHOOSETARGETS(String $filename$)
        $D \leftarrow$new COLLECTION()
        if $regex$.MATCHES($filename$) then
            if SIZE($M$) $> 0$ then
                $Q \leftarrow$ GETDATANODES($M$)        ▷ Sort $M$ descendingly by the number of blocks and add data
nodes to the queue $Q$.
                while SIZE($Q$) $> 0$ and SIZE($D$) $< n$ do
                    $d \leftarrow Q$.POP()
                    if ISGOODTARGET($d$) then
                        $D$.ADD($d$)
                        $M[d] \leftarrow M[d] + 1$
                while SIZE($D$) $< n$ do
                    $d \leftarrow$ CHOOSERANDOM()
                    if ISGOODTARGET($d$) then
                        $D$.ADD($d$)
                        $M[d] \leftarrow M[d] + 1$
            else
                $D \leftarrow$ CHOOSETARGETSBYDEFAULTPOLICY($filename$)
                for all $d \in D$ do
                    $M[d] \leftarrow M[d] + 1$
        else
            $D \leftarrow$ CHOOSETARGETSBYDEFAULTPOLICY($filename$)
        return $D$
```

client writes a block, it first asks the name node to choose target data nodes for the block and its replicas. The name node checks if the name of the file matches the regular expression. If the name matches, the name node chooses the targets based on the statistics in M. If M is empty, the client is writing the first block of co-located data and the name node chooses the targets by the default policy and updates M (see line 28–31). If M is non-empty, the name node chooses the targets based on the existing co-located data blocks in HDFS. The name node selects a data node for each replica to store. As in CoHadoop, the data nodes with the highest number of blocks and with enough space are selected (this is done by sorting M by values, adding the nodes into a queue Q, and checking the nodes in an descending order, see line 17–22). When all the nodes in Q have been checked, but fewer data nodes than needed have been selected, the name node chooses the remaining nodes randomly. Each of the chosen nodes is tested to see if it meets the selection criteria and has sufficient space (see line 23–27). If the file name does not match the regular expression, the file should not be co-located with anything and the name node uses the default policy of HDFS to choose the targets (see line 33).


Incremental data file, S1
s12
s11
Data transformations
Existing dimension data file, D1
d13
d12
d11
Merge
(Update SCD attr. values)

< url,	size,	moddate >
<www.dom.com/p0.htm,	30,	2011-03-01>
<www.dom.com/p0.htm,	40,	2011-04-01>
...		

<id,	url,	size,	version,	validfrom,	validto>
<4,	www.dom.com/p0.htm,	30,	nil ,	2011-03-01,	nil >
<5,	www.dom.com/p0.htm,	40,	nil,	2011-04-01,	nil >
		...			

<id,	url,	size,	version,	validfrom,	validto >
<1,	www.dom.com/p0.htm,	10,	1,	2011-01-01,	2011-02-01>
<2,	www.dom.com/p0.htm,	20,	2,	2011-02-01,	nil >
<3,	www.dom.com/p1.htm,	10,	1,	2011-01-01,	nil >
		...			

<id,	url,	size,	version,	validfrom,	validto >
<1,	www.dom.com/p0.htm,	10,	1,	2011-01-01,	2011-02-01>
<2,	www.dom.com/p0.htm,	20,	2,	2011-02-01,	2011-03-01>
<4,	www.dom.com/p0.htm,	30,	3,	2011-03-01,	2011-04-01>
<5,	www.dom.com/p0.htm,	40,	4,	2011-04-01,	nil >
<3,	www.dom.com/p1.htm,	10,	1,	2011-01-01,	nil >
		...			

Figure 5.7: Type-2 SCD updates in mapper for D_1 and S_1

CloudETL also offers a program for partitioning data that is not already partitioned. It runs a MapReduce job to partition the data into a number of files in HDFS based on a user-specified business key. Data with the same business key value is written into the same file and sorted by SCD attribute values. For example, before loading `pagedim`, we can partition the incremental source data on `url` and sort it on `moddate` if it is not already partitioned.

As a partitioned file and its co-located existing dimension data file both are sorted, we can simply run a map-only job to merge the data from the two co-located files. Figure 5.7 illustrates the processing of `pagedim` on `node1` which contains the co-located files D_1 and S_1. The lines from S_1 (in the blocks s_{11} and s_{12}) are first processed by user-defined transformers. We then merge them with the lines from the existing dimension data file D_1 (in the blocks d_{11}, d_{12} and d_{13}). The SCD updates are performed during the merging, and the final dimension values are written to HDFS. As explained above, it can, however, happen that the incremental data does not get totally co-located with the existing dimension data, e.g., if a data node lacks free space. In that case, a reduce-side update is used. The job submission client checks the co-location and decides which update method to use.

5.4.6 Snowflaked Dimensions

We now describe how to process snowflaked dimension tables. We thus normalize the `pagedim` dimension of the running example and get the schema in Figure 5.8. As shown, there exist several foreign-key references between the snowflaked dimension tables, e.g., `pagedim` references `domaindim` and `domaindim` references `topdomaindim`. When the jobs are planned, the dependencies of the tables are taken into account.

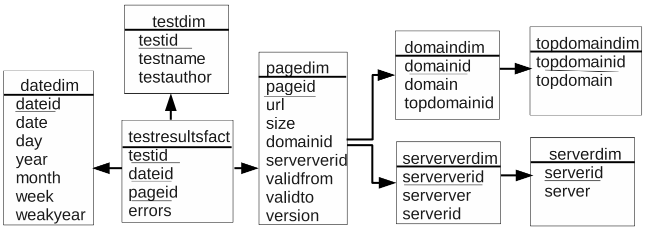


Figure 5.8: Partial snowflake schema of the running example

For simplicity, consider just the two dependent tables `topdomaindim` (for brevity, referred to as T_1) and `domaindim` (T_2). The execution flow is shown in Figure 5.9. It consists of the two sequential jobs, Job$_1$ and Job$_2$, that process T_1 and T_2, respectively. The input of Job$_1$ consists of two parts: the incremental data (denoted $\triangle_{T_1}$) and the existing dimension data (denoted D_1). When Job$_1$ has completed successfully, T_1 holds both the old and the newly added dimension values, i.e., $D_1 + \triangle_{T_1}$. When Job$_1$ writes the output into Hive, a side-product, called a *lookup* index (denoted LKI_{T_1}), is also generated for T_1. LKI_{T_1} is distributed to all nodes and is used for retrieval of dimension key values by the subsequent job Job$_2$.

The input of Job$_2$ consists of the lookup index LKI_{T_1}, the incremental data $\triangle_{T_2}$ and the existing dimension data D_2. LKI_{T_1} is read into an in-memory hash dictio-

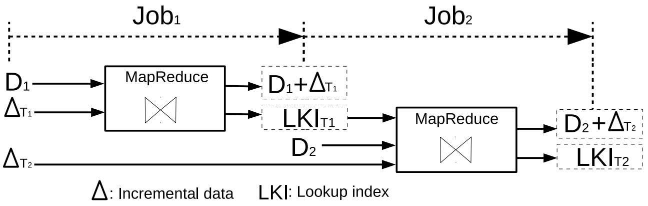


Figure 5.9: Process Snowflaked dimensions, T_1 and T_2

nary in the mapper initialization, and the foreign key values are retrieved in the map method. The retrieval of the dimension key values is similar to the lookups in fact processing which are described in the next section.

5.5 Fact Processing

Fact processing is the second step in the ETL flow in CloudETL. It involves retrieving surrogate key values from the referenced dimension tables. We call this operation *lookup*. Hive, however, does not support fast lookups from its tables. In addition, the size of fact data is typically very large, several orders of magnitude larger than the size of dimension data. It is, thus, very important to support efficient fact processing. Therefore we use *multi-way lookups* to retrieve the dimension key values through the so-called *lookup indices* and exploit a map-only job to process fact data.

5.5.1 Lookup Indices

A lookup index (which is generated during dimension processing) contains the minimal information needed for doing lookups, including the business key values, the dimension key values, and the SCD dates for type-2 SCDs. The data of lookup indices is read into main memory in the map initialization. The structures of lookup indices for type-2 and type-1 SCDs are shown in Figure 5.10. As a type-2 SCD index needs to keep track of the changes of the dimension, the index maps a business key value to all the versions that have the business key value. The result is thus a list of key-value pairs with the format $\langle$ *SCD effective date, dimension key*$\rangle$ in descending order such that the newest version is the first item to be retrieved by a lookup operation. Given the business key `www.dom.com/p0.htm` and the date `20110301`, we can thus find the surrogate key value of the correct dimension member version by first using the hash dictionary and then getting the first element from the resulting list. For type-1 SCDs, only the first step is needed (see the lower part in Figure 5.10).

In dimension processing, a lookup index file is generated for the incremental data for each dimension table. It is stored as a Hadoop sequence file. Instead of distribut-

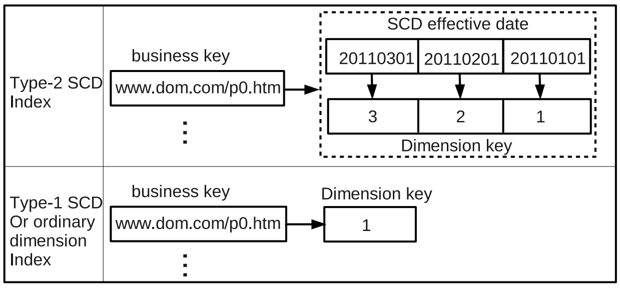


Figure 5.10: Lookup indices of page dim., $LKIs$

ing the full sets of the dimension values, only the minimum values are saved in the lookup index. The lookup index is distributed and kept in each node permanently (for handling of big dimensions, see Section 5.5.3).

5.5.2 Multi-way Lookups

We now describe how the lookup indices are used during the fact processing (see Algorithm 15). We run a map-only job to process the fact data. In the map initialization, the lookup indices relevant to the fact data are read into main memory (see line 2–7). A mapper first does all data transformations (see line 9). This is similar to the dimension processing. Then, the mapper does multi-way lookups to get dimension key values from the lookup indices (see line 10–11). Finally, the mapper writes the map output with the name of the fact table as the key, and the record (a processed fact) as the value (see line 12). The record is directly written to the directory in HDFS (named by the key of map output) using a customized record writer. Note that all mappers can work in parallel on different parts of the fact data since the lookup indices are distributed.

We now give more details about the lookup operator in the algorithm (see line 14–23). If it is a lookup in a type-2 SCD, we first get all the versions from the SCD index by the business key, bk (see line 17). Recall that different versions are sorted by the SCD effective dates in descending order. We get the correct version by comparing the effective date of a version and the SCD date, sd (see line 19–21). For the lookup index of a type-1 SCD table, the dimension key value is returned directly through a hash lookup operation (see line 23).

5.5.3 Lookup on a Big Dimension Table

Typically, the lookup indices are small and can be fully cached in main memory. However, when a dimension table is very big, and its lookup index is too big to

Algorithm 15 Map for fact processing and lookup

```
class Mapper
    function INITIALIZE( )
        $f \leftarrow$ GETCURRENTFACTTABLE()
        $D \leftarrow$ GETTHEREFERENCEDDIMENSIONS($f$)
        for all $d \in D$ do
            if $LKI[d] = \varnothing$ then
                $LKI[d] \leftarrow$ Read the lookup index of $d$ from local file system
    function MAP(offset $o$, Record $r$)
        $r \leftarrow$ TRANSFORM($r$)
        for all $d \in D$ do
            $r[d.key] \leftarrow LKI[d]$.LOOKUP($r$)
        EMIT($f, r$)

function LOOKUP(Record $r$)
    $bk \leftarrow r[bkey]$
    if $type = type2SCD$ then
        $sd \leftarrow r[scdDate]$
        $V \leftarrow LKI[d][bk]$                    ▷ Get versions by the business key, $bk$
        for all $v \in V$ do
            if $sd \geqslant v.date$ then
                return v.id
    else
        return $LKI[d][bk]$
```

be fully cached in the main memory, we propose the following two approaches for retrieving dimension key values. The first, called the *hybrid solution*, makes use of a Hive join and the multi-way lookups. The source fact data is first joined with the big dimension table to retrieve the dimension key values in Hive and then the lookup indices are used for the small dimension tables in CloudETL. The other solution, called the *partitioned lookup-index solution*, uses multi-way lookups for both big and small dimension tables which requires using a partitioned lookup index. The partitioned lookup index is generated for the big dimension table. Recall that we assume that the source data for a big dimension table is partitioned on the business key values. A partition of the data is processed into the dimension values saved in a data file in HDFS. At the same time, a lookup index is also generated for each partition. We call this a *partitioned lookup index*. If the fact source data is also partitioned with the same partitioning function as for the big dimension data (this is, e.g., possible in case the source data comes from another MapReduce job or from a database), a map-only job is run to do the multi-way lookups. We illustrate this in Figure 5.11. Suppose the job runs n mappers, each of which processes a partition of fact source data. Each mapper reads a partition of the lookup index for `pagedim`, and the (full) lookup indices for `datedim` and `testdim` (small dimension tables) into memory, and then does multi-way lookups.

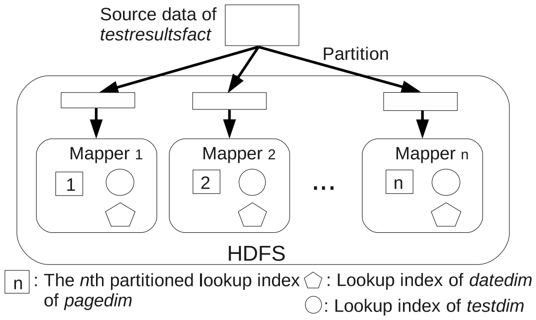

Source data of
testresultsfact
Partition
Mapper 1
1
Mapper 2
2
...
Mapper n
n
HDFS
n : The *n*th partitioned lookup index of *pagedim*
: Lookup index of *datedim*
: Lookup index of *testdim*

Figure 5.11: Fact processing with partitioned big lookup index

5.6 Implementation

5.6.1 Input and Output

Figure 5.12 shows the input and output of CloudETL (both located in HDFS). We first describe the output. The output is organized as hierarchical directories and data files in HDFS. Hive employs HDFS to store data. A folder and the files in it can be recognized as a logical table without changing the data format (via the `SerDe` customized storage formats). The schema of a table (names and data types of of attributes) are saved in the metastore. When a table is created, a folder with the same name is created in HDFS. We can "insert" data into the table by simply adding files in the folder. To make Hive recognize dimension data and fact data, we write the data into the files with the input format that we have specified when the table was created in Hive (a tabular file format is used as the default).

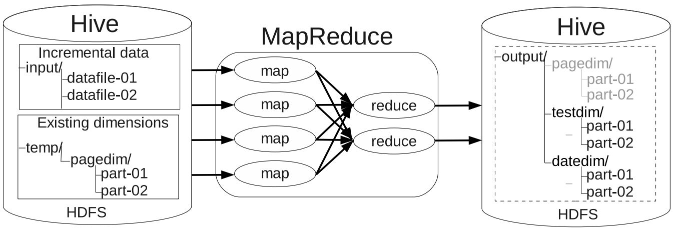


Figure 5.12: Input and output of CloudETL

We now describe the input. As discussed in Section 5.4, processing SCDs requires updating both existing and incremental data. Therefore, the input typically consists of two parts, incremental data and the existing dimension data (see Figure 5.12). The existing data files of each dimension table are in a separate folder in the HDFS (see the left of Figure 5.12). Suppose that we now process `pagedim`.

When the job starts, the existing dimension data of pagedim is first moved into a temp directory. (If there are other dimension tables to be processed together, their data files are moved into this temporary directory as well. Moving files between folders only needs to update the meta-data in the name node). The data in the temporary folder and the incremental data both serve as the input to the job (see the left of Figure 5.12). When the job has finished, the new output is written and the temporary folder and the files underneath are removed. If the job does not finish successfully, CloudETL does a *rollback* simply by moving the files back to the original folder.

In fact processing, no update is done on existing fact data. The incremental fact data files are just added to the directory of the fact table.

5.6.2 Transformation

CloudETL exploits transformers to execute the data transformations from sources to target tables, as described in Section 5.4. Within a job, CloudETL supports multiple transformers that process data into different tables. Figure 5.13 shows an example with three transformers in a job. The transformers 1 and 2 use the same data source, but process data into the two different tables Table 1 and Table 2. Transformer 3 uses a different source and processes data into Table 3. CloudETL provides built-in transformation operators, including filter, field selector, lookup, sequence, and splitter, and users can also create their own transformation operators by implementing a Java interface. The operators provide the abstractions to easily implement a dimensional ETL program while the details about the execution on Hadoop are transparent to users.

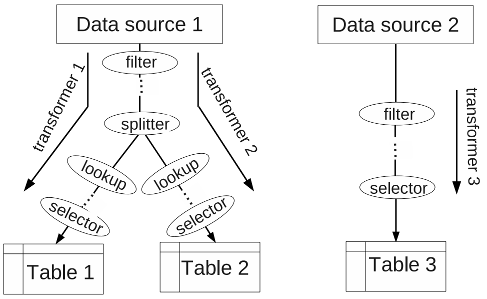


Figure 5.13: The data transformers in a job

In the following, we introduce the transformation operations of split, join and sequential number generation.

Split and Join: The operator splitter is used to process data from a single source into different tables. The upstream transform pipe of a splitter is shared by the downstream transform pipes. For example, Figure 5.14 shows three transform

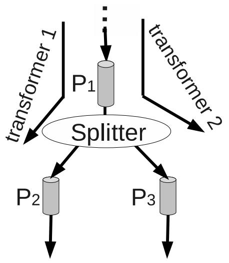


Figure 5.14: Splitter

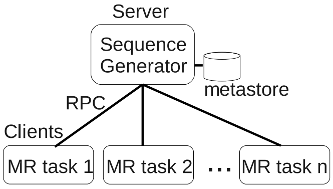


Figure 5.15: Sequential number generator

pipes, P_1, P_2, and P_3, which are used by two transformers that get data from the same source. P_1 is thus shared by P_2 and P_3 such that the data goes through P_1 is only processed once for the downstream transform pipes, P_2 and P_3. Here, the observer design pattern is used such that P_2 and P_3 are notified to process the data when the state of the `splitter` is changed. For join operations, we make use of Hive as it has intensively optimized join algorithms, including equijoins, outer joins, left semi joins, and map side joins. The data of two relations in Hive is joined by simply writing a join query. The output of the join then serves as the input to transformation pipes for further processing.

Sequential number generation: One of the features lacking in Hadoop is shared global states across nodes. In CloudETL, a sequential number generator is implemented (see Figure 5.15) and used to generate unique surrogate key values for a dimension table processed by all tasks. The number generator has a master-worker architecture and uses remote procedure call (RPC) as the communication protocol between the master and a worker (RPC is also the communication protocol between the name node and data nodes in Hadoop). During the data processing, a MapReduce task always asks for the next sequential numbers from the number generator. To reduce the network overhead of frequent calls, a task fetches a range of numbers in each call instead of a single number.

5.6.3 Job Planner

We now describe the job planner which plans jobs based on the dependencies of the tables to be processed. For example, a fact table has foreign-key dependencies on its referenced dimension tables and snowflaked dimension tables also contain foreign-key dependencies between the participating tables. Figure 5.16 shows the table dependencies of our running example (see the left), and the four sequential jobs that are planned to process the dimension and fact tables (see the right, the arrow represents the job execution order). In the job planning, a separate job is planned for a big dimension table whenever possible such that a map-only job can be planned to improve the performance if the data is co-located. The tables without dependencies are planned into a single job.

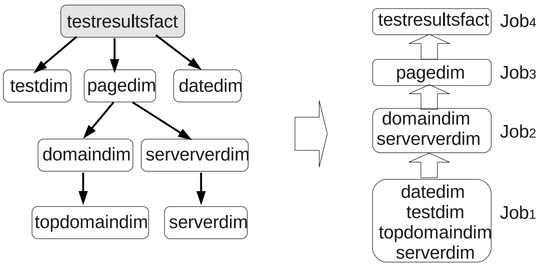


Figure 5.16: Job planning

The job planner plans jobs by pruning the leaves of the tree of the dependent tables (see the left of Figure 5.16). The leaves are the tables without foreign keys referencing other tables, e.g., `topdomaindim`, `serverdim`, `testdim`, and `datedim`. The leaves are pruned step by step until the whole tree is pruned. The leaves are put into a single job in each pruning. In the end, we get the execution plan in the right of Figure 5.16, in which `pagedim` is a big dimension table within its own job.

5.6.4 An ETL Program Example

We now show how to use CloudETL to easily implement a parallel dimensional ETL program. The code in Figure 5.17 shows how to process the `pagedim` SCD. The implementation consists of four steps: 1) define the data source, 2) create the transform pipe and the transformation operators, 3) define the target table, and 4) add the transformer to the job planner and start. When the data source is defined, the schema information, the business key, and the date attribute to use in SCD processing are specified (see line 1–5). CloudETL provides a variety of commonly used transformation operators and lines 10–12 show the operators for excluding a field, adding the dimension key (the defined sequence object will automatically get sequential numbers from the number generator), and renaming a field, respectively. Lines 16–21 define the target, a type-2 SCD table parameterized by the name of a table, the names of attributes and their data types, and SCD attributes. CloudETL also offers other dimension and fact classes. Here, only one statement is needed to define a target while the complexity of processing SCDs is transparent to users. Finally, the transformer is added to the job planner, and the job planner automatically plans and submits jobs to the Hadoop JobTracker (see line 24).

This is much more difficult to do in Hive. As Hive does not support UPDATEs needed for SCDs, we use a cumbersome and labour-intensive workaround including overwriting tables and joins to achieve the update effect. The HiveQL codes of initial

```
 /* 1) Define the source */
String inputDir = "/user/cloudetl/input/pages";
Reader  pagesReader = new CSVFileReader(inputDir)
                .setField("url", DataType.STRING, FieldType.BKEY)
                .setField("size", DataType.INT)
                .setField("moddate", DataType.DATE, FieldType.SCD_DATE);

 /* 2) Create the transform pipe, and add the  transformation
        operators for cleansing data. */
Pipe pipe = new TransformPipe(pagesReader)
                .add(new ExcludeFields("size"))
                .add(new AddField("pageid", new Seq("pageid"), DataType.INT))
                .add(new RenameField("moddate", "validfrom"));

 /* 3) Define the target */
 String outputDir = "/user/cloudetl/output";
 Writer pagedim = new SlowlyChangingDimensionWriter(outputDir, "pagedim")
                .setField("pageid", DataType.INT, FieldType.PKEY)
                .setField("url", DataType.STRING, FieldType.LOOKUP)
                .setField("version", DataType.INT, FieldType.SCD_VERSION)
                .setField("validfrom", DataType.DATE, FieldType.SCD_VALIDFROM)
                .setField("validto", DataType.DATE, FieldType.SCD_VALIDTO);

 /* 4) Add transformer and start ETL */
 JobPlanner.addTransformer(pipe, pagedim).start();
```

Figure 5.17: The ETL program for `page` SCD

```
 /* 1) Define the fact data source */
String inputDir =   "/user/cloudetl/input/testresults";
DataReader  testResultsReader = new CSVFileReader(inputDir)
                        .setField("localfile", DataType.STRING)
                        .setField("url", DataType.STRING)
                        .setField("lastmoddate", DataType.DATE)
                        .setField("downloaddate", DataType.DATE)
                        .setField("test", DataType.STRING)
                        .setField("errors", DataType.INT);

 /* 2) Do the necessary data transformation and look up dimension key values */
Pipe testresultsfactPipe = new TransformPipe(testResultsReader)
   .add(new ExcludeFields("localfile"))
   .add(new LookupTransformer("pageid", new SCDLookup(pagedim, "url", lastmoddate", -1)))
   .add(new LookupTransformer("dateid", new Lookup(datedim, "downloaddate", -1)))
   .add(new LookupTransformer("testid", new Lookup(testdim, "test", -1)));

  /* 3) Define the target fact table */
   String outputDir = "/user/cloudetl/fact";
   DataWriter testresultsfact = new FactTableWriter(outputDir, "testresultsfact")
                        .setField("pageid", DataType.INT)
                        .setField("dateid", DataType.INT)
                        .setField("testid", DataType.INT)
                        .setField("errors", DataType.INT);

 /* 4) Add transformer and start ETL */
JobPlanner.addTransfer(testresultsfactPipe, testresultsfact).start();
```

Figure 5.18: The ETL program for fact processing

and incremental loads are shown in Program 6 and 7 respectively. CloudETL has much better programming efficiency and only needs 6 statements (818 characters, a statement ended by “;”), while Hive uses 112 statements (4192 characters, including the statements for HiveQL and UDFs) and the code is less intuitive.

The code in Figure 5.18 shows how to do the fact processing. Only 6 statements with 811 characters are needed. In Hive, we need 12 statements with 938 characters for the fact processing (see Program 8).

Program 6 HiveQL script for the initial load of type-2 SCDs

```
-- 1. Create the type-2 SCD table:

CREATE TABLE IF NOT EXISTS pagescddim(pageid INT, url STRING, serverversion
STRING, size INT, validfrom STRING, validto STRING, version INT) ROW FORMAT
DELIMITED FIELDS TERMINATED BY ’\t’ LINES TERMINATED BY ’\n’ STORED AS
TEXTFILE;

-- 2. Create the data source table and load the source data:

CREATE TABLE pages(localfile STRING, url STRING, serverversion STRING,
size INT, downloaddate STRING, moddate STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY ’\t’ STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH ’/q/disk_0/xiliu/dataset/pages40GB.csv’ OVERWRITE
INTO TABLE pages;

-- 3. Create UDFs, add the UDFs jars to the classpath of Hive, and create
--    the temporary functions for the UDFs:

// Create the UDFs for reading dimension key values from the global sequential
// number generator.
package dk.aau.cs.hive.udf;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;

public final class SeqUDF extends UDF {
        enum ServerCmd {BYE, READ_NEXT_SEQ}
        String seqName = null;
        byte[] nameInBytes;
        IntWritable curSeq = new IntWritable();
        IntWritable endSeq = new IntWritable();
        SocketChannel channel;
        ByteBuffer buffer = ByteBuffer.allocate(512);
        Configuration conf;
        static final String hostname = "localhost";
        static final int port = 9250;
        final IntWritable delta = new IntWritable(10000);

    public IntWritable evaluate(final Text name) {
          if (name.toString().equalsIgnoreCase("close")){
               this.cleanup();
               return new IntWritable(-1);
          }
          try {
               if (seqName == null) {
                    this.seqName = name.toString();
                    this.nameInBytes = SeqUDF.getBytesUtf8(this.seqName);
                    this.setup();
               }
               return new IntWritable(this.nextSeq());
          } catch (Exception e) {
               e.printStackTrace();
          }
          return new IntWritable(-1);
    }

    private void setup() {
        try {
            this.channel = SocketChannel.open(new InetSocketAddress(hostname, port));
            this.curSeq.set(this.readNextFromServer());
            this.endSeq.set(curSeq.get() + delta.get());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private int readNextFromServer() throws IOException {
        buffer.clear();
        buffer.putInt(ServerCmd.READ_NEXT_SEQ.ordinal())
              .putInt(nameInBytes.length).put(nameInBytes).flip();
        channel.write(buffer);

        buffer.clear();
        channel.read(buffer);
        return buffer.getInt(0);
    }

    private int nextSeq() throws IOException {
        if (curSeq.get() >= endSeq.get()) {
            this.curSeq.set(readNextFromServer());
            this.endSeq.set(curSeq.get() + delta.get());
        }
        int ret = curSeq.get();
        curSeq.set(ret+1);
        return ret;
    }
    private void cleanup() {
        try {
            buffer.clear();
            buffer.putInt(ServerCmd.BYE.ordinal()).flip();
            channel.write(buffer);
            channel.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static byte[] getBytesUtf8(String string) throws UnsupportedEncodingException{
        if (string == null) {
            return null;
        }
        return string.getBytes("UTF-8");
    }
}

// Create the UDF for generating version IDs for SCDs
package dk.aau.cs.hive.udf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;

public final class VersionIDUDF extends UDF {
    IntWritable curID = new IntWritable();
    Text bkey = new Text();

    public IntWritable evaluate(final Text bkey) {
        if (bkey.toString().equalsIgnoreCase(this.bkey.toString())) {
            int curVersionID = curID.get();
            IntWritable ret = new IntWritable(curVersionID);
            curID.set(curVersionID + 1);
            return ret;
        } else {
            IntWritable ret = new IntWritable(1);
            curID.set(2);
            this.bkey.set(bkey);
            return ret;
        }
    }
}

-- Add the UDF jar package to the classpath of Hive
ADD jar file:///hive/hiveudfs.jar;

-- Create the temporary functions in Hive for UDFs
CREATE TEMPORARY FUNCTION nextseq AS ’dk.aau.cs.hive.udf.SeqUDF’;
CREATE TEMPORARY FUNCTION versionid AS ’dk.aau.cs.hive.udf.VersionIDUDF’;

-- 4. Load data from the source to type-2 pagescddim table.
--    This requires several intermediate steps.

CREATE TABLE pagestmp1 (pageid INT, url STRING, serverversion STRING,
size INT, validFROM STRING, validto STRING, version INT) ROW FORMAT
DELIMITED FIELDS TERMINATED BY ’\t’ STORED AS TEXTFILE;

CREATE TABLE pagestmp2 (pageid INT, url STRING, serverversion STRING,
size INT, validfrom STRING, validto STRING, version INT) ROW FORMAT
DELIMITED FIELDS TERMINATED BY ’\t’ STORED AS TEXTFILE;
```

```
-- Sort by SCD date for each group
INSERT OVERWRITE TABLE pagestmp1 AS SELECT nextseq('pagedim_id') AS pageid,
url, serverversion, size, moddate AS validfrom, NULL AS validto, NULL AS version
FROM pages ORDER BY url, moddate ASC;

-- Add the version number
INSERT OVERWRITE TABLE pagestmp2 AS SELECT pageid, url, serverversion, size,
validfrom, validto, versionid(url) AS version FROM pagestmp1;

DROP TABLE pagestmp1;

-- Update the validto attribute values, 1 Job
INSERT OVERWRITE TABLE pagescddim SELECT a.pageid, a.url, a.serverversion, a.size,
a.validfrom, b.validfrom AS validto, a.version FROM pagestmp2 a  LEFT OUTER JOIN
pagestmp2 b ON (a.version=b.version-1 AND a.url=b.url);

DROP TABLE pagestmp2;
```

Program 7 HiveQL script for the incremental load of type-2 SCDs

```
CREATE TABLE pagestmp3 (pageid INT, url STRING, serverversion STRING, size INT,
validfrom STRING, validto STRING, version INT) ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t' STORED AS TEXTFILE;

INSERT OVERWRITE TABLE pagestmp3 SELECT * FROM (SELECT pageid, url,
serverversion, size, validfrom, validto, version FROM pagescddim UNION ALL
SELECT pageid, url, serverversion, size, validfrom, validto, version
FROM pagestmp2) a ORDER BY a.url, a.validfrom ASC;

CREATE TABLE pagestmp4 AS SELECT pageid, url, serverversion, size, validfrom,
validto, versionid(url) AS version FROM pagestmp3;

DROP TABLE pagestmp3;

INSERT OVERWRITE TABLE pagescddim SELECT a.pageid, a.url, a.serverversion,
a.size,  a.validfrom, b.validfrom as validto, a.version FROM pagestmp4 a
LEFT OUTER JOIN pagestmp4 b  ON (a.version=b.version-1 AND a.url=b.url);

DROP TABLE pagestmp4;
```

Program 8 HiveQL script for fact processing

```
-- 1. Create fact table:
CREATE TABLE IF NOT EXISTS testresultsfact(pageid INT, testid int, dateid int,
error int) ROW FORMAT DELIMITED FIELDS TERMINATED BY ’\t’ LINES
TERMINATED BY ’\n’ STORED AS TEXTFILE;

-- 2. Create the data source table and load the source data:
CREATE TABLE testresults(localfile STRING, url STRING, serverversion STRING,
test STRING, size INT, downloaddate STRING, moddate STRING, error INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ’\t’ STORED AS TEXTFILE ;

LOAD DATA LOCAL INPATH ’/q/disk_0/xiliu/dataset/testresults.csv’
OVERWRITE INTO TABLE testresults;

-- 3. Filter the unnecessary attribute values and look up dimension key values:
CREATE TABLE testresults_tmp1 AS SELECT b.pageid, downloaddate, serverversion,
test, error FROM testresults a LEFT OUTER JOIN pagedim b
ON (a.url=b.url AND moddate>=validfrom AND moddate<validto);

CREATE TABLE testresults_tmp2 AS SELECT a.pageid, b.testid, a.serverversion,
a.downloaddate, a.error FROM testresults_tmp1 a LEFT OUTER
JOIN testdim b ON (a.test = b.testname);

DROP TABLE testresults_tmp1;

CREATE TABLE testresults_tmp3 AS SELECT a.pageid, b.testid, a.dateid, a.error
FROM testresults_tmp2 a LEFT OUTER JOIN datedim b ON (a.downloaddate = b.date);

DROP TABLE  testresults_tmp2;

CREATE TABLE testresults_tmp4 AS SELECT * FROM (SELECT * FROM testresults_tmp3
UNION ALL SELECT pageid, testid, dateid, error FROM testresultsfact);

DROP TABLE  testresults_tmp3;

INSERT OVERWRITE TABLE testresultsfact AS SELECT * from testresults_tmp4;

DROP TABLE testresults_tmp4;
```

5.7 Performance Study

In this section, we empirically evaluate the performance of CloudETL by studying 1) the performance of processing different DW schemas, including a star schema, a snowflake schema, and schemas with an SCD and a big dimension table, and 2) the effect of the various optimization techniques applied to CloudETL, including the pre-updates in the mapper and co-location of data. We compare CloudETL with our previous work ETLMR [50] which is a parallel ETL programming framework using MapReduce. ETLMR is selected because CloudETL is implemented with the same spirit as ETLMR and both make use of MapReduce to parallelize ETL execution. ETLMR is, however, designed for use with an RDBMS-based warehouse system. We also compare with Hive and the co-partitioning of Hadoop++ [25].

Cluster setup: Our experiments are performed using a local cluster of nine machines: eight machines are used as the DataNodes and TaskTrackers, each of them has two dual-core Intel Q9400 processors (2.66GHz) and 3GB RAM. One machine is used as the NameNode and JobTracker and it has two quad-core Intel Xeon E5606 (2.13GHz) processors and 20GB RAM. The disks of the worker nodes are 320GB SATA hard drives (7200rpm, 16MB cache, and 3.0Gb/s). Fedora 16 with the 64-bit Linux 3.1.4 kernel is used as the operating system. All machines are connected via a gigabit Ethernet switch with bandwidth 1Gbit/s.

We install Hadoop 0.21.0 on all machines and use Hive 0.8.0 as the warehouse system. Based on the number of cores, we configure Hadoop to run up to four map tasks or four reduce tasks concurrently on each node. Thus, at any point in time, at most 32 map tasks or 32 reduce tasks run concurrently. The following configuration parameters are used: the sort buffer size is set to 512MB, JVMs are reused, speculative execution is turned off, the HDFS block size is set to 512MB, and the replicator factor is set to 3. Hive uses the same Hadoop settings. For ETLMR, we use Disco 0.4 [24] as MapReduce platform (as required by ETLMR), set up the GlusterFS distributed file system (the DFS that comes with Disco) in the cluster, and use PostgreSQL 8.3 as the DW DBMS.

Data sets: We use generated data sets for the running example and consider both the star schema (see Figure 5.2) and the snowflake schema (see Figure 5.8). In the experiments, the *pagedim* and *datedim* dimensions get data from the same source data set which we scale from 40 to 320GB. Every 10GB of source data result in 1.85GB *pagedim* dimension data (113,025,455 rows) and 1.01MB *datedim* dimension data (41,181 rows). The *testdim* dimension has its own source data set with a fixed size of 32KB (1,000rows). The fact source data set is also scaled from 40 to 320GB. Every 10GB of source data result in 1.45GB fact data (201,233,130 rows) in the DW. The reason that the size of the loaded data is smaller is than the dimension source data contains redundant data, while the dimension attribute values in fact source data are replaced by integer values referencing the dimension keys. The data generator and the CloudETL source code are available at `http://people.cs.aau.dk/~xiliu/CloudETL`.

5.7.1 Dimension Data Processing

Star and snowflaked dimension tables: In the first experiments, we process data into the dimension tables of both the star schema and the snowflake schema. To make Hive support dimension processing, we implement a number generator similar to CloudETL's to generate the dimension key values and use a user-defined function (UDF) to get the numbers. We use all 32 tasks to process the data without any SCD and scale the data from 40GB to 320GB. We measure the time from the start to the end of each run.

Figure 5.19 shows the results for the star schema. CloudETL processes the three dimension tables within one job and does data transformations in mappers. The data for a dimension table is collected and written to HDFS in a reducer. Hive, however, has to process the statements for different dimension tables in different jobs. The total time used by Hive is up to 28% higher than the time used by CloudETL (the time for `testdim` is not shown in Figure 5.19 since it is negligible). During the tests, we observe that Hive uses only map to process `pagedim` and `testdim`, but uses both map and reduce to process `datedim` since `datedim` requires the DISTINCT operator to find duplicate records. ETLMR uses its so-called offline dimension scheme in which the data is first processed and stored locally on each node, then collected and loaded into the DW by the DBMS bulk loader (PostgreSQL's COPY is used in this experiment). As shown, ETLMR is efficient to process relatively small-sized data sets, e.g., 40GB, but the time grows fast when the data is scaled up and ETLMR uses about 81% more time than CloudETL for 320GB.

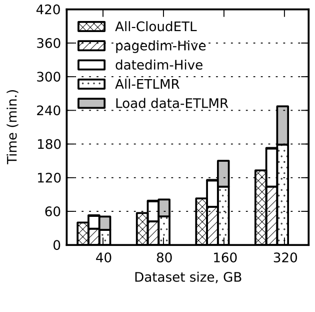


Figure 5.19: Star schema, no SCDs

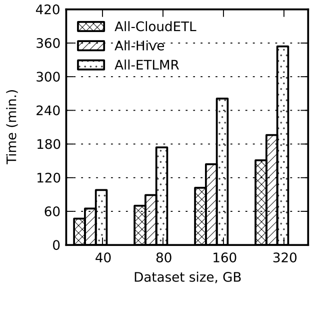


Figure 5.20: Sf. schema, no SCDs

We now load the snowflaked dimension tables. Due to the dependencies of the tables, three jobs are planned by CloudETL's job planner, i.e., Job_1 handles `da-`

tedim, testdim, topdomaindim, and serverdim while Job_2 then handles domaindim and serververdim and Job_3 handles pagedim. However, the tables in Job_2 and Job_3 are normalized from the single flat pagedim table of the star schema, and they use the same source data. To optimize, we first do the data transformations for the tables handled by Job_2 and Job_3 in the same job that handles the tables of Job_1. We call this combined job Job'_1. In other words, Job'_1 processes datedim, testdim, topdomaindim, serverdim, domaindimtmp, serververdimtmp, and pagedimtmp. Job_2 and Job_3 then use the intermediate data from the three temporary tables domaindimtmp, serververdimtmp and pagedimtmp as their inputs, and process the data into the final three dimension tables, respectively. This optimization reduces the size of the inputs for Job_2 and Job_3, and both the jobs only need to look up the foreign key values and do the dimension key values assignment. Figure 5.20 shows the test results. All the three systems use more time to process the snowflake schema than the star schema, which is due to more dimension tables being processed. Hive uses more time than CloudETL and uses 30% more time when processing 320GB. This is because Hive has to process the snowflaked dimension tables in several jobs which involve several rounds where the same source data is read. In addition, getting the foreign key values requires Hive to do joins. In contrast, CloudETL uses only three jobs that can process multiple tables. ETLMR uses about 2.05–2.56 times longer time than CloudETL since ETLMR only supports snowflake schemas in the online dimension scheme in which SQL INSERT and SELECT operations are used. The operations become more expensive when more data has been loaded.

Processing a type-2 SCD: We now study the performance when processing the big dimension table pagedim which is a type-2 SCD (see Figure 5.21 and Figure 5.22). We test both initial load ("init.") and incremental load ("incr.") in this experiment. In an initial load, pagedim is cleared before a job starts. In an incremental load, 320GB source data is already loaded into pagedim before the job starts. For CloudETL, the initial and incremental loads are both tested using data with and without co-location. Figure 5.21 and Figure 5.22 show the results of the initial and incremental loads, respectively. The results show that data co-location improves the performance significantly and between 60% and 73% more time is used when there is no co-location. This is because the co-located data can be processed by a map-only job which saves time. CloudETL outperforms ETLMR. When the data is scaled to 320GB, ETLMR uses up to 2.2 and 2.3 times as long for the initial and incremental loads (without co-location), respectively. The processing time used by ETLMR grows faster which is mainly due to the database-side operation called post-fixing [50] used to set SCD attribute values correctly. CloudETL also outperforms Hive significantly. For example, when tested using 320GB data with and without co-location respectively, Hive uses up to 3.9 times and 2.3 times as long for the initial load, while for the incremental load, it uses up to 3.5 times and 2.2 times as long. This is due to that the workaround to achieve the update effect for the SCD handling requires several sequential jobs (4 jobs for the initial load and 5 jobs for the incremental load, see Program 6 and 7).

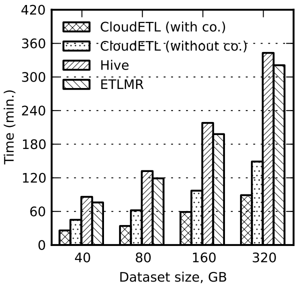


Figure 5.21: Init. loading type-2 *page* SCDs

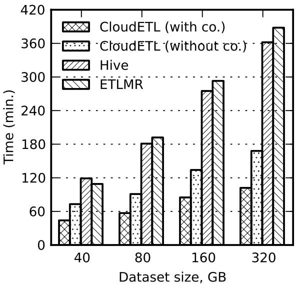


Figure 5.22: Incr. loading type-2 *page* SCDs

Compared with Hadoop++: We now compare with Hadoop++ for incremental load of `pagedim`. Hadoop++ co-partitions two data sets by adding a “Trojan” join and index through MapReduce jobs. The lines (from the two data sets) with identical join key values are put into the same data split and the same data node. We change our program to read the co-partitioned data splits, and to run a map-only job. Figure 5.23 shows the test results. The results include the times for prepartitioning the data sets and indicate that Hadoop++ uses nearly 2.2 times longer than CloudETL to partition the same data set. Processing the co-partitioned data also takes longer, 8%–14% more time. We found that Hadoop++ co-partition is much more tedious and has jobs for the following tasks: converting textual data to binary, co-partitioning, and creating index. In addition, for an incremental load, Hadoop++ has to rebuild the index from scratch which is increasingly expensive when the data amounts grow. The co-location of CloudETL, however, makes use of the customized block placement policy to co-locate the data. It is very light-weight and more suitable for incremental load.

Processing a type-1 SCD: We now process `pagedim` as a type-1 SCD and do the following three tests: T_1) we do the type-1 updates in reducers; T_2) we first do “pre-updates” in mappers, then do “post-updates” in reducers; and T_3) we first partition the source data, co-locate the partitioned files, and then do map-only updates. The results are shown in Figure 5.24. The map-only updates (T_3) are the fastest followed by pre- and post-updates (T_2) which use between 16% and 42% more time. Updates in the reducers (T_1) use between 28% and 60% more time. The ETLMR offline dimension scheme supports type-1 SCDs by processing data on MapReduce and loading the processed data into the DW (see also the discussion of Figure 5.19). It

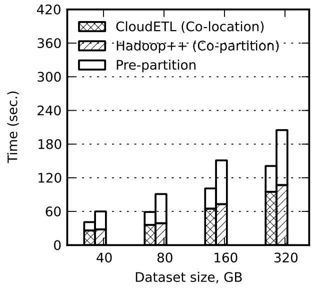


Figure 5.23: Proc. type-2 *page* SCDs

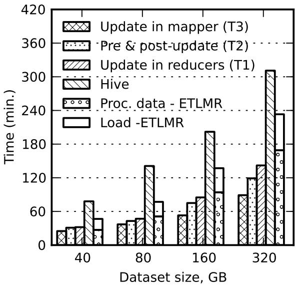


Figure 5.24: Proc. type-1 *page* SCDs

uses more time to process the scaled data, e.g., the time for 320GB is 16%, 42% and 90% more than that of T_1, T_2 and T_3, respectively. Hive requires 4 jobs to process the type-1 SCD and takes 3.5 times longer than CloudETL.

Speedup: We now study the speedup by varying the number of cores from 4 to 32. We do the speedup tests using 320GB `pagedim` data with and without co-location, respectively, i.e., the tests are done when only map is used, and when map and reduce both are used. Figure 5.25 shows the speedup lines of both tests. The results indicate that the speedup with data co-location is close to linear, and better than without co-location. In other words, loading co-located data can achieve better speedup since a map-only job is run for the data. The sub-linearity is mainly due to the communication cost as well as task setup overheads on Hadoop.

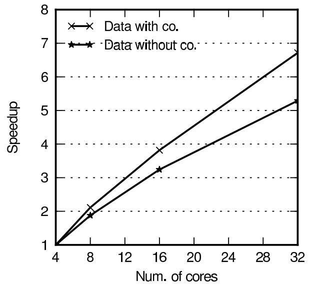


Figure 5.25: Speedup of dim. processing

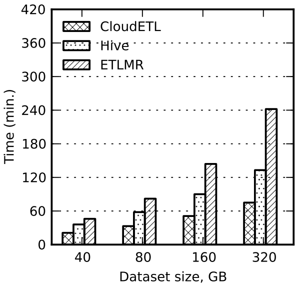


Figure 5.26: Proc. facts, small dims (SCD)

5.7.2 Fact Data Processing

We now study the performance of fact processing. Fact processing includes doing data transformations and looking up dimension key values. We load fact data into testresultsfact in this experiment and use both small and big dimension tables by varying the size of pagedim. With the small dimension table, the lookup indices are used and cached in main memory for multi-way lookups. The lookup index sizes are 32KB (testdim), 624KB (datedim), 94MB pagedim (as a traditional dimension, i.e., not an SCD) and 131MB pagedim (as a type-2 SCD). They are generated from 2GB dimension source data. For ETLMR, we use its offline dimension scheme when pagedim is used as a non-SCD. This is the ETLMR scheme with the best performance [50]. When pagedim is used as an SCD, however, the online dimension scheme is compared. This scheme retrieves dimension key values from the underlying RDBMS.For Hive, we join the fact data with each of the dimension tables to retrieve dimension key values. Figures 5.26 and 5.27 present the results from using the small dimension tables with and without SCD support, respectively. The comparison of the results in two figures shows that CloudETL (without SCD support) has the highest performance while the processing with an SCD uses about 5%–16% more time than without an SCD. CloudETL outperforms both ETLMR and Hive when using small dimension tables since a map-only job is run and in-memory multi-way lookups are used. In contrast, Hive requires four sequential jobs (an additional job is used for projection after getting the dimension key values) and uses up to 72% more time. ETLMR (with SCD support) takes about 2.1 times longer than CloudETL (see Figure 5.26) due to the increasing cost of looking up dimension key values from the DW. ETLMR (without SCD support) also runs a map-only job, and its performance is slightly better when processing the relatively small-sized data (see Figure 5.27), e.g., 40GB, but the time grows faster when the data is scaled.

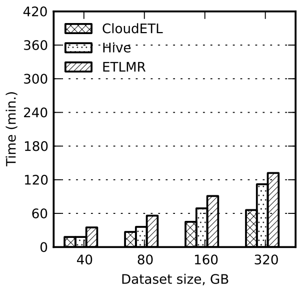


Figure 5.27: Proc. facts, small dims (no SCD)

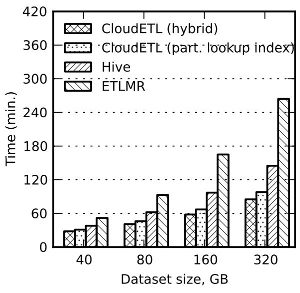


Figure 5.28: Proc. facts, big dims (SCD)

We now study the performance with big dimension tables. The dimension values in the big table pagedim is generated from 40GB source data. We use the following two approaches for the lookups. The first is the hybrid solution which uses a Hive join to retrieve the key values from the big dimension table and then uses multi-way lookups to retrieve the key values from the small dimension tables (the lookup index sizes are 4.5MB for `datedim` and 32KB for `testdim`). The other is the partitioned lookup-index solution. Again, we assume that the fact source data has already been partitioned. Each mapper caches only one partition of the big lookup index, but all the small lookup indices. A partition of the fact data is processed by the mapper which caches the look index partition that is relevant to the fact partition. A map-only job is run to do multi-way lookups. Figure 5.28 and 5.29 show the results with and without SCD support, respectively. As shown, CloudETL again outperforms both Hive and ETLMR. When the partitioned big lookup index is used, CloudETL is more efficient than when the hybrid solution is used. The partitioned lookup-index solution requires between 11% and 18% more time. Hive and ETLMR do not scale as well and, e.g, for 320GB and no SCD, they require 71% and 211% more time, respectively. We

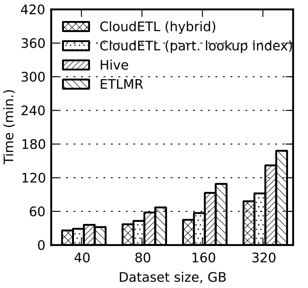


Figure 5.29: Proc. facts, big dims (no SCD)

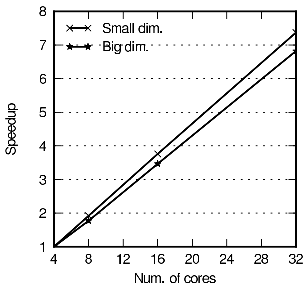


Figure 5.30: Speedup of fact processing

now study the speedup with the scaling up of the number of the nodes. Figure 5.30 shows the speedup when processing 320GB fact source data using small dimension tables and big dimension tables (using the hybrid solution). As shown, CloudETL achieves a nearly linear speedup in both cases. The speedup is slightly better when using small dimension tables than using big dimension tables. The reason for this is that the hybrid solution requires an additional Hive job for the big dimension.

5.8 Related Work

To tackle large-scale data, parallelization is the key technology to improve the scalability. The MapReduce paradigm [22] has become the de facto technology for large-scale data-intensive processing due to its ease of programming, scalability, fault-tolerance, etc. Multi-threading is another parallelization technology which has been used for a long time. Our recent work [83] shows multi-threading is relatively easy for ETL developers to apply. It is, however, only effective on Symmetric Processor

Systems (SMP) and does not scale out on many clustered machines. Thus, it can only achieve limited scalability. The two widely studied parallelism models Parallel Virtual Machine (PVM) [77] and Message Passing Interface (MPI) [30] provide abstractions for cluster-level parallelization by extensive messaging mechanisms. However, they are mostly designed for tackling processor-intensive problems and are complicated to use such that developers, e.g., have to manage the assignment of computing resources. The recent parallelization systems Clustera [23] and Dryad [38] support general cluster-level computations to data management with parallel SQL queries. They are, however, still only available as academic prototypes and remain far less studied than MapReduce.

Stonebraker et al. compare MapReduce with two parallel DBMSs (a row-store and a column-store) and the results show that the parallel DBMSs are significantly faster than MapReduce [60, 76]. They analyze the architectures of the two system types, and argue that MapReduce is not good at query-intensive analysis as it does not have a declarative query language, schema, or index support. Olston et al. complain that MapReduce is too low-level, rigid, hard to maintain and reuse [57]. In recent years, HadoopDB [2], Aster, Greenplum, Cloudera, and Vertica all have developed hybrid products or prototypes by using two class systems which use both MapReduce and DBMSs. Other systems built only on top of MapReduce but providing high-level interfaces also appear, including Pig [57] and Hive [86]. These systems provide MapReduce scalability but with DBMS-like usability. They are generic for large-scale data analysis, but not specific to ETL. For example, they do not support SCD updates. In contrast, CloudETL provides built-in support for processing different DW schemas including SCDs such that the programmer productivity is much higher. Our previous work, ETLMR, [50] extends the ETL programming framework pygrametl [84] to be used with MapReduce. ETLMR, however, is built for processing data into an RDBMS-based DW and some features rely on the underlying DW RDBMS, such as generating and looking up dimension key values. In contrast, CloudETL provides built-in support of the functionality needed to process dimensions and facts. ETLMR uses the RDBMS-side operation "post-fixing" to repair inconsistent data caused by parallelization, while CloudETL solves this issue by providing global key value generation and SCD updates on Hadoop. The recent project Cascading [18] is able to assemble distributed processes and plan them to run in a Hadoop cluster. The workflow of Cascading is somewhat similar to the transformers of CloudETL. However, Cascading does not consider DW-specific data processing such as key generation, lookups, and processing star, snowflake schemas and SCDs.

The co-location of data in CloudETL is similar to the co-location in CoHadoop [29]. CoHadoop is, however, a general extension to Hadoop that requires applications to explicitly co-locate files by using a common identifier (a "locator") when a file is created. Instead of changing Hive to do that, we in CloudETL exploit how files are named and define co-location by means of a regular expression. Further, this is fully implemented by using HDFS's standard block placement mechanism such that CloudETL requires no changes of the Hadoop/HDFS code. HadoopDB [2] and Hadoop++ [25] do also co-locate data. HadoopDB does so by using RDBMS instances instead of HDFS. Hadoop++ considers the entire data set when doing co-location which is impractical when incremental data is added.

Pig and Hive provide several join strategies in terms of the features of the joined data sets. HadoopDB [2] is a hybrid solution that uses both Hadoop and DBMS instances. It pushes joins to the DBMSs on each node. The join algorithms for MapReduce are compared and extensively studied in [14, 40]. The join implementations above process a join operation within one MapReduce job, which causes a non-trivial cost. To address this issue, [3, 39] propose multi-way joins which shuffle the joining of data to reducers in an one-to-many fashion and do the joins on the reduce side. This, however, becomes expensive if there are many tables to join. In contrast, CloudETL does "multi-way lookups" (similar to the multi-way joins) in map side when processing fact data and only a minimal amount of data is saved in the lookup index files and used for joins. This is much more efficient for our particular purpose.

5.9 Conclusion and Future Work

With the ever-growing amount of data, it becomes increasingly challenging for data warehousing technologies to process the data in a timely manner. This chapter presented the scalable dimensional ETL framework CloudETL for data warehousing. Unlike traditional data warehousing systems, CloudETL exploits Hadoop as the ETL execution platform and Hive as the warehouse system. CloudETL provides built-in support of high-level ETL-specific constructs for different DW schemas, including star schemas, snowflake schemas, and SCDs. The constructs facilitate easy implementation of parallel ETL programs and improve programmer productivity very significantly. We presented an approach for efficient processing of updates of SCDs in a distributed environment. We proposed a method for processing type-1 SCDs which does pre-updates in mappers and post-updates in reducers. We also presented a block placement policy for co-locating the files in HDFS to place data to load such that a map-only job can do the load. In fact processing, we proposed to use distributed lookup indices for multi-way lookups, and a hybrid approach to achieve efficient retrieval of dimension key values. We conducted extensive experiments to evaluate CloudETL and compared with the similar works ETLMR and Hive. The results showed that CloudETL achieves better performance than ETLMR when processing different dimension schemas and outperforms the dimensional ETL capabilities of Hive: It offers significantly better performance than Hive when processing different DW schemas and it is much easier to implement parallel ETL programs for the schemas.

There is a number of future research directions for this work. First, we plan to make CloudETL support more ETL transformation operators. Second, it would be interesting to consider more efficient backends allowing fast ETL processing. Last, it would be good to create a graphical user interface (GUI) where developers can "draw" an ETL flow by using visual transformation operators.

Chapter 6

Summary of Conclusions and Future Research Directions

This chapter summarizes the conclusions and directions for future work presented in Chapter 2-5, Appendix A and B.

6.1 Summary of Results

This thesis is about the data warehousing technologies for large-scale and right-time data. The work led to this thesis was primarily done in relation to the eGovMon project for which several aspects of the data warehousing technologies were proposed. The thesis has thus among other things presented the technologies to deal with several problems of the data warehousing, including how to handle OWL Lite data efficiently, how to handle the near real-time/right-time data, and especially, how to tackle the exponential growth of the data today. The developed technologies were all made to be general, and although the work was done in relation to the eGovMon project, the developed technologies can also be applied to the other environments. In the following, we go through each of the presented chapters and summarize the important results.

Chapter 2 addressed how to efficiently store and retrieve OWL Lite data. 3XL is a DBMS-based triple-store with a specialized data-dependent schema derived from an OWL Lite ontology. Unlike the traditional triple-store of saving triples in a narrow and giant table, 3XL stores triples in many class tables. In other word, 3XL performs the "intelligent partitioning" of the triples, which is efficiently used by the system to answer queries. In this chapter, we first described how to generate the database schema based on the ontology that describes the triples. We introduced the rules on how to map OWL classes, and properties, and the property restrictions to an object-relational database schema. We then presented a number of techniques to optimize the loading performance, including the in-memory value holders, partial-commit, using the main-memory Berkeley DB for caching map table, and bulk-load. We described the queries where the subject and/or the predicate is known, as we found such queries are the most important for most bulk data management applications. We also described the composite queries, and introduced how to make use of the *map* table to speed up query performance. We evaluated 3XL by the extensive experiments of using real-world EIAO data sets and LUBM benchmark, and compared with other similar works. The results showed that 3XL has loading and query performance comparable to the best file-based triple-store, BigOWLIM, and outperforms other DBMS-based solutions. At the same time, 3XL provides flexibility as it is DBMS-based and uses a specialized and intuitive schema to represent the data. 3XL thus bridges the gap between efficient representations and flexible and intuitive representations of the data. 3XL thus places itself in a unique spot in the design space for triple-stores.

Chapter 3 presented an ETL middle-ware system, All-RiTE, for live DW data. We introduced the supported operations of All-RITE for live DW data, including INSERT, UPDATE, DELETE and SELECT. We proposed a novel intermediate in-memory buffer *catalyst* to cache live DW data, and to perform the on-the-fly updates and deletions when the data in the catalyst is queried or materialized. We also presented a number of flush policies of defining *when* the data is available to external users, e.g., users can specify the time accuracy of the data read from the catalyst. In this chapter, we introduced dual data stores to optimize read/write performance of the data in the catalyst, and to manage the concurrency of read/write. We compared the supported operations with bulk loading, RiTE, traditional JDBC insert, JDBC batch and RDBMS selection. The results showed that in short transactions All-RiTE insertion is 4.7, 4.4, 9.2 and 2.4 times faster than bulk loading, RiTE, JDBC insert, and JDBC batch, respectively. This is an important contribution since short transactions are typically used for processing live DW data. Thus, with All-RiTE it is possible to insert data quickly and also make it available to consumers quickly. When performing the on-the-fly updates and deletions for the live DW data, All-RiTE works 4, 3.9, 8.9 and 3.6 times faster than bulk loading, RiTE, JDBC insert and JDBC batch, respectively. When selecting the full set of the rows of a table, RiTE is comparable to the select on an ordinary table in the DW. All-RiTE has better performance when selecting the the rows with the condition on a non-indexed column. Thus, the results indicates that All-RiTE provides a new “sweet spot” with the best of both worlds for live DW data: JDBC-like data availability but with bulk-load speed.

Chapter 4 presented ETLMR, a dimensional ETL programming framework using MapReduce. The presented framework supports the high-level ETL-specific constructs on MapReduce, including those for processing star schema, snowflake schema, SCDs, and data-intensive dimensions. We proposed a number of parallelization methods for processing different DW schemas on MapReduce, including one dimension one task (ODOT), one dimension all task (ODAT), level-wise and hierarchy-wise methods for snowflaked dimension tables. We also presented the offline dimension scheme for efficiently processing a big dimension table, in which the dimension data is partitioned, saved in each node locally and cached to main memory for lookups. This chapter also proposed the methods for processing fact data, and the method of parallel bulk-loading processed facts into the DW. The experiments first evaluated the scalability of processing different DW schemas, then compared with the other data warehousing tools using MapReduce including PDI v4.1. The results showed that ETLMR achieves good scalability, and the performance outperforms PDI significantly. This chapter also compared the programming effort of implementing a parallel program using ETLMR, Hive and Pig. ETLMR needs the least effort, e.g., it only needs 14 statements for a snowflake schema, while Hive and Pig need 23 and 40 statements, respectively. Therefore, ETLMR can achieve high programmer productivity for implementing the parallel ETL programs for different DW schemas.

Chapter 5 presented CloudETL, a scalable dimensional ETL framework for cloud warehouse. This chapter presented a novel data warehousing architecture for large-scale data, i.e., using Hadoop as the platform to parallelize ETL execution, and Hive as the warehouse to store data. CloudETL is applicable for the data warehousing in a cloud environment. This chapter detailed how to processing different DW schemas on Hadoop, and how to make use of Hive such as join in data warehousing. A novel approach was presented for doing updates for SCDs in a distributed environment. To process dimensions efficiently, this chapter first proposed the in-map updates for type-1 SCDs, then proposed the data co-location for very big dimension processing. In fact processing, this chapter presented the method of using lookup indices and using multi-way lookups. This chapter also presented the hybrid approach (using Hive joins and multi-way lookups) for fact processing with very big dimensions. In addition, this chapter introduced other technologies that are necessary for the data warehousing to Hive, including dimension key value generations, transformation operators and job planner. The experiments studied the performance of CloudETL, and compared with Hive and ETLMR. The results showed that CloudETL has better performance than ETLMR when processing scalable sizes of data, and outperforms the dimensional ETL capabilities of Hive. To process an identical DW schema, Hive requires much more programming effort and more jobs, e.g., for a type-2 SCD, CloudETL only needs 6-code lines and 1 job, while Hive requires 112 code lines and 5 jobs. The co-location of CloudETL was also compared with the co-partition of Hadoop++, and the results showed that CloudETL co-location was 2.2 times faster.

Appendix A is the supplement to the 3XL system in Chapter 2. It demonstrated how to use 3XL, including data dependent schema generation, triple loading, pointwise and composite queries. It also showed how to configure the run-time setting to tune the loading performance. Appendix B is the supplement to ETLMR in Chapter 2, which showed that it is very easy to implement parallel ETL programs for different DW schema using ETLMR. Appendix C compared the programming effort for a snowflake schema using ETLMR, Hive and Pig, which further shows that ETLMR is more efficient.

Data warehousing is a collection of decision support technologies, which enable enterprises to make better and faster decision, and to gain competitive advantage. Traditional technologies become increasingly challenging to meet the emerging requirements of data warehousing today. This thesis has considered several aspects of the current challenges and issues of data warehousing. The first proposed technology is a RDBMS-based triple-store for OWL Lite data. Since 3XL makes use of the object-relational features of PostgreSQL, i.e., inheritance, it give an intuitive representation of the triples stored in the RDBMS. 3XL not only has the performance offered by the file-based triple-stores, but also has the flexibility offered by the DBMS-based ones, e.g., very convenient to integrate with other non-OWL Lite data. 3XL is suitable for the projects which emphasize the efficiency of inserting and retrieving large amounts of triples (bulk operations), but without requiring the advanced features, such as logical inference etc.

The presented middle-ware system, All-RiTE, in Chapter 3 offers the capability of right-time and near real-time data integration. Traditionally, the data refreshment of DW proceeds in regular and off-line fashion. The right-time/near real-time data availability offered by All-RiTE enables enterprises to get the information quickly. Particularly, All-RiTE supports on-the-fly data modifications including UPDATE and DELETE. This enables to integrate live DW data in right-time/near real-time fashion, which is widely recognized as one of the most tricky problems in data warehousing. Since the data can be modified before, but not *after*, being loaded into the DW, it achieves much very good loading performance, e.g., the experiments showed that the performance even faster than bulk-load with deletions and updates. All-RiTE can be integrate with the ETL solutions which require right-time/near real-time date availability, or process live DW data.

The programming framework, ETLMR, presented in Chapter 4 facilitates to implement parallel dimensional ETL programs. Although ETL parallelization has long been studied, it is not easy to implement, such as using MPI, or it has certain restrictions, such as requiring using high-end machines. ETLMR considers these difficulties and restrictions. Thus, it provides the high-level ETL-specific constructs for quickly implementing parallel ETL programs for different schemas, and cost-effective scalability. The examples in Chapter 4 and Appendix B showed that it was very easy to implement a parallel ETL program using ETLMR, only a few code lines needed in configuration. Thus, ETLMR achieves both good scalability and programmer productivity.

The presented CloudETL in Chapter 5 supports scalable operations for cloud warehouse. The architecture of CloudETL differs dramatically from those of the in-house data warehousing systems. Due to its use of the cloud products, Hadoop and Hive, CloudETL is applicable to be deployed in a cloud environment for processing large-scale data for data warehousing. The idea of SCD updates is also applicable to the other works that require data modification operations in a distributed environment. The method of co-locating data in HDFS can also be applied to the other works which require some sets of data being processed together.

CloudETL and ETLMR both apply the cutting-edge cloud computing technology, MapReduce, to the parallelization of dimensional ETL. However, they differentiate in several aspects. First, they have different design purposes. ETLMR is designed for the traditional DBMS-based warehouse, while CloudETL is designed for the cloud warehouse, and CloudETL has much better scalability. Second, the parallelization methods in ETLMR are the foundation for the technologies used by CloudETL. In ETLMR, we emphasize the trials of the parallelization methods for different DW schemas, such as ODOT and ODAT. In CloudETL, however, we focus on using all the tasks but different optimization methods to improve the performance, including in-map updates, co-location and lookup indices. Third, they have different implementations, including MapReduce platforms and the programming languages. The experiences and lessons from ETLMR are taken into account during the implementation of CloudETL. For example, we introduce a metastore in CloudETL to maintain the global states of ETL. CloudETL introduces the key value generation service, and SCD updates to avoid the post-fixing of ETLMR.

In summary, the thesis has thus presented several aspects of data warehousing technologies of recent focus. The proposed technologies are designed to be general. They can be applied to various areas such as web data, large-scale data and right-time/near real-time data warehousing.

6.2 Research Directions

There are a number of future work for the technologies presented in this thesis. The open source technologies for BI has been evolving. Many volunteers participate in the development of open source projects around the world. However, in general the open source technologies for BI still lag behind the commercial counterparts for many years. There are still no complete solutions that can replace the commercial ones. It would be interesting to continue developing more open source software for BI. In addition, it would also be interesting to keep monitoring the evolution of the open source BI technologies. So, we expect in the near future complete open source BI solutions could be available to enterprises, organizations or even persons.

For the 3XL system, it would be interesting to integrate 3XL triple-store with All-RiTE, which builds a right-time triple warehousing solution. 3XL triple-store is thus used as the data source system, and the triples in 3XL are loaded into the DW by All-RiTE. According to the flush policies used, the warehousing system, thus, can offer right-time or near real-time data availability. It would be desirable to extend 3XL to support more or all the ontology constructs of OWL Lite, and to support an OWL/RDF query language that exploits the specialized database schema. As large amounts of web data are produced each day, and much of them is in OWL/RDF format, it is interesting to build a triple-store that uses distributed file system (DFS) to store large-scale OWL/RDF data, and uses MapReduce to parallelize loading and querying the triples.

Today, the Web increasingly becomes one of the most important sources for decision support systems. An increasing number of DW projects for web data appear. It is very attractive to implement a DW system that can easily integrate the dynamic data from the Web, such as using virtual dimension or fact tables over the web resources including RDF documents, XML documents, web services, etc. It would be interesting to represent the knowledge in the DW in a semantic way, i.e., RDF, such that users can do the reasoning and find new knowledge.

There are several future directions for Chapter 3. The first one is to support parallelization operation. As it is very common to process several tables together in data warehousing, and currently almost all the computers have multi-core CPUs, it would be interesting to extend All-RiTE to support the data-level parallelization, i.e., using multi-threading to process data into a single table. The current implementation, however, only supports loading multiple tables at the same time, each of which uses a separate thread to load data into a separate table. Second, it would be interesting to implement All-RiTE as a module for the host DBMS, instead of a middle-ware, such that it would provide the DBMS with the near real-time and right-time capabilities for live DW data. Third, more complete functionalities implemented for the catalyst are desirable, e.g., offers some DBMS functionalities, supports logging, and provides a domain language to operate the data in the catalyst, etc. Fourth, the current implementation is more or less coupled with the host DBMS, e.g., creates the MinMax table in the underlying DBMS, and uses a table function to read the data from the catalyst. It would be interesting to union the fresh data with the materialized data in the catalyst, instead of in the underlying DBMS.

For ETLMR described in Chapter 4, there are also some interesting directions for future work. First, it seems attractive to investigate how to avoid the *post-fixing* step that repairs the data inconsistent problem caused by the data processed by multiple parallel tasks. Second, it would be ideal to implement a global sequential number generator as we did in Chapter 5 such that ETLMR can be decoupled from the underlying DBMS, and be more portable for the use. Third, it would be interesting to extend ETLMR to support more complex ETL transformation operators, such as join, split, merge, etc.

Also with respect to CloudETL there are several future directions as follows. First, it would be desirable to implement more transformation operators for CloudETL. Second, it would be interesting to offer a tool for migrating the data from HDFS to RDBMS-based warehouse systems in parallel such that CloudETL can be integrated with the traditional DW systems. Third, it is also desirable to implement a domain specific language for the dimensional ETL operations on Hadoop. Last, as some ETL developers prefer to use a graphical user interface (GUI) for their development, it would be interesting to develop a GUI-based tool for CloudETL such that the developers can "draw" ETL flows by using the visualized transformation operators.

In the future, we will continue to focus our attention on the following aspects. First, with the exploding data growth today, enterprises are forced to process data quickly and efficiently, however, most of the existing data warehousing systems can not be scaled out cost-effectively. MapReduce for data warehousing deserves our further research, such as exploring its use for more complex DW schemas and ETL operations. Second, with the increasing popularity of cloud computing technology, it seems highly appealing to provide "out-of-the-box" BI solutions on the cloud such that organizations can build their BI systems quickly, with less effort, and at a potentially lower cost. Third, with the popularity of the social network and the widely use of portable devices, there would be an increasing need for the near real-time analysis on the data from hundreds, or even thousands of sources. It is thus interesting to investigate the technologies that support integrating data from a large number of data sources in near real-time/right-time fashion. Fourth, with the increasing diversity of the data sources today, data warehousing has to handle the unstructured data from many sources, including online surveys, Web forums, e-mail, etc. Since the data from these sources does not conform to any standard data models, and possibly has a big size, it is interesting to discover how to incorporate the unstructured data information into BI systems.

In conclusion, we see that data warehousing comprises a wide variety of interesting research topics, however, this thesis has only addressed several aspects. In the future, we will do scientific research on a wide range of topics related to data warehousing, and discover the new research opportunities.

Bibliography

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proc. of VLDB*, pp. 411–422, 2007.

[2] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[3] F. N. Afrati and J. D. Ullman. Optimizing Joins in a Map-reduce Environment. In *Proc. of EDBT*, pp.99-110, 2010

[4] S. Alexaki, V. Chrisophides, G. Karvounarakis, and D. Plexousakis. On Storing Voluminous RDF Descriptions: The Case of Web Portal Catalogs. In *Proc. of WebDB*, pp. 43–48, 2001.

[5] S. Alexaki, V. Chrisophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing Volumiunous RDF Description Bases. In *Proc. of ISWC*, pp. 1–13, 2001.

[6] Amazon Elastic Compute Cloud (Amazon EC2). Available at `aws.amazon.com/ec2` as of 2012-08-01[1].

[7] G. Antoniou and F. van Harmelen. A Semantic Web Primer. MIT press, 2004.

[8] Applications and Organizations Using Hadoop. Available at `wiki.apache.org/hadoop/PoweredBy` as of 2012-08-01.

[9] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons and R. Stoica. MaSM: Efficient Online Updates in Data Warehouses. In *Proc. of SIGMOD*, 2011.

[1] This bibliography has been created by merging the bibliographies from the individual papers that appear in this thesis. The dates given in “as of ...” for web resources have in general been updated, but for web resources that do not exist anymore or have had their content significantly updated, the date given in the original paper has been kept.

[10] B. Azvine, Z. Cui and D. Nauck. Towards Real-time Business Intelligence. *BT Technology Journal*, 23(3):214-225, 2005.

[11] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. OWL Web Ontology Language Reference, W3C Recommendation, 2004. Available at `w3.org/TR/REC-rdf-syntax` as of 2012-08-01.

[12] Berkeley DB - Oracle Embedded Database. Available at `oracle.com/us/products/database/berkeley-db` as of 2012-08-01.

[13] BigOWLIM - Semantic Repository for RDF(S) and OWL. Available at `www.ontotext.com/owlim/OWLIM_primer.pdf` as of 2012-08-01.

[14] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E.J. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proc. of SIGMOD*, pp.975-986, 2010.

[15] M. Bouzeghoub, F. Fabret, and M. Matulovic. Modeling Data Warehouse Refreshment Process as a Workflow Application. In *Proc. of DMDW*, 1999.

[16] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proc. of ISWC*, pp. 54–68, 2002.

[17] R. M. Bruckner, B. List and J. Schiefer. Striving Towards Near Real-Time Data Integration for Data Warehouses. In *Proc. of Dawak*, pp. 317–326, 2002.

[18] Cascading, `www.cascading.org` as of 2012-08-01.

[19] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2):1265–1276, 2008.

[20] S. Das, E. Chong, W. Zhe, M. Annamalai, and J. Srinivasan. A Scalable Scheme for Bulk Loading Large RDF Graphs into Oracle. In *Proc. of ICDE*, pp. 1297–1306, 2008.

[21] J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *CACM*, 53(1):72–77, 2010.

[22] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 1(51):107–113, 2008.

[23] D. DeWitt, E. Robinson, S. Shankar, E. Paulson, J. Naughton, A. Krioukov, and J. Royalty. Clustera: An Integrated Computation and Data Management System. *PVLDB*, 1(1):28–41, 2008.

[24] Disco project. Available at `discoproject.org` as of 2012-08-01.

[25] J. Dittrich, J. -A. Quiane-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *PVLDB*, 3(1):518–529, 2010.

[26] J. Dolby, A. Fokoue, A. Kalyanpur, A. Kershenbaum, E. Schonberg, K. Srinivas, and L. Ma. Scalable Semantic Retrieval Through Summarization and Refinement. In *Proc. of AAAI*, pp. 299–304, 2007.

[27] eGoverment Monitor. Available at `egovmon.no` as of 2012-08-01.

[28] EIAO Ontology. Available at `people.cs.aau.dk/~xiliu/3xlsystem/experiment/ontology/eiao.owl` as of 2012-08-01.

[29] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. *PVLDB*, 4(9):575–585, 2011.

[30] M. Forum. MPI 2.0 Standard. 1997.

[31] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions. *PVLDB*, 2(2):1402–1413, 2009.

[32] Gartner. Available at `www.gartner.com/it/page.jsp?id=1856714` as of 2012-08-01.

[33] L. Golab, T. Johnson and V. Shkapenyuk. Scheduling Updates in a Real-time Stream Warehouse. In *Proc. of ICDE*, pp. 1207–1210, 2009.

[34] Y. Guo, J. Heflin, and Z. Pan. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005

[35] Hadoop. `hadoop.apache.org` as of 2012-08-01.

[36] S. Han, D. Chen, M. Xiong, and A. K Mok. Online Scheduling Switch for Maintaining Data Freshness in Flexible Real-Time Systems. In *Proc. of RTSS*, pp. 115–124, 2009.

[37] S. Harris and N. Gibbins. 3Store: Efficient Bulk RDF Storage. In *Proc. of PSSS*, pp. 1–15, 2003.

[38] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proc. of EuroSys*, pp. 59–72, 2007.

[39] D. Jiang, A. K. H. Tung, and G. Chen. Map-join-reduce: Towards Scalable and Efficient Data Analysis on Large Clusters. *TKDE*, 23(9):1299–1311, 2010.

[40] D. Jiang, B.C. Ooi, L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3(1):472–483, 2010.

[41] T. Jörg and S. Dessloch. Near Real-time Data Warehousing Using State-of-the-art ETL Tools. In *Enabling Real-Time Business Intelligence*, 41:100-117, 2010.

[42] R. Kimball. Design Tip #57: Early Arriving Facts. `www.kimballgroup.com`, 2004.

[43] R. Kimball and W.H. Inmon. The Data Warehouse Toolkit. John Wiley: New York, 1996.

[44] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM – A Pragmatic Semantic Repository for OWL. In *Proc. of SSWS*, pp. 182–192, 2005.

[45] G. Kovoor, J. Singer, and M. Lujan. Building a Java MapReduce Framework for Multi-core Architectures. In *Proc. of MULTIPROG*, pp. 87–98, 2010.

[46] O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. Available at `w3.org/TR/REC-rdf-syntax` as of 2012-08-01.

[47] A. Labrinidis and N. Roussopoulos. A Performance Evaluation of Online Warehouse Update Algorithms. *Technical Report (CS-TR-3954)*, Dept. of Computer Science, University of Maryland, Nov. 1998.

[48] J. Lin and C. Dyer. Data-Intensive Text Processing with MapReduce. Morgan & Claypool Publishers, 2010.

[49] X. Liu, C. Thomsen, and T. B. Pedersen. CloudETL: Scalable Dimensional ETL for Hadoop and Hive. *Technical Report (TR-31)*, Dept. of Computer Science, Aalborg University, Available at `dbtr.cs.aau.dk/pub.htm` as of 2012-08-01.

[50] X. Liu, C. Thomsen, T. B. Pedersen, ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce. In *Proc. of Dawak*, pp. 96–111, 2011.

[51] LUBM queries. swat.cse.lehigh.edu/projects/lubm/query.htm as of 2012-08-01.

[52] H. P. Luhn. A Business Intelligence System. *IBM Journal of Research and Development*, 2(4):314-319, 1958.

[53] J. S. Narayanan and T. Kurc. DBOWL: Towards Extensional Queries on a Billion Statements Using Relational Databases. *Technical Report (2006)*, Available at bmi.osu.edu/resources/techreports/osubmi.tr.2006.n3.pdf as of 2012-08-01.

[54] G. Luo, J. F. Naughton, C. J. Ellmann and M. W. Waltzke. Transaction Reordering and Grouping for Continuous Data Loading. In *Proc. of BIRTE*, pp. 34-49, 2006

[55] T. Neumann and G. Weikum. RDF-3X: A RISC-style Engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[56] T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *Proc. of SIGMOD*, pp. 627–640, 2009.

[57] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proc. of SIGMOD*, pp. 1099–1110, 2008.

[58] Oracle Semantic Technologies Center. Available at oracle.com/technology/tech/semantic_technologies as of 2012-08-01.

[59] Z. Pan and J. Heflin. DLDB: Extending Relational Databases to Support Semantic Web Queries. In *Proc. of PSSS*, pp. 109–113, 2003.

[60] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In *Proc. of SIGMOD*, pp. 165–178, 2009.

[61] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Proc. of OSDI*, pp. 251–264, 2010.

[62] Pentaho. Available at www.pentaho.com as of 2012-08-01.

[63] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis and N. E. Frantzell. Supporting Streaming Updates in an Active Data Warehouse. In *Proc. of ICDE*, pp. 476–485, 2007.

[64] Press Releases. Available at www.gartner.com/it/page.jsp?id=1856714 as of 2012-08-01.

[65] G. Prud' Hommeaux, A. Seaborne, and others. SPARQL query language for RDF. *W3C Working Draft*, 2006.

[66] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proc. of HPCA*, pp. 13–24, 2007.

[67] Resource Description Framework (RDF) Model and Syntax Specification. Available at `www.w3.org/TR/PR-rdf-syntax` as of 2012-08-01.

[68] R. J. Santos and J. Bernardino, Real-time Data Warehouse Loading Methodology. In *Proc. of IDEAS*, pp. 49–58, 2008.

[69] J. Schiefer and R. M. Bruckner. Container-Managed ETL Applications for Integrating Data in Near Real-Time. In *Proc. of ICIS*, pp. 604–616, 2003.

[70] Shelve - Python object persistence. Available at `docs.python.org/library/shelve.html` as of 2012-08-01.

[71] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store Support for RDF Data Management: Not All Swans Are White. *PVLDB*, pp. 1553–1563, 2008.

[72] A. Simitsis, P. Vassiliadis and T. Sellis. Optimizing ETL Processes in Data Warehouses. In *Proc. of ICDE*, pp. 564–575, 2005.

[73] P. Snyder. tmpfs: A Virtual Memory File System. In *Proc. of EUUG*, pp. 241–248, 1990.

[74] SoftReference. Available at `download.oracle.com/javase/6/docs/api/java/lang/ref/SoftReference.html` as of 2012-08-01.

[75] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a Column-oriented DBMS. In *Proc. of VLDB*, pp. 553–564, 2005.

[76] M. Stonebraker, D. J. Abadi, D. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and Parallel DBMSs: friends or foes?. *CACM*, 53(1):64–71, 2010.

[77] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing, *Concurrency - Practice and Experience*, 2(4):315–339, 1990.

[78] The Apache Hadoop Project. Available at `hadoop.apache.org` as of 2012-08-01.

[79] The Java Database Connectivity (JDBC). Available at `www.oracle.com/technetwork/java/javase/jdbc/` as of 2012-08-01.

[80] C. Thomsen and T. B. Pedersen. A Survey of Open Source Tools for Business Intelligence. *Data Warehousing and Knowledge Discovery*, pp. 74–84, 2005.

[81] C. Thomsen and T. B. Pedersen. A Survey of Open Source Tools for Business Intelligence. *IJDWM*, 5(3):56–75, 2009.

[82] C. Thomsen and T.B. Pedersen. Building a Web Warehouse for Accessibility Data. In *Proc. of DOLAP*, pp. 43–50, 2006.

[83] C. Thomsen and T. B. Pedersen. Easy and Effective Parallel Programmable ETL. In *Proc. of DOLAP*, pp. 37–44, 2011.

[84] C. Thomsen and T. B. Pedersen. pygrametl: A Powerful Programming Framework for Extract-Transform-Load Programmers. In *Proc. of DOLAP*, pp. 49-56, 2009.

[85] C. Thomsen, T. B. Pedersen and W. Lehner. RiTE: Providing On-demand Data for Right-time Data Warehousing. In *Proc. of ICDE*, pp. 456–465, 2008.

[86] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution Over a Map-reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[87] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive-A Petabyte Scale Data Warehouse Using Hadoop. In *Proc. of ICDE*, pp. 996–1005, 2010.

[88] TPC-H. Available from `tpc.org/tpch/` as of 2012-08-01.

[89] R. Yoo, A. Romano, and C. Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-memory System. In *Proc. of IISWC*, pp. 198–207, 2009.

[90] P. Vassiliadis. A Survey of ExtractTransformLoad Technology. *IJDWM*, 5(3):1–27, 2009.

[91] P. Vassiliadis and A. Simitsis. Near Real Time ETL. *New Trends in Data Warehousing and Data Analysis*, pp. 1–31, Springer, 2009.

[92] P. Vassiliadis, Z. Vagena, S. Skiadopoulos and N. Karayannidis. ARKTOS: Towards the Modeling, Design, Control and Execution of ETL Processes. *Information Systems*, 26(8):537–561, 2001.

[93] H. Watson and B. Wixom. The Current State of Business Intelligence. *Computer* 40(9):96–99, 2007.

[94] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proc. of SWDB*, pp. 131–150, 2003.

[95] J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, and Y. Pan. Minerva: A Scalable OWL Ontology Storage and Inference System. In *Proc. of ASWC*, pp. 429–443, 2006.

Appendix A

3XL: An Efficient DBMS-Based Triple-Store

This demonstration presents *3XL*, a DBMS-based triple-store for OWL Lite data. 3XL is characterized by its use of a database schema specialized for the data to represent. The specialized database schema uses object-relational features – particularly inheritance – and partitions the data such that it is fast to locate the needed data when it is queried. Further, the generated database schema is very intuitive and it is thus easy to integrate the OWL data with other kinds of data. 3XL offers performance comparable to the leading file-based triple-stores.

We will demonstrate 1) how a specialized database schema is generated by 3XL based on an OWL ontology; 2) how triples are loaded, including how they pass through the 3XL system and how 3XL can be configured to fine-tune performance; and 3) how (simple and complex) queries can be expressed and how they are executed by 3XL.

A.1 Introduction

In recent years, the *Web Ontology Language*[1] (OWL), a semantic markup language recommended by W3C for publishing and sharing ontologies, has gained popularity. OWL is layered on top of the *Resource Description Framework*[1] (RDF). OWL (and RDF) data takes the form of (subject, predicate, object) triples. These triples are typically stored in specialized storage engines called *triple-stores*. We have seen that in some projects, the triple-stores are used mainly as specialized *bulk data stores*,

[1] `www.w3.org/TR`

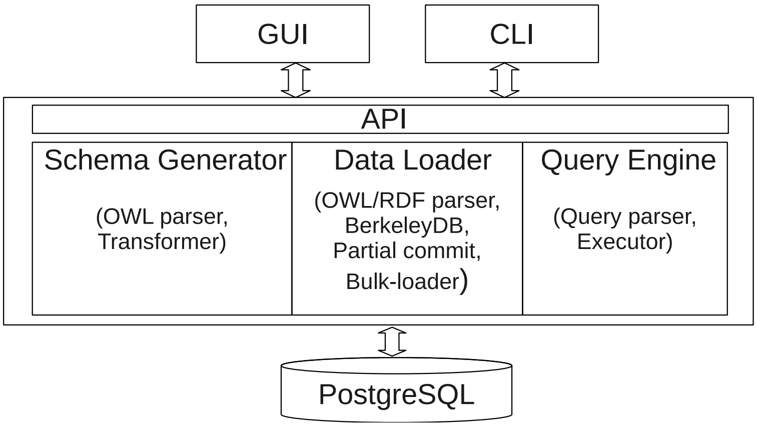

GUI
CLI
API
Schema Generator
(OWL parser, Transformer)
Data Loader
(OWL/RDF parser, BerkeleyDB, Partial commit, Bulk-loader)
Query Engine
(Query parser, Executor)
PostgreSQL

Figure A.1: 3XL Architecture

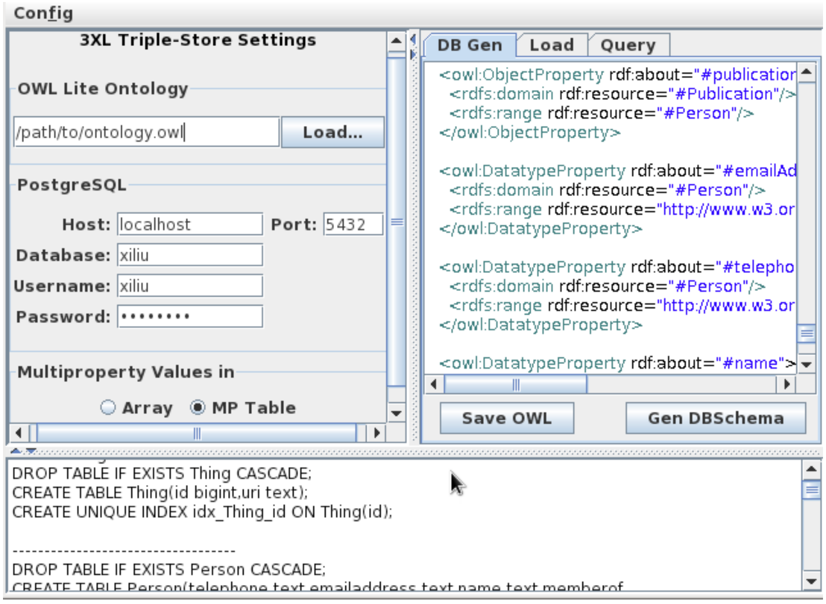
i.e., for inserting and retrieving large amounts of triples (bulk operations). More advanced features such as logical inference, etc., are often not used in such projects. Additionally, for basic representation of OWL instances, we found that even a subset of the least expressive OWL layer (OWL Lite) was enough. A well-known example of such instances is the data generated by the data generator for the de-facto industry standard OWL benchmark Lehigh University Benchmark (LUBM) [34].

This demonstration shows 3XL (see Chapter 2), a triple-store offering highly efficient loading and retrieval for OWL Lite data with a known ontology using a subset of OWL Lite. 3XL has a number of unique characteristics:

- 3XL is DMBS-based which makes it flexible and easy to integrate the OWL data with other data.
- Based on an OWL ontology, 3XL generates a specialized and intuitive database schema which intelligently partitions the data.
- 3XL uses object-relational features of the DBMS.
- Caching and bulk loading is intensively used in the implementation such that 3XL offers performance comparable to state-of-the-art file-based triple-stores.

This combination of efficiency and flexibility positions 3XL in a unique spot.

Figure A.1 shows the major components of 3XL and their interactions. 3XL consists of GUI and command line (CLI) interfaces, an API, a database schema generator, a data loader, a query engine, and an underlying DBMS (PostgreSQL). The database schema generator is responsible for parsing an OWL Lite ontology to create a hierarchical object-relational database schema according to a number of mapping



OWL Lite Ontology

/path/to/ontology.owl

PostgreSQL

Host: localhost Port: 5432
Database: xiliu
Username: xiliu
Password: ••••••••

Multiproperty Values in

○ Array ● MP Table

```
<owl:ObjectProperty rdf:about="#publicatior
  <rdfs:domain rdf:resource="#Publication"/>
  <rdfs:range rdf:resource="#Person"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:about="#emailAd
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="http://www.w3.or
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#telepho
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="http://www.w3.or
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#name">
```

```
DROP TABLE IF EXISTS Thing CASCADE;
CREATE TABLE Thing(id bigint,uri text);
CREATE UNIQUE INDEX idx_Thing_id ON Thing(id);

----------------------------------
DROP TABLE IF EXISTS Person CASCADE;
CREATE TABLE Person(telephone text,emailaddress text,name text,memberof
```

Figure A.2: 3XL's GUI for database schema generation

rules. The data loader parses OWL data and inserts the data into the database. To speed up the data loading, it uses several cache and bulk schemes, including an embedded instance of BerkeleyDB. The query engine has a query parser and an executor which are used to generate SQL and run the SQL in the underlying database, respectively. In the demonstration, we use the GUI, but we note that a client application also could use the API.

A.2 Specialized Schema Generation

Figure A.2 shows the interface for database schema generation. In this interface, the user can load and/or edit an OWL ontology. Further, the connection to PostgreSQL can be set up and the support for "multiproperty values" (to be explained later) can be configured. Based on the ontology (shown to the right in Figure A.2), a specialized database schema with an inheritance layout is generated by 3XL. We show how this is done by means of a running example.

Figure A.3 shows our example's small OWL Lite ontology which is inspired by the LUBM ontology. For brevity, it is shown as a graph. The ontology defines classes (shown as ellipses), properties (shown as rectangles), and their relationships (shown as labelled edges). In our example, each student has exactly one email address and one name and each course has exactly one name. A student can take several courses.

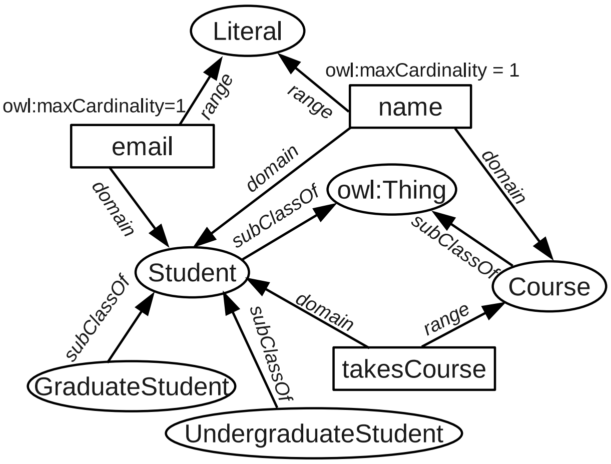


Figure A.3: OWL Lite schema for the running example

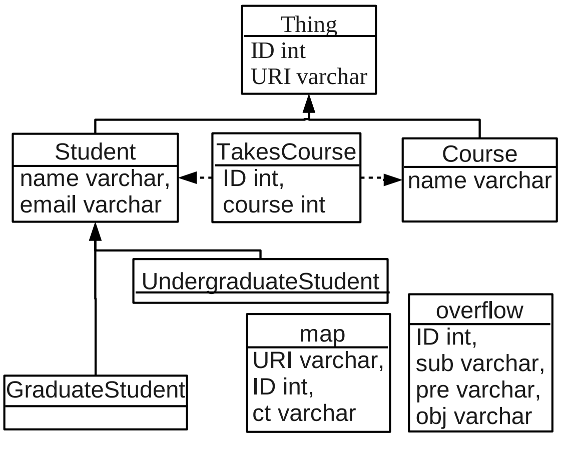


Figure A.4: 3XL database schema

When creating a database schema, 3XL creates a *class table* for each class in the ontology. Class tables have columns for those properties that have owl:maxCardinality 1. Figure A.4 shows the resulting database schema for our example and it can, for example, be seen that a class table is generated for Student. The Student class table has columns (name and emailAddress, both of type varchar) to represent the literal values of Student's data properties. When storing data, each Student instance results in a row in the Student class table.

The class table for a given class *inherits* the class table for the parent of the represented class. In Figure A.4 it can be seen that the class tables for UndergraduateStudent and GraduateStudent inherit from the class table for Student. When reading from the Student class table, PostgreSQL also includes data from the inheriting class tables. This represents that an UndergraduateStudent also is a Student (and similarly for GraduateStudent). In OWL, every class is (explicitly or implicitly) a subclass of owl:Thing. Therefore, a class table representing owl:Thing is always present in a generated database schema. The class table for owl:Thing has two columns: ID (of type int) and URI (of type varchar) such that each of its descendants (i.e., each class table) at least has these two columns. The column URI represents the URI of the represented instance while ID represents a unique numerical identifier for the instance assigned by 3XL.

Data properties in the ontology result in columns of an appropriate type in the database schema (e.g., varchar for literal values as previously shown). For object properties, 3XL creates a column of type int. This column holds the IDs of the referenced objects and acts like a foreign key. However, the foreign key is not declared in the database schema and we thus denote the column as a *loose foreign key*.

In OWL Lite, some properties do not have maxCardinality 1. We refer to such properties as *multiproperties*. In the database schema, a multiproperty can be represented in two ways: By a column of an *array* type in the class table or by a special table called a *multiproperty table*. When an array type is used, it is possible to represent several values in a single column of a row. As can be seen in Figure A.2, it is possible to chose from the GUI how to represent multiproperties. In this example, we use multiproperty tables. As a student can take several courses, the takesCourse property is represented by means of a multiproperty table in Figure A.4. This multiproperty table holds a row for each value the multiproperty takes for a given instance. ID in the table for takesCourse is a loose foreign key to Student to represent which instance the value belongs to. As takesCourse also is an object property, a value of the property is represented by a loose foreign key to the class table for Course. If it had been a data property, a column of the appropriate type would have been used.

Based on the mapping rules (see Chapter 2) summarized in Table A.1, 3XL generates DDL as shown in the bottom of Figure A.2. We emphasize how easy it is to understand and use a generated schema as the one in Figure A.4 with plain SQL (although queries by means of triples also are supported as demonstrated next). This makes it very easy to integrate the OWL data with other data.

Table A.1: Transformation rules

OWL constructs	**Results in database schema**
owl:Class	a class table
rdfs:subClassOf	inheritance between two class tables
owl:ObjectProperty or owl:DatatypeProperty	a column in a class table if owl:maxCardinality is 1, and in a multiproperty table or array column otherwise
rdfs:domain	a column in the class table if the maxCardinality is 1, and a loose foreign key from a multiproperty table to a class table otherwise
rdfs:range	a type for the column representing the property

Besides class tables and multiproperty tables, 3XL always creates the table `map (URI, ID, ct)` which makes it fast to find the ID and/or class table representing an instance with a given URI. Although 3XL is specialized for data with known ontologies, it also supports the standard *open world assumption* by creating the table `overflow(ID, sub, pre, obj)` which is used to hold triples that are not described by the ontology.

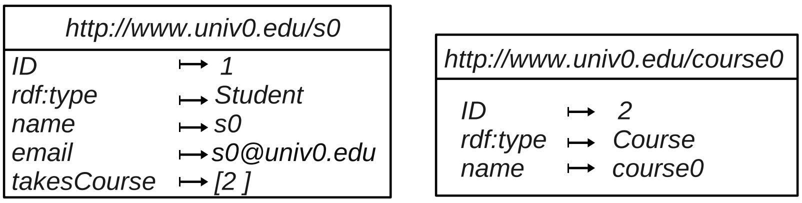
A.3 Triple Loading

We demonstrate how data is loaded into the 3XL system and consider the following triples:

```
(http://www.univ0.edu/s0, rdf:type, Student)
(http://www.univ0.edu/s0, name, "s0")
(http://www.univ0.edu/s0, emailAddress, "s0@univ0.edu")
(http://www.univ0.edu/s0, takesCourse, http://www.univ0.edu/course0)
(http://www.univ0.edu/course0, rdf:type, Course)
(http://www.univ0.edu/course0, name, "course0")
```

Data from triples with a common subject is first gathered in a *value holder* that maps from predicates to values. A value holder holds all known data about a specific individual. In the shown triples, there are two unique subjects. Thus, 3XL creates the two value holders shown in Figure A.5.

Note that each value holder also represents a unique ID which is assigned by 3XL. Further, the rdf:type is represented. It is required in OWL Lite that the rdf:type is explictly given to declare the class of an instance. In many cases, 3XL can, however, also *deduce* the rdf:type based on the seen predicates. For example, only students have email addresses in our scenario. Multiproperty values, as takesCourse in the



http://www.univ0.edu/s0	
ID	↦ 1
rdf:type	↦ Student
name	↦ s0
email	↦ s0@univ0.edu
takesCourse	↦ [2]

http://www.univ0.edu/course0	
ID	↦ 2
rdf:type	↦ Course
name	↦ course0

Figure A.5: Value holders

example, are represented as lists in value holders. Since takesCourse is an object property, 3XL's IDs of the referenced instances are held in the list instead of the (longer) URIs.

A value holder eventually results in a row in a class table and possibly some rows in multiproperty tables. The value holder to the left in Figure A.5 results in a row in the class table for Student and a row in the multiproperty table for takesCourse. For efficiency reasons, 3XL only updates the underlying database in bulks. Thus, many value holders are held in a data buffer. When the data buffer is full, the value holders in the data buffer are transferred to the underlying database in a bulk operation (instead of using the slower INSERT SQL statements). This is shown in the upper part of Figure A.6.

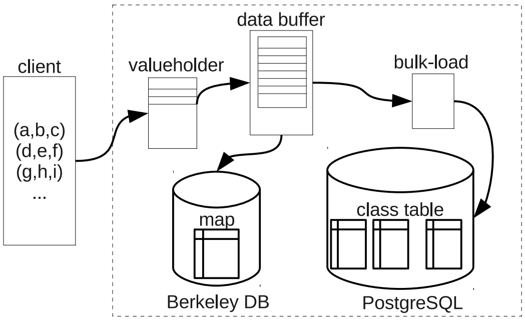


Figure A.6: Data flow in triple loading

When a value holder's data is inserted into the database, the value holder is deleted from the data buffer. When another triple is added, it can, however, happen that its corresponding value holder was just loaded into the database such that 3XL has to re-generate this value holder from the database. To avoid this, only the least-recently-used value holders are loaded into the database. We call this *partial commit*. This exploits that the data can have locality such that triples describing the same instance appear close to each other. For example, it is likely that the triples we insert in our scenario are more or less sorted on student name and/or course name.

As will be shown, the percentage of the data buffer to load into the database is configurable.

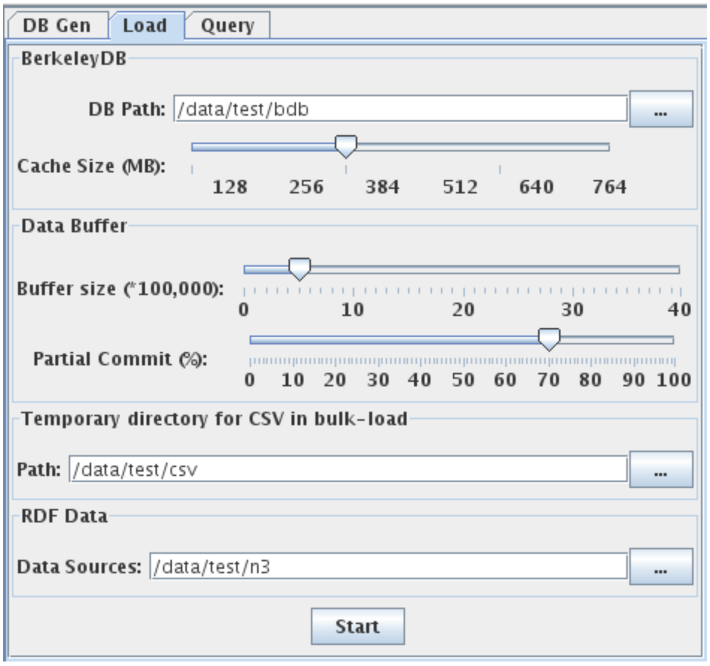
When the first triple in our example is seen by 3XL, there is no value holder for `http://www.univ0.edu/s0`. If a corresponding value holder for a new triple is not found in the data buffer, 3XL tries to look it up in the database. In our example, there is also no information about `http://www.univ0.edu/s0` in the database when 3XL sees the first triple. This is, however, difficult to determine as 3XL after seeing the first triple still does not know what type the instance has (and thus the appropriate class table is not known). Instead of searching all class tables, which would be very expensive, the `map` table is used. This table maps from a URI to the ID assigned by 3XL and the class table holding data about the instance (i.e, from `http://www.univ0.edu/s0` to the ID 1 and the class table Student in our case). For better performance, this table is loaded into BerkeleyDB which can keep a configurable amount of data in main memory. It is then very fast to determine which class table to look into (if the instance is already represented) or to decide that a new empty value holder should be created and no database lookup is necessary (if the instance is not already represented). For details about how bulk loading, partial commit, and use of BerkeleyDB for `map` influence the performance, see Chapter 2.

3XL can also load triples that are not described by the OWL Lite ontology used for the schema generation. If we, e.g., load the triple `(http://www.univ0.edu/s0, supervisor, http://www.univ0.edu/prof0)`, the data will be represented in the `overflow` table.

Figure A.7 shows the interface for configuring the loading of triples into 3XL. Triples in the N3-format can be loaded from a file using this interface. It is possible to configure the amount of memory used by BerkeleyDB. Further, the size of the data buffer can be set as well as the percentage of least-recently-used value holders to insert into the database when the data buffer is full. It is thus possible to fine-tune 3XL's performance. Details about the configuration parameteres are available in Chapter 2 and will be discussed during the demonstration. With proper configuration, 3XL achieves load-speeds of up to 25,000 triples/second on a normal notebook (see Chapter 2) which is comparable to the leading file-based triple-stores BigOWLIM [13] and RDF-3X [55].

A.4 Triple Queries

We demonstrate how the triple-store can be queried. Figure A.8 shows the interface for querying. A query can be entered in the upper part of the window and the result can either be written to a file or shown in the lower part of the window. 3XL supports two classes of queries: point-wise and composite queries. They are described in the following.



DB Path: /data/test/bdb

Cache Size (MB):

Buffer size (*100,000):

Partial Commit (%):

Path: /data/test/csv

Data Sources: /data/test/n3

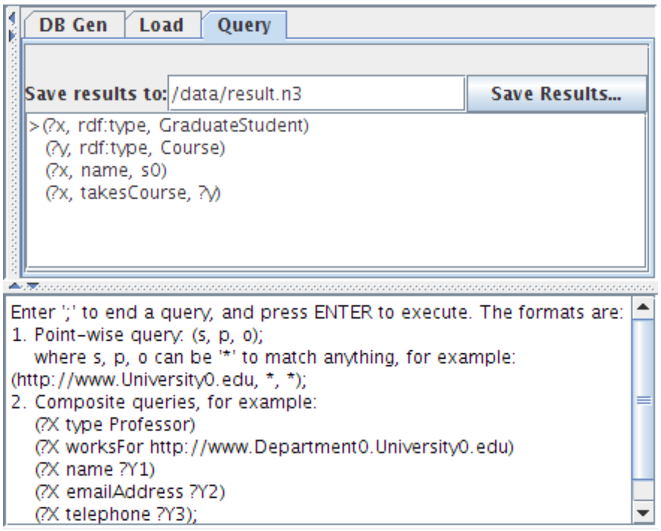
Figure A.7: 3XL's GUI for triple loading

A.4.1 Point-wise Queries

A *point-wise query* is a single triple, i.e., $Q = (s, p, o)$. The result includes all triples that have identical values for s, p, and o. Each part of the query triple may, however, be a wildcard "*" which is considered to be identical to everything. If we issue the query `(http://www.univ0.edu/s0, takesCourse, ⋆)` on the previously loaded triples, the result is `http://www.univ0.edu/s0`, `takesCourse`, `http://www.univ0.edu/course0)`.

To answer such queries, 3XL first identifies the relevant class tables and the IDs of the instances by means of the `map` table. In our example, 3XL thus finds the ID 1 for the instance. The class table for Student does, however, not hold the information needed to answer the query. The reason is that takesCourse is a multiproperty and we use multiproperty tables. As takesCourse also is an object property, 3XL has to join with the class table of takesCourse's range (i.e., Course) to find the URIs of the courses taken by the represented student:

```
SELECT Course.URI FROM TakesCourse, Course
WHERE TakesCourse.courseId = Course.ID
AND TakesCourse.studentID = 1
```



Save results to: /data/result.n3

```
>(?x, rdf:type, GraduateStudent)
 (?y, rdf:type, Course)
 (?x, name, s0)
 (?x, takesCourse, ?y)
```

```
Enter ';' to end a query, and press ENTER to execute. The formats are:
1. Point-wise query: (s, p, o);
   where s, p, o can be '*' to match anything, for example:
(http://www.University0.edu, *, *);
2. Composite queries, for example:
   (?X type Professor)
   (?X worksFor http://www.Department0.University0.edu)
   (?X name ?Y1)
   (?X emailAddress ?Y2)
   (?X telephone ?Y3);
```

Figure A.8: 3XL's GUI for triple querying

From the result of this SQL query, 3XL generates the triple result set – which in this case consists of a single triple. For the query `(http://www.univ0.edu/s0, emailAddress, *)` it is enough to identify the relevant class table (by means of the `map` table) and then select the email address from that class table. The reason is that emailAddress has maxCardinality 1 and is a data property. These examples illustrate how 3XL efficiently answers the important category of $(s, p, *)$ queries. For details about these and other point-wise queries, see Chapter 2.

It should be noted that when a query is made on a table with inheriting tables, the inheriting tables are queried as well. If, for example, a query is made on the Student table in our case, the UndergraduateStudent and GraduateStudent tables are also queried due to PostgreSQL's object-relational features.

A.4.2 Composite Queries

3XL also supports *composite queries*. Composite queries consist of several query triples and are more expressive than point-wise queries. Unknown *variables* used for *linking* triples together are specified using a string starting with a question mark while known *constants* are expressed using their full URIs or in an abbreviated form where prefixes can be replaced with shorter predefined values. The upper part of Figure A.8 shows an example of a composite query to find the courses taken by a student with a certain name. Figure A.9 shows the result as a graph. The result consists of all triples as those in the query, but with variable names replaced by actual values.

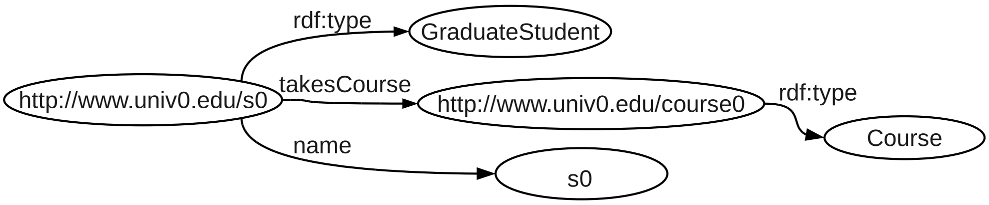


Figure A.9: The result of the composite query

To answer a composite query, 3XL generates more complex SQL than for the point-wise queries. For the shown example, the following SQL is enough. 3XL can then generate all the triples of the result.

```
SELECT GraduateStudent.uri AS x, Course.uri AS y
FROM GraduateStudent, TakesCourse, Course
WHERE GraduateStudent.ID = TakesCourse.ID AND
      TakesCourse.takesCourse = Course.id AND
      GraduateStudent.name = 's0'
```

For details, see Chapter 2 which also presents a performance study showing that 3XL's query performance is comparable to the leading file-based triple-stores BigOWLIM [13] and RDF-3X [55] on a big real-world data set as well as a big synthetic LUBM-based data set. Generally, file-based triple-stores are considered to be faster than DBMS-based ones. It is thus remarkable that 3XL offers the both of best worlds: performance *and* flexibility.

Appendix B

MapReduce-based Dimensional ETL Made Easy

This demonstration presents *ETLMR*, a novel dimensional Extract–Transform–Load (ETL) programming framework that uses MapReduce to achieve scalability. ETLMR has built-in native support of data warehouse (DW) specific constructs such as star schemas, snowflake schemas, and slowly changing dimensions (SCDs). This makes it possible to build MapReduce-based dimensional ETL flows very easily. The ETL process can be configured with only few lines of code. We will demonstrate the concrete steps in using ETLMR to load data into a (partly snowflaked) DW schema. This includes configuration of data sources and targets, dimension processing schemes, fact processing, and deployment. In addition, we also present the scalability on large data sets.

B.1 Introduction

In data warehousing, ETL flows are responsible for collecting data from different data sources, transformation, and cleansing to comply with user-defined business rules and requirements. Current ETL technologies are demanded to process many gigabytes of data each day. The vast amount of data makes ETL extremely time-consuming. The use of parallelization technologies is the key to achieve better ETL scalability and performance. In recent years, the “cloud computing” technology MapReduce [22] has been widely used for parallel computing in data-intensive areas due to its good scalability. We see that MapReduce can be a good foundation for ETL parallelization. The ETL processing exhibits the *composable* property such that the processing of dimensions or facts can be split into smaller computations and the partial results from these computations can be merged to constitute the final results in a DW. Further, the MapReduce programming paradigm is very powerful and flexible. MapReduce makes it easier to write a distributed computing program by providing interprocess communication, fault-tolerance, load balancing, and task scheduling. However, MapReduce is a *general* framework and lacks support for high-level ETL-specific constructs such as star and snowflake schemas, SCDs, etc. This results in low ETL programmer productivity. To implement a parallel dimensional ETL program on MapReduce is thus still very costly and time-consuming due to the inherent complexities of ETL-specific activities when processing dimensional DW schemas with SCDs etc.

In this demonstration, we present the MapReduce-based framework *ETLMR* (see Chapter 4). ETLMR directly supports high-level ETL-specific constructs on fact tables and dimensions (including SCDs) in both star schemas and snowflake schemas. A user can implement parallel ETL programs by using these constructs without knowing the details of the parallel execution of the ETL processes. This makes MapReduce-based ETL very easy. It reduces tedious programming work such that the user only has to make a configuration file with few lines of code to declare dimension and fact objects and the necessary transformation functions. ETLMR achieves this by using and extending pygrametl [84], a Python-based framework for easy ETL programming. Figure B.1 shows parallel ETL using ETLMR. The ETL flow consists of two sequential phases: dimension processing and fact processing. Data is read from the sources, i.e., files on a distributed file system (DFS), transformed, and processed into dimension values and facts by parallel ETLMR instances which consolidate the data in the DW. To make a parallel ETL program, only few lines of code declaring target tables and transformation functions are needed.

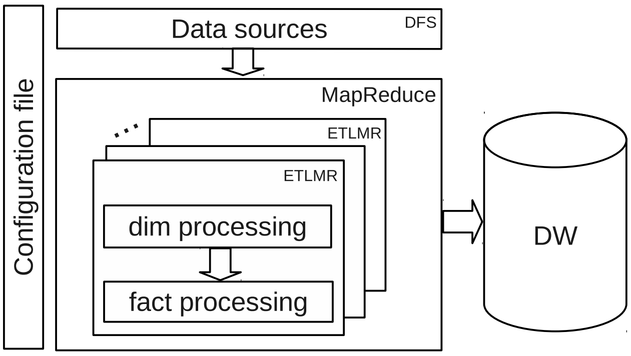


Figure B.1: Parallel ETL using ETLMR

In this demonstration, we will focus on a complete scenario where we process data for a partly snowflaked schema which also has an SCD. We demonstrate the configuration of data sources and targets, three dimension processing schemes, and fact processing as well as deployment and scalability. For a full description of the research challenges in making ETLMR, we refer to Chapter 4.

B.2 Sources and Targets

Throughout the demonstration, we use a running example inspired by a project which tests web pages with respect to accessibility (i.e., the usability for disabled people) and conformance to certain standards. Each test is applied to all pages. For each page, the test outputs the number of errors detected, and the results are written to a number of tab-separated files which form the data sources. The data is split into approximately equal-sized files and uploaded to the DFS. The files are located by ETLMR through URLs.

```
# Define the sources in the main program paralleletl.py:
fileurls = ['dfs://localhost/TestResults0.csv',
            'dfs://localhost/TestResults1.csv',  ...]
```

Lines from the input files are read into Python dictionaries for manipulation in ETLMR. Here, we call them *rows* and they map attribute names to values. An example of a row is

```
row={'url':'www.dom0.tl0/p0.htm', 'size':'15998',
        'serverversion':'1.0', 'downloaddate':'2011-01-31',
        'lastmoddate':'2011-01-01', 'test':'Test001', 'errors':'7'
     }
```

Figure B.2 shows the partly snowflaked target schema (ETLMR supports star and snowflake schemas, and their combinations). The schema comprises *testdim*, *datedim*, five snowflaked *page* dimension tables, and the fact table *testresultsfact*. *pagedim* is an SCD. The declarations of the dimension tables are seen in the following (we show the declaration of the fact table in Section B.4).

```
# Declared in the configuration file, conFigurepy
from odottables import *

# Declare the dimensions:
testdim=CachedDimension(name='test',key='testid',defaultid=-1,
        attributes=['testname'], lookupatts=['testname'])

datedim = CachedDimension(name='date',key='dateid',
        attributes=['date', 'day', 'month', 'year', 'week', 'weekyear'],
        lookupatts=['date'])

# Declare the dimension tables of the normalized pagedim.
```

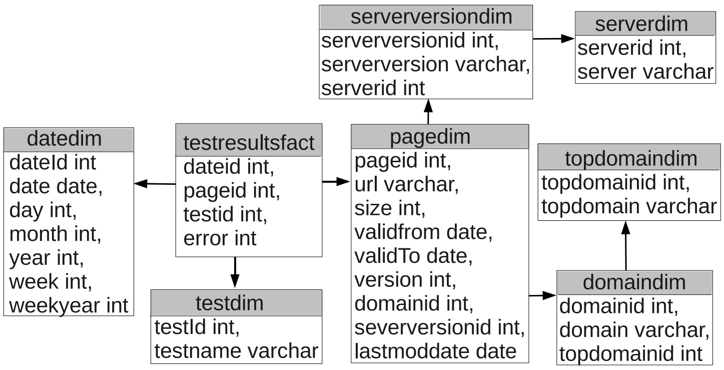


Figure B.2: The running example

```
pagedim = SlowlyChangingDimension(name=’page’,
        key=’pageid’, lookupatts=[’url’], attributes=[’url’,
        ’size’,’validfrom’,’validto’,’version’,’domainid’,  ’serverversionid’],
        versionatt=’version’, srcdateatt = ’lastmoddate’, fromatt=’validfrom’,
        toatt=’validto’,srcdateatt=’lastmoddate’)

topdomaindim=CachedDimension(name=’topdomaindim’,
        key=’topdomainid’,attributes=[’topdomain’], lookupatts=[’topdomain’])

domaindim = CachedDimension(name=’domaindim’,key=’domainid’,
        attributes=[’domain’,’topdomainid’],lookupatts=[’domain’])

serverdim = CachedDimension(name=’serverdim’,key=’serverid’,
        attributes=[’server’], lookupatts=[’server’])

serverversiondim = CachedDimension(name=’serverversiondim’,
        key=’serverversionid’,attributes=[’serverversion’,’serverid’],
        lookupatts=[’serverversion’], refdims=[serverdim])

# Define the references in the snowflaked dimension:
pagesf=SnowflakedDimension(
        (pagedim, (serverversiondim, domaindim)),
        (serverversiondim, serverdim), (domaindim, topdomaindim))
```

Different parameters are given when declaring a dimension table instance, including the dimension table name, the key column, and lists of attributes and lookup attributes (sometimes referred to as the “business key”). Besides, optional parameters can be given, such as a default value for the dimension key when a dimension value is not found in a lookup, e.g., `defaultid=-1` in the declaration of `testdim`. For an instance of `SlowlyChangingDimension`, additional SCD related parameters – such as the columns for the version number and the timestamps – are given. Note how easy it is to declare and use a snowflaked SCD in ETLMR. This is very complex with traditional tools. Other settings of the dimension tables for different processing schemes are discussed later.

B.3 Dimension Processing Schemes

ETLMR has several dimension processing schemes. We will demonstrate how to configure and choose the schemes.

B.3.1 One Dimension One Task

We first consider an intuitive approach to process dimensions in parallel, namely “one dimension, one task” (ODOT) where there is one (and only one) map/reduce task for each dimension table.

We first define the corresponding attributes of the data source for each dimension table (the `srcfields`), and the transformations for extracting values from the field values of a row (the `rowhandlers`). The user implements transformations as normal Python functions. The user-defined functions (UDFs) for transformations are not shown here for space reasons, but given self-explanatory names starting with UDF_.

```
# Defined in config.py
dims={
pagedim:{’srcfields’:(’url’,’serverversion’,’domain’,’size’, ’lastmoddate’),
       ’rowhandlers’:(UDF_extractdomain,UDF_extractserverver)},
domaindim:{’srcfields’:(’url’,),
       ’rowhandlers’:(UDF_extractdomain,)},
topdomaindim:{’srcfields’:(’uri’,),
       ’rowhandlers’:(UDF_extracttopdomain,)},
serverversiondim:{’srcfields’:(’serverversion’,),
       ’rowhandlers’:(UDF_extractserverver,)},
serverdim:{’srcfields’:(’serverversion’,),
       ’rowhandlers’:(UDF_extractserver,)},
datedim:{’srcfields’:(’downloaddate’,),
       ’rowhandlers’:(UDF_explodedate,)},
testdim:{’srcfields’:(’test’,), ’rowhandlers’:()}
}
```

As there are references between the tables of the snowflaked (normalized) dimension, the processing order matters and is specified in the following. It is illustrated in Figure B.3.

```
order=[(’topdomaindim’, ’serverdim’), (’domaindim’,
     ’serverversiondim’), (’pagedim’, ’datedim’, ’testdim’)]
```

With this order, ETLMR processes the dimensions from the leaves towards the root (the dimension table referenced by the fact table is the root and a dimension table without a foreign key to another dimension tables is a leaf). The dimension tables with dependencies (from foreign key references) are processed in sequential jobs, e.g., *Job1* depends on *Job0*, and *Job2* depends on *Job1*. Each job processes independent dimension tables by parallel tasks. Therefore, *Job0* first processes `topdomaindim` and `serverdim`, then *Job1* processes `domaindim` and

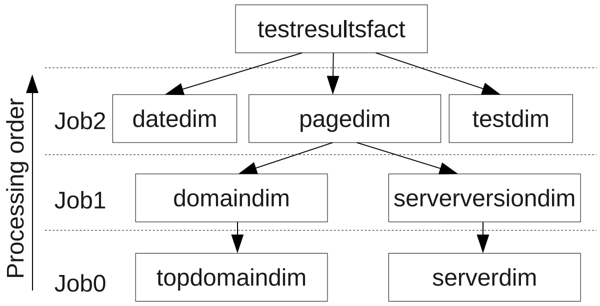


Figure B.3: Process snowflake schema

`serverversiondim`, and finally *Job2* processes `pagedim`, `datedim`, and `testdim`.

When processing, ETLMR does data projection in the mappers to select the necessary values for each dimension table. This results in key/value pairs of the form *(dimension table name, tuple of values)* that are given to the reducers. ETLMR partitions the map outputs based on the dimension table names such that the values for one dimension table are processed by a single reducer (see Figure B.4). For example, the reducer for `pagedim` receives among others the values `{'url':'www.dom0.tl0/p0.htm','serverversion':'1.0','size':'12','lastmoddate':'2011-01-01'}`. In the reducers, ETLMR automatically applies UDFs for transformations (if any) to each row. The row is then automatically ensured to be in the dimension table, i.e., ETLMR inserts the dimension value if it does not already exist in the dimension table, and otherwise updates it as needed. For a type-2 SCD (where versioning of rows is applied), ETLMR also adds a new version and updates the valid timestamps of the old version as needed. The programmer thus only has to program the transformations to apply and ETLMR takes care of the rest.

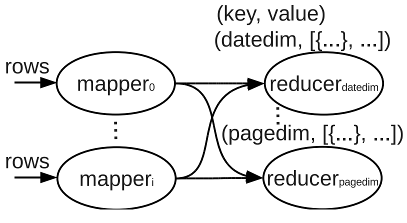


Figure B.4: ODOT

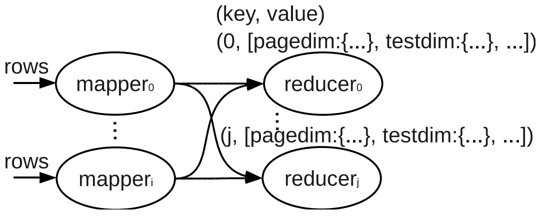

(key, value)
(0, [pagedim:{...}, testdim:{...}, ...])
rows
mapper$_0$
reducer$_0$
(j, [pagedim:{...}, testdim:{...}, ...])
rows
mapper$_i$
reducer$_j$

Figure B.5: ODAT

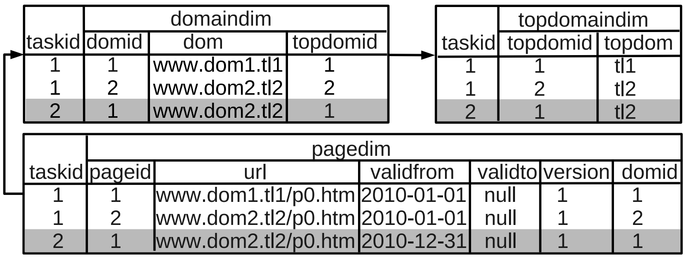
B.3.2 One Dimension All Tasks

We now consider another approach to process dimensions (see Figure B.5), namely “One dimension, all tasks” (ODAT). This approach makes use of all the reducers to process the map outputs, i.e., one dimension is processed by all tasks unlike ODOT where only a single task is utilized for a dimension. Thus ODAT can use more nodes and has better scalability. ETLMR has a separate Python module implementing this approach, and only a single line, `from odattables import *`, is needed to use it. As the ODAT dimension and fact classes have the same interfaces as the ODOT classes, nothing is changed in the declarations in Section B.2, except that the processing order is not needed any more.

With ODAT, the map output is partitioned in a round-robin fashion such that all reducers receive an almost equal-sized map output containing key/values pairs for all dimension tables (see Figure B.5). A reducer processes map output for all dimension tables. Therefore, a number of issues need to be considered, including the uniqueness of dimension keys, concurrency problems when different tasks are operating on the same dimension values to update the timestamps of SCD dimensions, and duplicated values of the same dimension. To remedy this, ETLMR automatically employs an extra step called *post-fixing* to fix the problematic data when the dimension processing job has finished. Figures B.6 and B.7 illustrate post-fixing. The details about the post-fixing steps can be found in Chapter 4.

Post-fixing *Consider two map/reduce tasks, task 1 and task 2, which process the snowflaked dimension* `page`*. Each task uses a private ID generator. The root dimension,* `pagedim`*, is a type-2 SCD. Both task 1 and task 2 process rows with the lookup attribute value url=’www.dom2.tl2/p0.htm’.*

Figure B.6 depicts the resulting data in the dimension tables. White rows were processed by task 1 and grey rows were processed by task 2. Each row is labelled with the `taskid` *of the task that processed it. The problems include duplicated IDs in each dimension table and improper values in the SCD attributes,* `validfrom`*,* `validto`*, and* `version`*. The post-fixing program first fixes the* `topdomaindim`

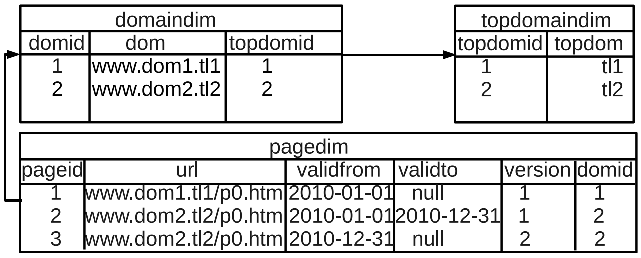


	domaindim		
taskid	domid	dom	topdomid
1	1	www.dom1.tl1	1
1	2	www.dom2.tl2	2
2	1	www.dom2.tl2	1

	topdomaindim	
taskid	topdomid	topdom
1	1	tl1
1	2	tl2
2	1	tl2

	pagedim					
taskid	pageid	url	validfrom	validto	version	domid
1	1	www.dom1.tl1/p0.htm	2010-01-01	null	1	1
1	2	www.dom2.tl2/p0.htm	2010-01-01	null	1	2
2	1	www.dom2.tl2/p0.htm	2010-12-31	null	1	1

Figure B.6: Before post-fixing



domaindim		
domid	dom	topdomid
1	www.dom1.tl1	1
2	www.dom2.tl2	2

topdomaindim	
topdomid	topdom
1	tl1
2	tl2

pagedim					
pageid	url	validfrom	validto	version	domid
1	www.dom1.tl1/p0.htm	2010-01-01	null	1	1
2	www.dom2.tl2/p0.htm	2010-01-01	2010-12-31	1	2
3	www.dom2.tl2/p0.htm	2010-12-31	null	2	2

Figure B.7: After post-fixing

such that rows with the same value for the lookup attribute are merged into one row with a single ID. Thus, the two rows with `topdom` *= tl2 are merged into one row. The references to* `topdomaindim` *from* `domaindim` *are also updated to reference the correct (fixed) rows. In the same way,* `pagedim` *is updated to merge the two rows representing www.dom2.tl2. Finally,* `pagedim` *is updated. Here, the post-fixing also has to fix the values for the SCD attributes.*

B.3.3 Offline Dimensions

ETLMR also has a module for processing dimensions which are stored on the nodes. We say such dimensions are *offline* as opposed to the previously described approaches where the dimensions reside in the DW database and are *online*. The interface of offline dimensions is similar to that of online dimensions except that there is an additional parameter, *shelvedpath*, to denote the local path for saving the dimension data. The following code snippet exemplifies a declaration:

```
from offdimtables import *
datedim = CachedDimension(
        name=’date’,
        key=’dateid’,
        attributes=[’date’,’day’,’month’,’year’,’week’,’weekyear’],
        lookupatts=[’date’],
        shelvedpath=’/path/to/datedim’  )
```

When offline dimensions are used, the map/reduce tasks do not interact with the DW by means of database connections and as the data is stored locally in each node, the network communication cost is greatly reduced. The dimension data is expected to reside in the nodes, and is not loaded into the DW until this is explicitly requested.

B.3.4 How to Choose

The ODOT scheme is preferable for small-sized dimensions when high scalability is not required. On the contrary, the ODAT scheme is preferable for dimension tables with big data volumes as the data can processed by all tasks which gives better scalability. When immediate data availability is not required, the offline dimension scheme can be chosen for better performance.

B.4 Fact Processing

We now demonstrate how to configure the fact processing. For our example, the fact table is declared as below:

```
# In config.py
# Declare the fact table (here we support bulk loading):
testresultsfact = BulkFactTable(name='testresultsfact',
     keyrefs=['pageid','testid','dateid'], measures=['errors'],
     bulkloader=UDF_pgcopy, bulksize=5000000)

# Set the referenced dimensions and the
# transformations to apply to facts:
facts = {testresultsfact:
              {'refdims':(pagedim, datedim, testdim),
               'rowhandlers':(UDF_convertStrToInt,)}
        }
```

It is, of course, also possible to declare several fact tables if needed, these can be processed in parallel. The declaration of a fact table includes the name of the fact table and the column names of the dimension referencing keys and measures. Here, the `BulkFactTable` class is used to enable bulk loading of facts. As the bulk loader varies from DBMS to DBMS, the user has to declare which function to call to perform the actual bulk loading. After the declaration, ETLMR must be configured to use the instance correctly. This involves specifying the dimension objects from which ETLMR looks up dimension keys and specifying the transformations ("`rowhandlers`") which ETLMR should apply to the facts.

When ETLMR processes the fact data, the data files are assigned to the map/reduce tasks in a round-robin fashion. Each task automatically does its work by applying the user-defined transformations to the rows from the data files, looking up dimension keys, and inserting the rows into a buffer. When the buffer is full, the data in the buffer is loaded into the DW by means of the bulk loader. This is again very easy for the user who just has to program transformations.

B.5 Deployment

ETLMR uses the Python-based Disco [24] as its MapReduce platform. The system uses a master/worker architecture with one master and many workers (or nodes). Each worker has a number of map/reduce tasks which run the ETLMR parts in parallel. The master is responsible for scheduling the tasks, distributing partitioned data, tracking status, and monitoring the status of workers. Each worker has a worker supervisor responsible for spawning and monitoring its tasks. If a worker crashes, the MapReduce framework automatically assigns the node's task to another node and thus provides us with restart and checkpointing capabilities.

To deploy ETLMR, a configuration module is placed on the master. This configuration defines dimension and fact tables. The distributed ETL program is then started by the following code which defines which node is the master, where the input is located, the name of the configuration module, and the numbers of mappers and reducers.

```
# Start the ETLMR main program, paralleletl.py:
ETLMR.start(master='masternode',inputs=fileurls,required_\
modules=[('config','config.py')], nr_maps=20, nr_reduces=20)
```

We note that the number of mappers and reducers can easily be changed in this program by only updating the `nr_maps` and `nr_reduces` arguments. This makes it very easy for the ETL developer to scale up/scale down ETL programs.

B.6 Scalability

We now present the scalability of ETLMR. Details about the used cluster are available in Chapter 4. Table B.1 shows the time of dimension processing when we use the ODAT and offline dimension processing scheme (the fastest). We use an 80 GB fixed-size data set for the running example represented in a star schema (with 13,918,502 rows in the page dimension). We scale the number of map/reduce tasks from 4 to 20. As the data is equally split and processed by all tasks, ETLMR achieves a nearly linear speedup in processing the big dimension. The speedup is *nearly* linear as the partitioning costs become more dominating when each map/reduce task gets less data and run for a shorter time. Further, the costs from the MapReduce framework (e.g., for communication) increase when more tasks are added.

Table B.1: The time of dimension processing

no. of tasks	4	8	12	16	20
Time (min)	260.95	135.65	91.39	70.73	55.22

We also consider another 80 GB data set which has small-sized dimensions (19,460 rows in the page dimension) to study the scalability. Figure B.8 shows ETLMR has a nearly linear speedup in the increasing number of tasks. When we use a fixed number of tasks (such that the MapReduce costs don't increase) and vary the size of the test data, the processing time grows linearly in the data size (see Figure B.9).

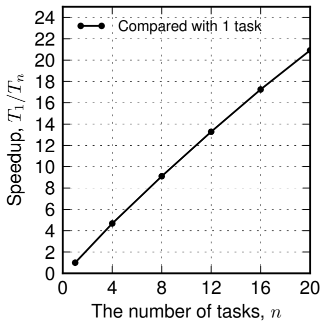


Figure B.8: Speedup with increasing tasks, 80GB

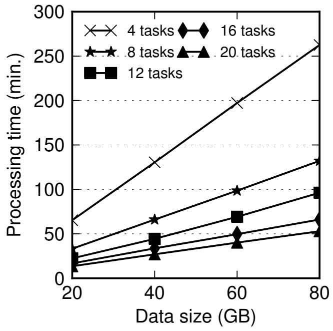


Figure B.9: Processing time when scaling up data size

More studies of the scalability can be found in Chapter 4 which also considers the use of ETLMR compared to doing ETL operations by means of the MapReduce tools Hive and Pig. As ETLMR is a specialized ETL tool, it is much simpler to create an ETL solution with ETLMR. Further, Chapter 4 compares the performance of ETLMR to the leading open source ETL tool PDI which also supports MapReduce. In the comparison, ETLMR is significantly faster. The licenses of commercial ETL tools do not allow us to publish performance results for the tools.

Appendix C

Summary in Danish / Dansk Resumé

Data Warehousing er en vigtig teknologi til informationsintegration og dataanalyse. På grund af den eksponentielle vækst af data i dag er det blevet en almindelig praksis for mange virksomheder at behandle hundredvis af gigabytes data per dag. I data warehousing indsættes data fra heterogene kilder traditionelt i et centralt data warehouse (DW) vha. en Extract-Transform-Load (ETL) proces med regelmæssige tidsintervaller, f.eks. månedligt, ugentligt eller dagligt. Men efterhånden bliver det udfordrende at behandle store datamængder og opfylde nær real-time/right-time forretningsmæssige krav. Denne afhandling omhandler nogle af disse nye udfordringer og problemstillinger, og har følgende bidrag:

For det første præsenterer denne afhandling 3XL, et OWL Lite triple-store. Triple-stores har i stigende grad brug for store operationer (såkaldte "bulk-operationer") til at indsætte og hente store mængder tripler effektivt. 3XL har et specialiseret database skema der er udledt fra en OWL Lite ontologi og gør brug af de objekt-relationelle egenskaber fra RDBMS, såsom nedarvning. I modsætning til det traditionelle triple-store med smalle og store tabeller til opbevaring af tripler, partitionerer 3XL triplerne på baggrund af OWL klasser, og gemmer dem i mange klassetabeller. 3XL gør indsætning af tripler effektiv vha. omfattende brug af bulk-teknikker, bl.a. in-memory buffer, commit af data i henhold til datalokalitet og bulk-loading. 3XL understøtter en effektiv hentning af tripler fra klassetabeller. 3XL har ikke kun god ydeevne (sammenlignelig med førende fil-baserede triple-stores, RDF-XL og BigOWLIM), men har også god fleksibilitet grundet brugen af RDBMS.

For det andet præsenterer denne afhandling All-RiTE, som er "right-time" ETL middle-ware til levende DW data. All-RiTE anvender en nyskabende mellemliggende in-memory databuffer til at akkumulere levende DW data, og kan foretage dataændringer i luften. Data fra kildesystemer overføres til den mellemliggende buffer og bliver sidenhen materialiseret til DW'et baseret på brugerdefinerede politikker. Dataene i den mellemliggende buffer, er imidlertid også tilgængelige for læsning med specificeret tidsnøjagtighed, før det materialiseres. De eksperimentelle resultater viser, at All-RiTE kombinerer det bedste fra standard JDBC og bulk-load: datatilgængeligheden af INSERTs, load-hastigheder hurtigere end bulk-loading, og meget effektive UPDATEs og DELETEs af levende DW data .

For det tredje foreslår denne afhandling det første dimensionelle ETL-programmeringsframework, der anvender MapReduce, kaldet ETLMR. ETLMR tilbyder højniveau ETL-specifikke konstruktioner til forskellige DW-skemaer, herunder stjerneskema, snowflakeskema, slowly changing dimensioner (SCDs) og meget store dimensioner. Dette framework giver forskellige paralleliseringsmetoder til behandling af data ind i et RDBMS-baseret DW. Det muliggør implementering af et parallelt ETL-program med frameworket kan man opnå høj programmeringseffektivitet, dvs, kun få kommandoer er nødvendige for at implementere et parallelt ETL-program med god skalerbarhed.

For det fjerde præsenterer denne afhandling et dimensionelt ETL-framework for cloud warehouses, kaldet CloudETL. I forhold til ETLMR udnytter CloudETL Hadoop til at parallelisere ETL-udførelsen, og processerer data ind i Hive-tabeller. Således understøtter CloudETL skalerbare operationer for både loading og analytiske forespørgsler på store data. Vi præsenterer en række nye dimensionelle ETL-teknologier til data warehousing på Hadoop, herunder understøttelse af opdateringer af SCDs, samplacering på Hadoop Distributed File System (HDFS), in-map opdateringer, opslagsindekser og multivejs opslag til fact-behandling. De eksperimentelle resultater viser at CloudETL har bedre ydeevne end ETLMR og overgår de dimensionelle ETL-muligheder, som Hive har, dvs. CloudETL har brug for mindre programmeringsindsats for at behandle et identisk DW skema, og ydeevnen er også meget bedre.

Sammenfattende diskuterer denne afhandling adskillige aspekter af de nuværende udfordringer og problemstillinger i data warehousing herunder integration af web data, næsten real-time/right-time data ware housing der håndterer eksponentiel vækst af data og data warehousing i skyen. Denne afhandling foreslår en række teknologier til at håndtere disse specifikke problemstillinger.