Incremental Recomputations in Materialized Data Integration

vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation von

Dipl.-Inf. Thomas Jörg

Dekan des Fachbereichs:
Prof. Dr. Arnd Poetzsch-Heffter

Berichterstatter:
Prof. Dr.-Ing. Stefan Deßloch
Prof. Dr.-Ing. Bernhard Mitschang

Datum der wissenschaftlichen Aussprache:
18.01.2013

D 386

Abstract

Data integration aims at providing uniform access to heterogeneous data, managed by distributed source systems. Data sources can range from legacy systems, databases, and enterprise applications to web-scale data management systems. The materialized approach to data integration, extracts data from the sources, transforms and consolidates the data, and loads it into an integration system, where it is persistently stored and can be queried and analyzed.

To support materialized data integration, so called Extract-Transform-Load (ETL) systems have been built and are widely used to populate data warehouses today. While ETL is considered state-of-the-art in enterprise data warehousing, a new paradigm known as MapReduce has recently gained popularity for web-scale data transformations, such as web indexing or page rank computation.

The input data of both, ETL and MapReduce programs keeps changing over time, while business transactions are processed or the web is crawled, for instance. Hence, the results of ETL and MapReduce programs get stale and need to be recomputed from time to time. Recurrent computations over changing input data can be performed in two ways. The result may either be recomputed from scratch or recomputed in an incremental fashion. The idea behind the latter approach is to update the existing result in response to incremental changes in the input data. This is typically more efficient than the full recomputation approach, because reprocessing unchanged portions of the input data can often be avoided.

Incremental recomputation techniques have been studied by the database research community mainly in the context of the maintenance of materialized views and have been adopted by all major commercial database systems today. However, neither today's ETL tools nor MapReduce support incremental recomputation techniques. The situation of ETL and MapReduce programmers nowadays is thus much comparable to the situation of database programmers in the early 1990s. This thesis makes an effort to transfer incremental recomputation techniques into the ETL and MapReduce environments. This poses interesting research challenges, because these environments differ fundamentally from the relational world with regard to query and programming models, change data capture, transactional guarantees and consistency models. However, as this thesis will show, incremental recomputations are feasible in ETL and MapReduce and may lead to considerable efficiency improvements.

Zusammenfassung

Datenintegration ermöglicht einen einheitlichen Zugriff auf heteregene Daten, die in verteilten Quellen vorliegen. Das Spektrum von Datenquellen reicht dabeit von Legacy-Systemen, Datenbanken und Geschäftsanwendungen bis hin zu Systemen zur Verwaltung sehr großer Datenmengen, wie sie im Web vorkommen. Bei der sogenannten materialisierten Datenintegration werden Daten aus den Quellen extrahiert, transformiert und konsolidiert, und in das Integrationssystem geladen, wo sie für Anfragen und Analysen zur Verfügung stehen.

Für die materialisierte Integration von Daten wurden sogenannte Extract-Transform-Load (ETL) Systeme entwickelt, die heute primär zum Laden von Data-Warehouse-Systemen eingesetzt werden. Für die Transformation von Daten, die um ein Vielfaches größer als klassische Geschäftsdaten sind, beispielsweise die Berechnung eines Web-Suchindexes, wird in jüngerer Zeit ein neues Paradigma names MapReduce eingesetzt.

Die Eingabedaten von ETL und MapReduce Programmen verändern sich kontinuierlich, daher muss ihr Ergebnis von Zeit zu Zeit neu berechnet werden. Wiederholte Berechnungen auf Grundlage sich verändernder Eingabedaten können auf zweierlei Arten durchgeführt werden. Die Berechung kann entweder erneut vollständig durchgeführt werden oder auf inkrementelle Weise erfolgen. Die Idee dabei ist es, ein zuvor berechnetes Ergebnis anhand der inkrementellen Änderungen der Eingabedaten anzupassen. Dieser Ansatz ist in der Regel effizienter, da sich eine erneute Verabeitung unveränderter Eingabedaten häufig vermeiden lässt.

Techniken zur inkrementellen Neuberechnungen wurden von der Datenbankgemeinde zur Wartung materialisierter Sichten untersucht und werden heute in kommerziellen Datenbanksystemen eingesetzt. Im Gegensatz dazu unterstützen weder ETL noch MapReduce inkrementelle Neuberechnungen. Die Situation von ETL und MapReduce Programmierern ist daher vergleichbar mit der Situtation von Datenbankprogrammierern in den 1990er Jahren. Ziel dieser Arbeit ist es Techniken zur inkrementellen Neuberechnung in die Welt von ETL und MapReduce zu transferieren. Diese unterscheiden sich von klassischen Datenbanksystemen im Hinblick auf die verwendeten Anfragesprachen und Programmiermodelle, die Erfassung von Datenänderungen, sowie die transaktionalen Garantieen und Konsistenzmodelle. Wie diese Arbeit zeigen wird, ist eine Übertragung von Sichtwartungskonzepten auf ETL und MapReduce dennoch möglich und kann zu einer erheblich gesteigerten Berechnungseffiziens führen.

Acknowledgements

I would like to express my sincere gratitude to my advisor Prof. Dr.-Ing. Stefan Deßloch for his continuous support and guidance. Furthermore, I would like to thank Prof. Dr.-Ing. Bernhard Mitschang for taking the role of the second examiner, Prof. Dr. Arnd Poetzsch-Heffter for acting as chair of the PhD committee, and Prof. Dr. Reinhard Gotzhein for being on the PhD committee.

In the beginning of my PhD studies, I had the opportunity to gain invaluable insights into data integration in practice as an intern at IBM and I would like to express my gratitude to Dr. Albert Maier and Oliver Suhre for hosting me.

I am grateful to Michael Koch, Roya Parvizi, Marc Schäfer, and in particular Johannes Schildgen who substantially contributed to my research through their Bachelor and Master theses. Furthermore, I want to thank my former fellow student Sebastian Baumgärtner for explaining the ins and outs of SAP's warehousing solutions to me.

My PhD studies would not have been nearly as much fun without my former "roommate" Dr. Nikolas Nehmer. I would like to thank him and my other colleagues Dr. Sebastian Bächle, Andreas Bühmann, Dr. Philipp Dopichaj, Dr. Jürgen Göres, Volker Höfner, Yong Hu, Joachim Klein, Dr. Yi Ou, Weiping Qu, Daniel Schall, Dr. Karsten Schmidt, Dr. Boris Stumm, and Dr. Andreas Weiner, who made this time enjoyable.

Thomas Jörg
München, March 2013

Contents

1 Introduction

Recurrent computations over changing input data can be performed through one of two approaches. We may either fully recompute from scratch or, alternatively, recompute incrementally. The idea behind incremental recomputations is to consider incremental changes in the input data and derive incremental changes to the output data. That way, reprocessing unchanged portions of input data may be avoided and the efficiency may be improved considerably. The concept of incremental recomputations has been studied in the database research community mainly in the context of incremental maintenance of materialized views. In this thesis, we argue that the same fundamental concept can be advantageously applied in other areas of data management, in particular materialized data integration using Extract-Transform-Load (ETL) tools and MapReduce computations. In either area recurrent computations are very common; however, recomputations are typically done from scratch. We will show that incremental approaches are feasible in these areas as well and may lead to considerable efficiency improvements.

We briefly describe the concepts of materialized views, materialized data integration, and MapReduce to point out similarities and draw an analogy between them.

- A database view is defined over one or more base tables by a given view definition and can thus be thought of as a derived table. Unlike standard database views, materialized views are pre-computed and stored physically in the database much like a regular table. Materialized views can thus be queried very efficiently.

- Data integration is an important concept in enterprise information management. It addresses the need to jointly query and analyze data managed by heterogeneous source systems. Data integration techniques fall into one of two categories: virtual and materialized. The focus of this thesis is on materialized data integration, which consolidates data from heterogeneous sources in a central repository, usually referred to as data warehouse. For this purpose, data of interest is extracted from the heterogeneous source systems, transformed to resolve data heterogeneity and improve data quality, and finally loaded into the data warehouse. Materialized data integration is usually supported by middleware systems

referred to as Extract-Transform-Load (ETL) tools. The ETL process is typically repeated periodically.

- MapReduce is a programming model and an associated framework for distributed data processing on large clusters of commodity hardware. In recent years, MapReduce became very popular in web-scale data management including web indexing or page rank computation. The MapReduce programming model is simple but powerful in the sense that MapReduce programs are easily parallelizable and thus highly scalable. MapReduce is layered on top of a distributed file system from which input data is read and to which output data is written.

From an abstract point of view, materialized views, materialized data integration, and MapReduce computations show interesting similarities. To compute a materialized view, data is read from the base tables, transformed according to a user-defined view definition, and written back to the database system. To populate a data warehouse, data is read form heterogeneous source systems, transformed according to a set of ETL jobs, and written to the warehouse. To evaluate a MapReduce computation, data is read from the underlying distributed file system, transformed according to user-defined map and reduce functions, and written back to the distributed file system.

Irrespective of apparent similarities, there is an important difference. When the underlying base tables of a materialized view are updated, the view is usually not recomputed from scratch but rather in an incremental fashion. Incremental view maintenance has been intensively studied in database research and is supported by all major relational database products today. However, neither state-of-the-art ETL tools nor the MapReduce framework support incremental recomputations. In fact, ETL and MapReduce jobs are typically fully re-evaluated on a regular basis to obtain up-to-date results. We argue that the efficiency of recurrent computations in both ETL and MapReduce can vastly be improved by incremental approaches.

This thesis describes our work in transferring incremental view maintenance techniques into the world of materialized data integration and MapReduce. Doing so was by no means straightforward but posed a number of interesting research questions. This was due to the fact that ETL and MapReduce differ in many fundamental aspects from relational database systems that are typically used to manage materialized views.

Perhaps the most important difference exists between the query and transformation languages and underlying data models of these environments. In ETL, an industry-standard language such as SQL does not exist but ETL tools use proprietary operator models. ETL operators do not compare well to relational algebra operators but rather to operating system-provided command line utilities such as grep, sed, sort, etc. That is, ETL operators are rather

task-specific and non-minimal in the sense that they overlap in terms of their transformation semantics. Some ETL operator are highly specialized; all major ETL tools offer dedicated data cleansing operators, for instance.

MapReduce does not use a query and transformation language. The framework rather provides abstract methods, which need to be overwritten by MapReduce programmers using a general-purpose programming language. Furthermore, MapReduce does not make use of the relational data model but uses a generic key-value model instead.

All previous work on incremental view maintenance is based on database query languages such as SQL or relational algebra and thus cannot be applied to either ETL nor MapReduce. Apart from the query and transformation languages ETL, MapReduce, and relational database systems differ in further aspects such as transaction handling, change data capture, data access paths, and fault tolerance. In the course of this thesis, we reconsider relational view maintenance techniques, adapt them as needed if possible or come up with new ideas.

2 Materialized data integration – state of the art

In this chapter we will briefly review the state of the art in materialized data integration, which is the principal problem area of this thesis. In the Database Encyclopedia, the (more general) term "data integration" is used interchangeably with the term "information integration" and is defined as follows [Hal09].

> Information integration systems offer uniform access to a set of autonomous and heterogeneous data sources. Sources can range from database systems and legacy systems to forms on the Web, web services and flat files. The data in the sources need not be completely structured as in relational databases. The number of sources in an information integration application can range from a handful to thousands.

In the remainder of this section, we will discuss the challenges of data integration in Section 2.1. This thesis is focused on a specific form of data integration, known as materialized data integration, which will be addressed in Section 2.2. Section 2.3 will introduce several alternative architectures for materialized data integration and point out strengths and weaknesses. We will conclude the chapter with a statement of goals to be achieved by this thesis in Section 2.4, discuss possible use cases for our approach in Section 2.5, and review related work in Section 2.6.

2.1 Integration challenges

In short, data integration aims at providing uniform access to non-uniform data sources. In this section we describe fundamental challenges of data integration. The proposed classification scheme and the terminology are based on [Gör09, LN06, SL90]. These authors distinguish three major challenges of data integration caused by the distribution, autonomy, and heterogeneity of data sources, which are discussed in Section 2.1.1, 2.1.2, and 2.1.3, respectively. We perceive data quality issues as fourth independent integration challenge and elaborate on this in Section 2.1.4.

2.1.1 Distribution

Intuitively, data is said to be distributed, if it is stored in different "locations". Such locations can either be *physically* distributed or *logically* distributed or both.

Physical distribution means that data resides on physically separated and possibly geographically distributed machines. We call data *technically* distributed, if it is physically distributed in a heterogeneous environment, i.e. managed by software systems of different kinds. Hence, technical distribution is a special case of physical distribution. Most often physical distribution is visible to the user, i.e. the user needs to be aware of the location of data to access it. However, physical distribution may also be transparent to the user. Distributed database systems and distributed file systems, for instance, use physical distribution to improve access times and provide fail-safety and, at the same time, offer uniform data access.

Logical distribution means that different storage locations, such as database tables, exist for semantically related data records. This situation often occurs when information systems with overlapping functionalities exist. For example, customer relationship management systems and billing systems, both store customer-related data. Note that logically distributed data is not necessarily physically distributed. The customer relationship management system and billing system may, for instance, store data in the same back-end database.

Both, physical and logical distribution may result in data redundancy, i.e. semantically similar data is being stored in different (physical or logical) locations. Redundancy may be controlled, i.e. the storage system may keep redundant data in sync. This feature is typically provided by non-technical distributed storage systems only. Uncontrolled redundancy is more common in data integration, because data to be integrated is typically managed by independent systems.

2.1.2 Autonomy

Source systems to be integrated are typically autonomous. That is, the systems are under separate and independent control. Those who control the systems are often willing to open up their systems for integration only if they retain control. Frequently, system owners are rather reluctant to changes with regard to both, the system itself and the system's workload to limit the impact of the integration system on operational processing.

In literature, different forms of autonomy are distinguished. Design autonomy refers to the freedom of a system to choose the data being managed (the universe of discourse) and the data representation with regard to data model, data formats, schema, naming, and constraints. Design autonomy is the pri-

mary cause of various forms of heterogeneity, which will be discussed in the following section.

Interface autonomy means that a system can freely decide which data access interfaces to provide. This includes the choice of supported query mechanisms (canned queries vs. arbitrary queries), query languages, protocols, and advanced features such as change data capture mechanisms. Other forms of autonomy, such as access autonomy and judicial autonomy [LN06] as well as association autonomy and execution autonomy [SL90] are not directly relevant in the context of this thesis and are therefore omitted here.

2.1.3 Heterogeneity

In [LN06], Leser and Naumann consider two information systems to be heterogeneous, if they do not offer the exact same methods, models, and structures for data access. Resolving heterogeneity is a key task in data integration. Usually data sources are heterogeneous among each other; furthermore heterogeneity exists between the integration system and individual sources. Only the latter form of heterogeneity is relevant for data integration, because sources do not communicate directly.

In this section, different forms of heterogeneity are classified. Heterogeneity is considered only with regard to the structure of data, i.e. on the data model and schema level. Instance level integration challenges are summarized under the term *data quality* and discussed in the subsequent section.

Technical heterogeneity Technical heterogeneity subsumes differences in the way data is accessed from the source systems. This includes differences in the communication protocols (e.g. JDBC or SOAP), data exchange formats (e.g. binary data or XML), query languages (e.g. SQL or XQuery), and query mechanisms (e.g. arbitrary queries or canned queries). Technical heterogeneity is closely related to technical distribution.

Syntactical heterogeneity Syntactical heterogeneity denotes differences in the representation of identical application concepts on the syntactic level. Common examples are different byte orders of binary data (little endian vs. big endian), different character encodings (e.g. UTF-8, ASCII, or EBCDIC), or different separator characters for tabular data (tab-delimited or comma separated values). We furthermore subsume differences in the data types, domains, or physical units for representing identical application concepts under the term syntactical heterogeneity. This specific form of heterogeneity is referred to as domain heterogeneity in [Gör09].

What is common to all forms of syntactical heterogeneity is that they can rather easily be resolved for data integration. It is straightforward to convert between character encodings or replace character separators, for instance.

Data Model heterogeneity Structured data adheres to a schema definition. While it is not mandatory, semi-structured data such as XML documents may adhere to a schema definition too. A schema is defined based on a data model that specifies a set of modeling concepts. Commonly used data models are the relational model, object-oriented models in various flavors, and the XML data model. Data model heterogeneity occurs when the data model of a source systems differs from the data model of the integration system. In this thesis, we focus on the integration of relational data and do not consider issues related to data model heterogeneity.

Structural heterogeneity Structural heterogeneity denotes differences in the representation of identical application concepts using the same data model. The design of a schema is influenced by various factors such as the range of real-world concept to be modeled, preferences of the schema designer, and the technical capabilities of the storage system.

Structural heterogeneity is very common in schemata based on non-minimal data models such as XML. In the XML data model the same real-world concept can often be expressed using different modeling concept. For example, data items may be modeled either as elements or attributes. Furthermore, relationships may be represented by nesting related elements or by using references.

Structural heterogeneity may also occur in schemata based on the (more rigid) relational data model. A common form of structural heterogeneity in the relational model is different degrees of normalization. Operational databases are typically organized in at least third normal-form to avoid update anomalies. In contrary, analytical databases such as data warehouses are usually highly denormalized to reduce the number of joins required for query evaluation and improve query response times.

In the literature the term schematic heterogeneity is used to mean a special case of structural heterogeneity. Schematic heterogeneity denotes the usage of data model concepts from different meta-layers to model identical application concepts. In the relational model, for example, application concepts may be represented by relation names, attribute names, or attribute values. To resolve schematic heterogeneity so-called higher-order query languages have been introduced [LSS01, WR05]. This specific problem area is not in the scope of this thesis though.

Semantic heterogeneity Data in an information system does not have an inherent meaning. In fact, it needs to be understood in its context to be useful. The context includes the name of schema elements, the relative position of schema elements, and knowledge about the application domain, just to name a few. Typically, the context is only partly available in machine-readable form. While the schema definition is usually available, knowledge about the application domain is most often not explicitly modeled.

Essentially, a schema element is nothing but a plain character string specifying its name. Such a name characterizes a concept of the application domain that is referred to as "intention" or "semantic" of the schema element. Semantic heterogeneity denotes differences in the relationships of names and intentions of schema elements. Common causes of semantic heterogeneity are different names having the same intention, referred to as synonyms, or equal names having different intentions, referred to as homonyms. Furthermore, different names having somewhat related intentions that are neither disjoint nor identical are frequently found.

Resolving semantic heterogeneity is probably the biggest challenge of data integration. It is widely agreed that a strong artificial intelligence (AI) would be required to resolve it automatically and the problem is therefore considered to be AI-complete. Current approaches to resolve semantic heterogeneity are semi-automatic and only meant to support human experts in this task.

2.1.4 Data quality

In this section, we classify data quality problems, i.e. problems related to erroneous and inconsistent data. Techniques to resolve data quality problems are referred to as data cleansing or data cleaning. Different classifications of data quality are found in literature and terminology is diverse. We will review classifications by Leser and Naumann, Rahm and Do, and Kimball and Caserta and discuss how they are related.

Leser and Naumann distinguish two kinds of data quality issues, based on how they can be resolved [LN06]. A data quality issue is referred to as *simple*, if it can be identified and resolved by examining a single tuple at a time. A so-called *complex* data quality issue, in contrast, can only be identified and resolved when multiple tuples are analyzed together. Leser and Naumann furthermore distinguish two successive phases of data cleansing. In the first phase, which is referred to as data scrubbing, simple data quality issues are resolved. Complex data quality issues are resolved in the second phase, which is referred to as duplicate elimination.

In [RD00], Rahm and Do classify data quality problems based on where they occur. The authors distinguish between single-source and multi-source problems and between schema-related and instance-related problems. The classi-

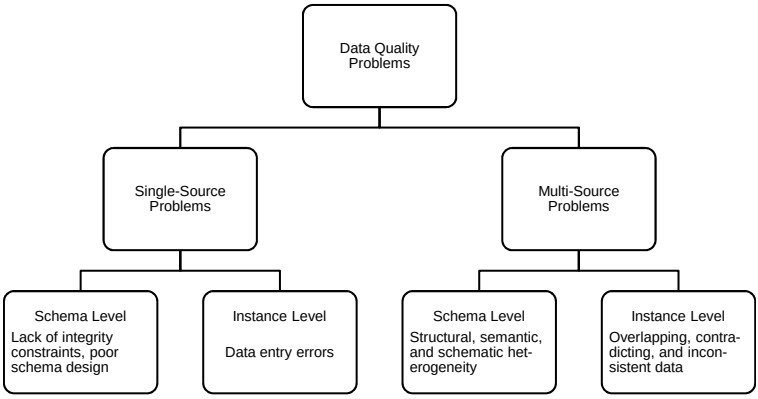


Figure 2.1: Classification of data quality problems after Rahm and Do [RD00]

fication scheme is shown in Figure 2.1. Single-source problems, as the name suggests, may already occur in a single data source. Multi-source problems occur not until data from multiple sources is integrated. Single-source problems do, however, occur in the multi-source case too.

In a single data source, schema-related problems result from the lack of model-specific or application-specific integrity constraints. For data sources without schema support, such as files for instance, few restrictions can be imposed on what data can be entered and stored. Database systems enforce restrictions of a fixed data model and a user-defined schema. Furthermore, they allow for application-specific integrity constraints to be specified. Table 2.1 shows some common schema-related data quality problems in a single data source. If supported, schemata and integrity constraints can help to prevent these kind of problems in the first place.

Instance-related data quality problems relate to errors and inconsistencies that cannot be prevented at the schema level. Common problems of this kind for the single-source case are shown in Table 2.2. Note that missing values, misspellings, cryptic, embedded, and misfielded values, violated attribute dependencies, and word transpositions are simple data quality issues in the terminology of Leser and Naumann, whereas duplicate and conflicting records are complex data quality issues. Unlike the name suggests, simple issues are not always simple to resolve. Correcting misspellings or translating cryptic values requires domain-specific dictionaries. Embedded values require sophisticated parsing to separate out individual parts. As we will show in Chapter 4, data scrubbing operations are often computationally expensive.

Problem	Reason	Example
Illegal values	A value is outside of its domain.	dateOfBirth = 30.13.70
Violated attribute dependency	An explicit dependency between attributes is violated.	age = (current date - birth date) should hold
Uniqueness violation	The same value appears multiple times in an attribute marked as being unique.	
Referential integrity violation	A foreign key refers to a non-existent primary key.	

Table 2.1: Single-source, schema-level data quality problems

Problem	Reason	Example
Missing values	Attribute values may be NULL. While NULL values can be prevented by integrity constraint, user often bypass them by entering dummy values.	phone = 9999-999999
Misspellings	Manual data entry and OCR often results in misspellings.	city = Kaiserlautern
Cryptic values and abbreviations	Abbreviations may obfuscate the meaning of values.	department = 'lgis'
Embedded values	Multiple values may be stored in one attribute. It needs to be parsed to extract individual values.	address = 'PO Box 3049, 67653 Kaiserslautern, Germany'
Misfielded values	Values may be assigned incorrectly.	city = 'Germany'
Violated attribute dependencies	An implicit dependency between attributes is violated.	zip = '67655', city = 'Bonn'
Word transpositions		$name_1$ = 'J. Smith', $name_2$ = 'Miller P.'
Duplicate records	Multiple tuples represent the same real-world entity.	$name_1$ = 'J. Smith', $name_2$ = 'John Smith'
Conflicting records	Duplicate records describe the same real-world entity by different values.	

Table 2.2: Single-source, instance-level data quality problems

Problem	Reason	Example
Contradicting values	Values describing the same real-world entity may contradict.	dateOfBirth = 01.01.1970, age = 30
Unequal representations	The same concepts may be represented differently.	gender = 'F', gender = 1
Unequal units	Different measuring units may be used.	price = '10,00' (in USD) price = '10,00' (in EUR)
Unequal precision	The precision of measurement not be the same.	length = 2h15min, length = 2h15min13sec
Unequal levels of aggregation	Similar data may be stored on different levels of aggregation.	Monthly sales vs. daily sales

Table 2.3: Multi-source, instance-level data quality problems

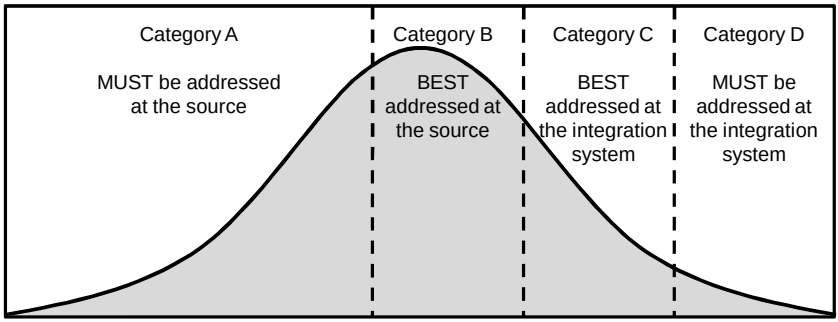


Figure 2.2: Classification of data quality problems after Kimball et al. [KC04]

Data quality problems, present in single sources, obviously occur in the multi-source case too. Moreover, the multi-source case introduces additional issues. Schema-level problems arise from syntactical, structural, or semantic heterogeneity of the sources to be integrated. Since Rahm and Do consider only structured data in the relational model, they do not discuss issues related to data model heterogeneity. Nevertheless data model heterogeneity also contributes to schema-level integration issues.

Similar to schema-level issues, single source instance-level issues also occur in the multi-source case. In fact, the likelihood of duplicate and potentially conflicting records (complex data quality issues) increases, because records referring to the same real-world entity are frequently found in multiple sources. In contrast to the single-source case, these records are usually structured differently and overlap only partly. During data integration, records referring to the same real-world entity need to be identified and consolidated.

Apart from complex data quality issues, the multi-source case introduces specific simple data quality issues too. Common issues of these kind are shown in Table 2.3. Even when similar attribute names and data types are used, the same concept may be represented by different values across sources. Moreover, similar values may be interpreted differently, e.g. because different measurement units are used). Furthermore, information may be provided with different precision or at different levels of aggregation across sources.

In [KC04], Kimball and Caserta classify data quality issues based on where they are best addressed. They distinguish four categories as depicted in Figure 2.2.

Category A issues are those issues that must be addressed at the data sources. Examples include missing information, dummy values, or “bogus” information. Most often, it is infeasible to recreate or correct such data if it has not been

captured correctly at the sources in the first place. Hence, category A issues cannot be resolved by the integration system. The integration system can only recognize these issues, remove any clearly bogus information, and report back the data quality problems to draw management focus on the source system defect. According to Kimball and Caserta, the majority of data-quality issues discovered during data warehouse projects fall into category A.

Category D issues are those issues that cannot be addressed at the data sources and thus, must be addressed at the integration system. Multi-source, instance-level data quality problems related to duplicates (i.e. complex data quality problems) clearly fall into this category. Obviously, duplicates across multiple sources can only be identified after the data has been gathered at the integration system.

The distinction between category B and C issues is less clear cut. Kimball and Caserta define category B issues to be those issues that are best addressed at the sources while this could, in principle, be done at the integration system too. Examples include misfielded values, violated attribute dependencies, and duplicate or conflicting records. Sometimes missing values also fall into this category rather than category A; a person's gender can usually be derived from the first name, for instance.

Analogous to the definition of category B, category C includes those data quality issues that are best addressed at the integration system, while this could also be done at the sources. Multi-source, schema-level issues clearly fall into this category. Likewise, multi-source, instance-level issues that are not caused by duplicates (i.e. unequal representations, units, precision, and levels of aggregation) belong to category C. Resolving these data quality problems at the sources would necessitate schema changes and undermine the sources' design autonomy.

Furthermore, certain single-source, instance-level issues, such as cryptic values, abbreviations, and embedded values fall into category C. These kind of data quality issues are often "not an issue" form a source perspective. In fact, it may be very appropriate to store address data as single attributes at the sources, for instance. However, such embedded values should be separated into individual parts like street, zip code, city, and so on, during data integration. In this way, the integration system provides better support for analysis such as address matching or householding.

2.2 Materialized data integration

The purpose of integration systems is to resolve heterogeneity and data quality problems (to a certain extent) and provide a single-system image to users. There are two fundamental data integration approaches known as virtual and

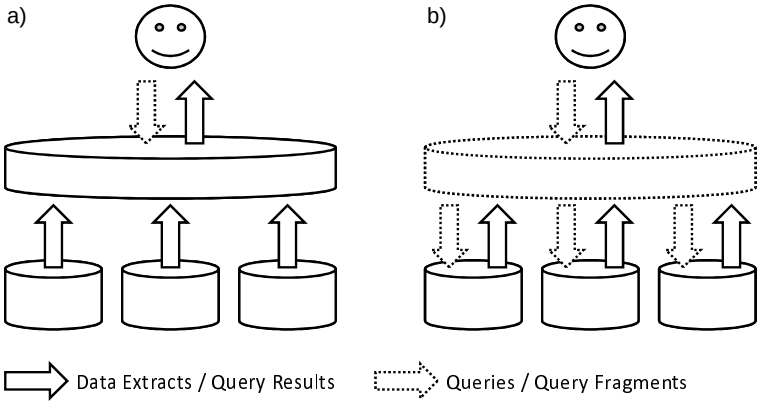


Figure 2.3: Materialized (a) and virtual (b) data integration

materialized integration, which are contrasted in Section 2.2.1. The focus of this thesis is materialized data integration. We identify desirable properties of materialized integration systems in Section 2.2.3. Subsequently, we review state-of-the-art architectures for materialized data integration, namely data replication systems, distributed materialized views, and Extract-Transform-Load (ETL) systems in Sections 2.3. We classify these integration architectures based on the properties defined in the Section 2.2.3 and point out strengths and weaknesses. Section 2.4 concludes the chapter with a statement of goals to be achieved by this thesis.

2.2.1 Materialized vs. virtual data integration

The key difference between materialized and virtual data integration is the place where data to be integrated is stored. In materialized integration, source data is copied to a central repository, whereas in virtual integration, all data remains in the source systems.

Figure 2.3 depicts materialized (a) and virtual (b) data integration on a high level of abstraction. Solid lines indicate the flow of data while dashed lines indicate the flow of queries. In materialized data integration, data is continuously moved from the sources to the integration system. Data movement is either performed periodically in an asynchronous manner or triggered by updates at the source systems. Query evaluation is done locally at the integration system and does not require any interaction with the sources.

Virtual data integration systems, in contrast, do not store data centrally but

provide an integrated view of source data [RS97]. To evaluate a user query, the integration system identifies query fragments that involve a particular source. Query fragments are send to the respective source systems and processed locally. The fragment query results are sent back to the integration system where the overall query result is assembled.

In summary, materialized data integration is continuously performed in the background, while virtual data integration is performed on the fly at the time of query evaluation. Materialized integration systems generally achieve better query response times, because queries can be evaluated locally on pre-integrated data. Furthermore, the impact on the operational source systems is typically less strong and more predictable, because it is independent from the query workload. Likewise, computationally expensive integration tasks such as data cleansing have no negative impact on query response times as these tasks are performed off-line. In fact, data cleansing may be prohibitively expensive if done at the time of query evaluation and thus infeasible for virtual data integration. Another advantage of materialized data integration is that historical data can be kept in the integration system even if it has been deleted at the sources. The biggest drawback of an asynchronous data integration pattern is that the integrated dataset is refreshed only periodically. Thus, the integration system will usually not contain the most recent data, i.e. user queries are evaluated on more or less stale data.

This thesis is focused on materialized data integration. In the following, we will simply speak of data integration to mean materialized data integration.

2.2.2 Data integration phases

Materialized data integration can be divided into three consecutive phases: the extract phase, the transform phase, and the load phase. The acronym ETL, which is used to mean one class of integration systems, is derived from these three phases. We will take a closer look at ETL and other classes of systems for materialized data integration in Section 2.3.

The phase model proposed in this section is based on [BG04, KC04]. In this section, we will frequently refer back to the integration challenges (distribution, autonomy, heterogeneity, and data quality issues) discussed in the previous sections. We will discuss which integration challenges are addressed by particular integration phases.

The overall data integration process is depicted in Figure 2.4. On the left-hand side, the figure shows the source systems storing data to be integrated. As companies evolve, they acquire or inherit various information systems to run their business, such as point-of-sale, inventory management, production control, and general ledger systems. Typically these systems are incompatible in the sense that they use different database systems, operating systems,

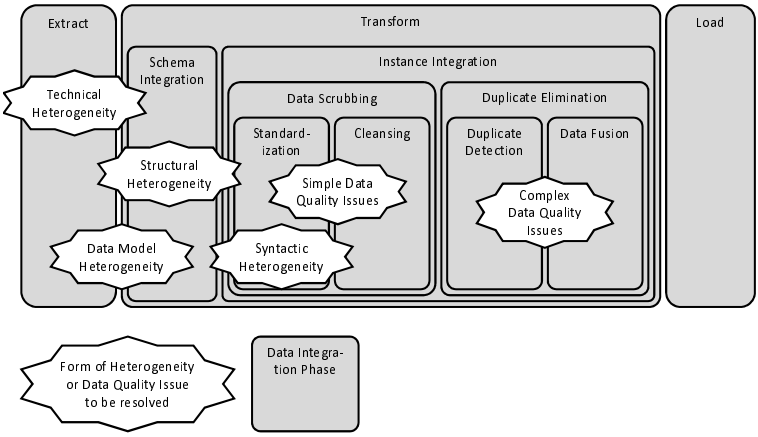


Figure 2.4: Phases of materialized data integration

hardware, or communication protocols. Hence, source systems are commonly heterogeneous (due to their design autonomy).

Extract The first step of data integration is the extraction of data of interest from the sources. To this end, the integration system must resolve technical heterogeneity. Integration systems differ in their ability to do so, as will be described in detail in Section 2.3.

Data extraction is performed by dedicated modules referred to as extractors. These extractors are tailored for a specific type of data source, such as database systems of different vendors, different file formats (e.g. flat files, COBOL copy books, or XML documents), or packaged applications (e.g. ERP or CRM systems). Extractors have two main responsibilities. First, extractors allow the integration system to connect to a source system and read out its data. Apart from technical heterogeneity, the extractor may also need to resolve data model heterogeneity at this point (e.g. by flattening XML data).

Second, extractors may optionally be able to monitor a source system and capture data changes. This feature is referred to as Change Data Capture (CDC). A classification of CDC approaches and implementation alternatives will be given in Chapter 5. CDC mechanisms allow the integration system to extract changed data from the sources, commonly referred to as deltas, and update the integrated dataset as needed.

In [BG04] four strategies to trigger data extraction are distinguished. Extraction may be performed periodically using a pre-defined interval. Extraction

may also be explicitly triggered by users. Furthermore, extraction may be performed in reaction to certain events, e.g. when a pre-defined number of changes has occurred. Finally, extraction may be done instantly when source data is changed, i.e. the deltas may be pushed to the integration system.

Transform The transformation phase can roughly be distinguished into schema integration and instance integration. These phases do not need to be strictly sequential but usually overlap to some extent.

In the schema integration phase, structural heterogeneity between the source schemata and the target schema is resolved. Unless done by the extractor, data model heterogeneity is also resolved at this point. Source systems are built for operational processing and typically use schemas in third normal-form. Integration systems, such as data warehouses, however, are build for data analytics and denormalized schemas are more appropriate for this purpose [KC04]. For this reason, structural heterogeneity between source and target schemas is very common in data integration scenarios.

Schema integration may also involve the transformation of instance data, when multiple values are stored in a single attribute in the source schema. Recall that such attributes have been classified as data quality problem and referred to as "embedded values" in [RD00]. Embedded values are typically decomposed into their individual parts and stored in separate attributes in the data warehouse.

Apart from structural heterogeneity, semantic heterogeneity commonly exists between source and target schemas. However, semantic heterogeneity at the schema level must be resolved during the design of an integration solution. That is, semantic correspondences between source and target schemas need to be identified by developers in a process referred to as schema matching. There are research efforts to automate this task. However, at the present time schema matching tools work (at best) semi-automatic. The result of schema matching is called schema mapping or logical data map [KC04]. A schema mapping captures the semantic correspondences between source and target schema elements together with descriptions of the required transformations at the instance level. It hence resolves semantic heterogeneity at the schema level and provides the basis for the development of an integration solution.

The second step of the transform phase is referred to as instance integration. The goal of instance integration is to homogenize data originating from different sources at the instance level and to identify and resolve data quality issues. When it comes to instance integration, the terminology used by different authors is diverse. We will review and compare classifications proposed by Bauer and Günzel [BG04], Leser and Naumann [LN06], and Kimball and Caserta [KC04].

The first step in instance integration is standardizing data formats and representations, i.e. resolving syntactic heterogeneity. This step is referred to as normalization [LN06] or standardization [KC04].[1] Data standardization may involve the following instance transformations [BG04]: Data type conversions, conversion between different data representations, standardization of string and date formats, unit and currency conversions, concatenation or separation of data values, computation of derived data, and data aggregation.

Subsequent to data standardization, there is a phase called data cleaning or data cleansing [BG04, KC04]. This phase addresses simple data quality problems, i.e. data quality problems that can be identified and resolved by examining a single tuple at a time. In Rahm and Do's classification (see Section 2.1.4), such data quality problems are missing values, misspellings, cryptic values and abbreviations, misfielded values, and violated attribute dependencies. Leser and Naumann refer to the resolution of simple data quality issues as data scrubbing [LN06].

The final step of instance integration is referred to as duplicate elimination. The aim of duplicate elimination is to identify tuples that relate to the same real-world entity and consolidate related tuples. Duplicate elimination requires standardized and cleansed input data, i.e. simple data quality issues must have been resolved.

In the terminology of Leser and Naumann, duplicate elimination addresses so-called complex data quality problems, i.e. problems that can only be identified and resolved when multiple tuples are analyzed together. Duplicate elimination can be subdivided into two consecutive steps. The identification of tuples related to the same real-world entity is referred to as duplicate detection [LN06], record matching [KC04], merge/purge problem [HS98], record linkage, or entity resolution [DF09]. The subsequent consolidation of tuples belonging to the same real-world object is referred to as data fusion [LN06] or surviving [KC04].

The measures taken to resolve data quality issues at the integration system are symptom-oriented. If the quality of source data is low, the integration system may not be able to improve the data quality to an acceptable level. Such data quality problems can only be addressed at their roots, i.e. directly at the source system [BG04].

Load The final phase of data integration is loading the transformed source data into the integration system for persistent storage. Depending on the frequency of data extraction, data may be loaded continuously or in batches.

[1] Bauer and Günzel use the term "data integration" ("Datenintegration" in German). This usage is misleading, because the term is usually used synonymous with information integration in literature, i.e. in a much wider sense. In this work, "data integration" is used in its common meaning.

Two types of loading can be distinguished, namely online and offline loading [BG04]. During online loading, the integration system can be queried while loading is in progress. If loading is performed offline, the integration system is unavailable. Offline loading may have performance benefits, especially when data is loaded in large batches.

Integration systems with an emphasis on data analysis, such as data warehouses, usually keep a history of data changes. That is, historical data is kept after it has been overwritten or deleted at the sources. Data warehouses usually employ a strategy known as "Slowly Changing Dimensions" for data historization, which will be discussed in detail in Section 2.5.1. Support for data historization needs to be built into the data loading module.

2.2.3 Properties of data integration systems

In this section, we introduce desirable properties of data integration systems. The set of properties will be used in the subsequent sections to classify existing integration systems and point out their strength and weaknesses.

In the beginning of this chapter, we discussed the challenges of data integration. Obviously, an integration system should provide the infrastructure to overcome these challenges, at least to some extent. In our opinion, integration systems can well be characterized by four key properties, namely the degree of tolerable source distribution, the degree of tolerable source autonomy, the data transformation capabilities provided, and the ability to incrementally maintain integrated data. These properties are described in greater detail in the following.

Degree of source distribution An integration system should at least be able to resolve physical data distribution, because data to be integrated is virtually always hosted by (geographically) distributed source systems. It is furthermore desirable to resolve technical distribution, i.e. integrate data that is hosted by systems of different kinds. Integration systems may allow for varying degrees of technical distribution. Some integration systems require their sources to be relational database systems, possibly from different vendors. Others are more flexible and allow for other kinds of systems, such as packaged applications or flat files for instance, to act as data sources too. Obviously, the stronger the assumptions about sources are, the more restricted the applicability of the integration system will be.

Degree of source autonomy It is a desirable property of an integration system to tolerate a large degree of source autonomy. Some integration systems tamper with the interface autonomy of source systems by imposing strong requirements. An integration system may require the sources to provide a specific

query interface or language such as SQL and tolerate some additional workload, it may require the sources to participate in dedicated communication protocols, or it may demand for specific features such as change data capture mechanisms. Furthermore, design autonomy may be tampered with by enforcing schema changes at the source systems or additional constraints to improve local data quality, for instance.

To meet the integration system's requirements, source systems may need to be adapted. However, such adaptations are often painful in practice. The reasons are twofold. First, system owners are usually eager to keep their autonomy and reluctant to change their systems. Changing a mature and proven system always entails the risk of introducing bugs and other problems. Second, system changes may be prohibitively complex. Legacy system, for instance, require significant re-engineering to be adapted to new requirements. Even worse, closed source packaged applications cannot be changed at all.

To sum up, an integration system should impose as few requirements as possible on the source systems and, at the same time, be able to exploit any interfaces and features currently provided by the system. That is, an ideal integration system should "make the most" of whatever the source systems are able and willing to offer.

Data transformation capabilities The transformation phase of materialized data integration has been discussed in Section 2.2.2. Recall that the transformation phase can roughly be divided into schema integration and instance integration. Schema integration is mainly about resolving structural heterogeneity, i.e. restructuring source data to match the target schema of the integration system. For this purpose, transformation capabilities comparable to the relational algebra (with aggregation) or SQL are sufficient. In fact, there are integration systems that use SQL to specify data transformations [Rau05]. Other systems use proprietary transformation languages. However, as recent studies suggest, these languages have a relational core [DHW+08, WCP09].

SQL-like transformation capabilities fall short of instance integration. Recall, that instance integration can be divided into data standardizing, cleansing, and duplication elimination. Data standardization can be performed using simple string manipulation and format conversion functions. However, special-purpose transformation engines have been built for data cleansing and duplication elimination that feature large domain-specific rule sets and ontologies. Only recently, integration system vendors started to integrate such engines into their products.

As said, the data transformation phase involves multiple tasks. However, integration systems may support only subsets of these tasks or may not fully support certain tasks. Details will be provided in Section 2.3. An ideal in-

tegration system should obviously provide the transformation capabilities to perform an "exhaustive" transformation.

Incremental maintenance In materialized data integration, data is extracted from the sources, integrated, and physically stored in the integration system. We will refer to the data content of the integration system as integrated dataset. Two phases of materialized data integration can be distinguished. A newly created integration system is initially empty; in the first phase, data is extracted exhaustively from the sources and loaded into the integration system. This phase is referred to as initial load.

When source data changes over time, the integrated dataset gets stale and needs to be maintained to regain consistency. Thus, maintaining the integrated dataset is the second phase of materialized data integration. While the first phase is performed only once, the second phase is an ongoing process. There are two fundamental approaches to maintain the integrated dataset such that it reflects the current state of the sources.

- The first approach is referred to as full recomputation. The idea is to exhaustively extract and integrate the source data in much the same way as it was done at the initial loading phase. We will refer to the resulting dataset as re-integrated dataset. To perform a full recomputation, the integrated dataset can be dropped and replaced with the re-integrated dataset. However, any historical data, which is no longer available at the sources, will be lost. To keep historical data in the integration system, which is a common requirement in data warehousing, the re-integrated dataset needs to be compared to the current integrated dataset to determine any data changes. These changes are then stored in the integration system, without deleting or overwriting any previous contents.

- Assuming that only a small fraction of source data is changed during loading cycles, a full recomputation is rather inefficient. To improve the efficiency, the repeated extraction and integration of unchanged source data should be avoided. To this end, changes are captured at the sources and used to directly determine the induced changes in the integrated dataset. This approach is referred to as incremental recomputation or incremental maintenance. A full recomputation may be done in much the same way as the initial computation, i.e. the transformation logic can be re-used. However, this cannot be done for recomputing incrementally. In fact, the integration logic needs to be adapted for change propagation, making it more complex in general.

In summary, support for incremental maintenance is a desirable property of an integration system. However, this property is the harder to achieve, the

bigger the system's data transformation capabilities are.

2.3 Integration systems

In this section, we will review different state-of-the-art integration systems. We propose to classify these systems into database replication systems, advanced database replication systems, materialized view systems, distributed materialized view systems, and Extract-Transform-Load (ETL) systems, which will be discussed in Sections 2.3.1, 2.3.2, 2.3.3, 2.3.4, and 2.3.5, respectively. We will characterize each class of systems in terms of the desirable properties discussed in the previous section.

2.3.1 Database replication

Database replication means that several copies of relations or relation fragments are stored at multiple physically distributed locations [RG03]. The aims of database replication are increased availability of data in case of node or communication link failures and faster query evaluation due to higher data locality.

Replication schemes can be classified along two dimensions, namely the propagation strategy and the ownership strategy [GHOS96]. The propagation strategy can either be *eager* or *lazy*, also referred to as synchronous or asynchronous replication, respectively. Eager update propagation means that any update is applied to each associated replica as part of the modifying transaction. Hence, the update propagation involves a distributed transaction across multiple replicas. The overhead and limited scalability of distributed transaction processing makes eager or synchronous replication undesirable or even unachievable in many situations. Gray et al. show that the time to wait for locks and the probability of deadlocks grow cubically with the number of nodes in an eager replication system [GHOS96].

Asynchronous or lazy replication abandons distributed transactions and applies updates to each replica in local transaction instead. Lazy replication achieves better scalability and is most often used in practice. However, the drawback is that different replicas of the same object may have different states at the same point in time.

There are two flavors of lazy replication depending on the ownership strategy used. In *master* or *primary site* replication, one replica of an object is designated the master replica. Only the master replica can be updated while all the others are read-only. Once the master replica has been updated the changes are propagated to other replicas. The alternative ownership strategy is referred to as *group* or *peer-to-peer* replication and allows all replicas to be updated concurrently. It is thus possible for two transactions to update the

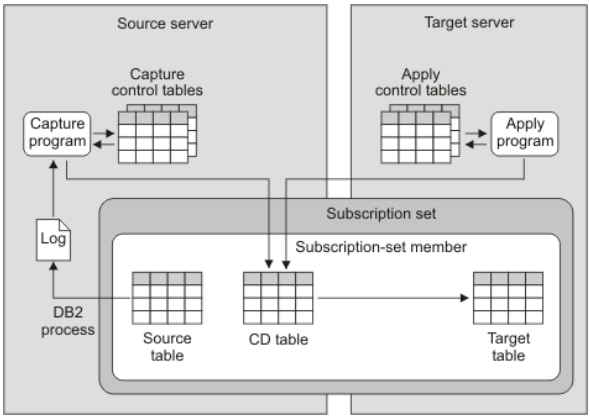


Figure 2.5: IBM DB2 SQL Replication overview [DB2a]

same object at different replicas and race each other to install their updates at the others. Such conflicts must be detected by the replication mechanism and reconciled so that updates are not lost.

Many commercial database replication offerings are available. As an example, we describe a replication solution provided by IBM DB2 called SQL replication [DB2a]. The overall architecture is depicted in Figure 2.5. DB2 SQL replication follows a lazy master replication scheme. It uses two programs referred to as Capture and Apply running on the source system and one or more target systems, respectively. The Capture program parses DB2 recovery logs and extracts committed changes to relevant source tables. These changes are written to a so-called change data (CD) table. It is possible to register just a subset of table columns for replication. In this case the Capture program ignores changes that affect unregistered columns only. The Apply program reads out the changes from the CD table and applies them to the replicated relations.

A subscription set associates source and target tables for replication and defines a mapping from source column names to target column names. A single source table can be associated with multiple target tables. For each source-target pair in a subscription set a predicate can be specified to select tuples for replication based on attribute values. Furthermore a scheduling strategy for data replication can be specified as part of a subscription set. DB2 SQL replication supports three alternative strategies. First, data replication may be performed at regular intervals. Second, it may be performed continuously,

i.e. as frequently as permitted by the current workload and the available system resource. Third, replication can be triggered by certain events signaled by an application or an user.

To sum up, we characterize database replication based on the desirable properties of data integration systems introduced in Section 2.2.3. Database replication techniques allow for physical distribution of source systems, however, technical distribution is often not supported. Replication solutions used to be vendor-specific and assume a homogeneous system environment. This has changed with the advent of a new class of systems we refer to as *advanced replication* systems, which will be discussed in the next section.

Database replication requires the source systems to be cooperative. Taking IBM SQL replication as an example, the source systems are required to switch to archive logging, run a capture program instance, and provide change data tables and further control tables. The source systems are thus forced to give up some of their autonomy.

The data transformation capabilities of traditional database replication are very limited. Typically replicated tables are one-to-one copies of their respective source tables. However, IBM SQL replication allows to restrict replicated tables to a subset of the source columns. Furthermore, replicated tables may be restricted to a subset of source tuples based on a given filter predicate. These subset operations correspond to relational projection and selection, respectively, which characterizes the transformation capabilities.

The last desirable property for data integration is the ability to incrementally maintain an integrated dataset. Obviously, database replication supports the incremental maintenance of replicated tables.

2.3.2 Advanced database replication

Traditionally, the focus of database replication was on increasing data availability in case of failure and query performance. The next generation of database replication solutions broadened its focus to include data integration. We refer to this class of systems as *advanced database replication* systems. These systems are build for heterogeneous environments and provide stronger data transformation capabilities.

Advanced database replication solutions have been developed by DataMirror and GoldenGate, just to name a few. DataMirror has been acquired by IBM in 2006. The product was renamed as IBM InfoSphere Change Data Capture (CDC) and integrated in the IBM InfoSphere Information Server offering [Inf]. GoldenGate has been acquired by Oracle in 2009. Both products are similar in functionality. We will discuss advanced replication features taking InfoSphere CDC as an example.

InfoSphere CDC supports a range of database systems of different vendors

including IBM, Microsoft, Oracle, Sybase, and Teradata. Like IBM SQL replication, the system harvests database recovery logs to capture changes of interest. InfoSphere CDC follows a lazy group replication scheme and hence, allows for bidirectional replication. To reconcile possible conflicts two automated strategies are provided. First, source updates may take precedence over target updates or vice versa. Second, larger values may take precedence over smaller ones or vice versa. The latter strategy is typically applied for timestamped updates. Additionally, conflicts may be reconciled by application logic.

InfoSphere CDC allows to transform data during replication in multiple ways. Similar to traditional database replication, a subset of source columns may be chosen for replication and particular source tuples may be selected based on a filter predicate. These operations are similar to relational projection and selection, respectively. Furthermore, InfoSphere CDC allows for value transformation. Specific source values may be replaced during replication based on user-defined value mappings. Abbreviations may be translated to an expanded form in this way, for instance. Additionally, InfoSphere CDC provides a set of so-called column functions to convert string, date, and time values. XML column values may be transformed using a subset of the XPath language.

Another interesting feature of InfoSphere CDC is so-called "one-to-many row consolidation". It allows to use a special kind of source table referred to as lookup table. Columns from both, regular source tables and lookup tables can be matched to columns of a single target table. During replication, source tuples are combined with lookup tuples based on the lookup tuples' primary key values. Thus, InfoSphere CDC essentially computes an inner equi-join between source and lookup tables. Updates at either of these tables are propagated to the target table. However, there are some restrictions. Deletions at the lookup table are not reflected at the target table. Furthermore, lookup tables and source tables must reside at the same system, i.e. joins across system boundaries are not supported. Such cross-system joins pose interesting synchronization problems, which will be discussed in Section 2.3.4 on distributed view maintenance.

In summary, advanced database replication enhances traditional database replication mainly in two aspects. First, a range of database systems from different vendors may be tied together. Thus, technical distribution is resolved to a larger degree. Second, additional data transformation capabilities are provided, including value transformations and local equi-joins.

2.3.3 Materialized views

A view is a special kind of relation that is derived from a set of base relations. Abstractly, a view can be regarded as a transformation function that maps base tuples to view tuples. The transformation function is typically re-

computed every time the view is referenced in a query. Alternatively, a view can be pre-computed and persisted; it is then referred to as *materialized* view. Materialized views provide efficient access, because they do not need to be computed at query evaluation time. Furthermore index structures may be built on materialized views.

However, the price to pay is the need to update materialized views upon updates to the base relations. This process is referred to as *view maintenance*. Most often it is wasteful to maintain a materialized view by recomputing it from scratch. Assuming that updates affect just a small part of the base data, it is often more efficient to compute only the changes in the view to update its materialization. This approach is referred to as *incremental view maintenance*.

Materialized views are not generally regarded as data integration technique, because the view and the base relations are managed by the same database system. We mention materialized views here for two reasons: First, the concept has been generalized to a distributed environment, where base tables and materialized views reside on different machines connected by a network. We refer to this generalization as *distributed materialized views* and provide details in the subsequent section. Second, our own work is based on a class of traditional incremental view maintenance algorithms that is introduced later in this section.

View maintenance

Maintenance of materialized views has been extensively studied in the database research community and numerous approaches have been proposed. A survey is provided in [GM95]. The authors suggest four dimensions along which the view maintenance problem can be studied.

- Information dimension: Refers to differences in the amount of information available for view maintenance. View maintenance techniques have been proposed to deal with situations where input data is not or only partially available. Work on *self-maintainable* views aimed at maintaining materialized views using just the deltas and the view itself, i.e. without accessing the base relations [GJM96]. We will discuss self-maintainability later in this section.

  Partial-reference maintenance considers only a subset of the base relations and the materialized view to be available. The *irrelevant update problem* means to decide whether a specific update leaves a view unchanged looking at the deltas and the view definition only, i.e. neither accessing the view nor the base relations.

  Interestingly, previous work has not considered deltas to be partial themselves. However, this situation often occurs in a warehousing environ-

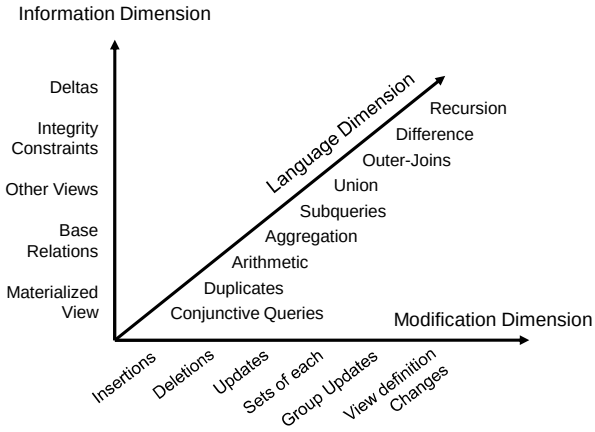


Figure 2.6: The problem space of view maintenance [GM95]

ment, where views reside on a remote database, at least from a conceptual point of view. In a warehousing environment, so-called Change Data Capture (CDC) techniques are used to capture deltas at the source systems. Many existing CDC techniques do not provide complete deltas but rather incomplete (or partial) deltas. Traditional view maintenance techniques, however, require complete deltas as input. One major contribution of this thesis is a generalized view maintenance technique that allows for maintaining a class of materialized views using partial deltas that will be proposed in Chapter 5.

- Modification dimension: Refers to the different types of modifications that a view maintenance algorithm can handle such as insertions, deletions, updates, or sets of each.

- Language dimension: Refers to the language used to express the view definition. Most algorithms assume (subsets of) relational algebra, SQL, or Datalog as view definition language.

- Instance dimension: Refers to whether a view maintenance algorithm works for all possible instances of the database and all possible instances of deltas or just for some instances thereof.

Figure 2.6 depicts the problem space defined by three of the four dimensions. Note, that there is no relative ordering between the points on each dimension.

$$
\begin{array}{rll}
\mathcal{E} ::= & R & \text{Base relation} \\
| & \sigma_p(\mathcal{E}) & \text{Selection} \\
| & \pi_A(\mathcal{E}) & \text{Projection} \\
| & \mathcal{E} \cup \mathcal{E} & \text{Union} \\
| & \mathcal{E} \cap \mathcal{E} & \text{Intersection} \\
| & \mathcal{E} - \mathcal{E} & \text{Difference} \\
| & \mathcal{E} \bowtie \mathcal{E} & \text{Join} \\
| & \mathcal{E} \times \mathcal{E} & \text{Cartesian product}
\end{array}
$$

Figure 2.7: Production rules for relational expressions

View maintenance algorithms have been proposed for many points in this problem space. Out of these, we will describe two approaches that are most relevant to our work in the remainder of this section.

First, we will give a review of *algebraic differencing*. In the problem space this technique is located as follows: It requires all base relations and non-partial deltas as input (information dimension), it applies to (sets of) insertions, deletions, and updates (modification dimension), view definitions are expressed in relational algebra (language dimension), and it applies to all database and delta instances (instance dimension). Second, we discuss a class of views referred to as *self-maintainable* views. These views can be maintained without accessing the base relations (information dimension) in response to insertions, deletions, and updates in the base relations (modification dimension). Self-maintainability has been studied for select-project-join (SPJ) views (language dimension) and applies to all database and delta instances (instance dimension).

Algebraic differencing

Much of our work is based on the algebraic differencing approach that was introduced in [KP81] and subsequently used for view maintenance in [QW91]. Some corrections to the minimality results of [QW91] and further improvements have been presented in [GLT97]. The approach has been extended to multiset algebra in [GL95]. Aggregation has first been considered in [Qua96] and further improved in [MQM97, LYC+00, GM06].

The basic idea is to differentiate the view definition to derive expressions that compute the changes in the view in response to changes to the underlying data.

	$\mathcal{E}$	$\Delta(\mathcal{E})$	$\nabla(\mathcal{E})$
1	R	ΔR	∇R
2	$\sigma_p(\mathcal{S})$	$\sigma_p(\Delta(\mathcal{S}))$	$\sigma_p(\nabla(\mathcal{S}))$
3	$\pi_A(\mathcal{S})$	$\pi_A(\Delta(\mathcal{S})) - \pi_A(\mathcal{S}_{old})$	$\pi_A(\nabla(\mathcal{S})) - \pi_A(\mathcal{S}_{new})$
4	$\mathcal{S} \cup \mathcal{T}$	$[\Delta(\mathcal{S}) - \mathcal{T}_{old}] \cup [\Delta(\mathcal{T}) - S_{old}]$	$[\nabla(\mathcal{S}) - T_{new}] \cup [\nabla(T) - S_{new}]$
5	$\mathcal{S} \cap \mathcal{T}$	$[\mathcal{S}_{new} \cap \Delta(\mathcal{T})] \cup [\Delta(\mathcal{S}) \cap \mathcal{T}_{new}]$	$[\mathcal{S}_{old} \cap \nabla(\mathcal{T})] \cup [\nabla(\mathcal{S}) \cap \mathcal{T}_{old}]$
6	$\mathcal{S} - \mathcal{T}$	$[\Delta(\mathcal{S}) - \mathcal{T}_{new}] \cup [\nabla(\mathcal{S}) \cap \mathcal{T}_{new}]$	$[\nabla(\mathcal{S}) - \mathcal{T}_{old}] \cup [\Delta(\mathcal{S}) \cap \mathcal{T}_{old}]$
7	$\mathcal{S} \bowtie \mathcal{T}$	$[\mathcal{S}_{new} \bowtie \Delta(\mathcal{T})] \cup [\Delta(\mathcal{S}) \bowtie \mathcal{T}_{new}]$	$[\mathcal{S}_{old} \bowtie \nabla(\mathcal{T})] \cup [\nabla(\mathcal{S}) \bowtie \mathcal{T}_{old}]$
8	$\mathcal{S} \times \mathcal{T}$	$[\mathcal{S}_{new} \times \Delta(\mathcal{T})] \cup [\Delta(\mathcal{S}) \times \mathcal{T}_{new}]$	$[\mathcal{S}_{old} \times \nabla(\mathcal{T})] \cup [\nabla(\mathcal{S}) \times \mathcal{T}_{old}]$

Table 2.4: Delta rules by Griffin et al.

Objects of interest are relations and expression in relational algebra generated by the grammar shown in Figure 2.7. As a convention, relational expressions are denoted by calligraphic letters to distinguish them from base relations. Relational expressions are used to define derived relations (or views). Changes to base relations are modeled as two sets – the set of deleted tuples and the set of inserted tuples. These sets are referred to as delta sets. For a relation R the set of inserted tuples is denoted by ΔR and the set of deleted tuples is denoted by ∇R. Updates are not modeled explicitly but represented by delete-insert-pairs, i.e. for each update in R there is a corresponding delta tuple in ∇R and in ΔR. The state of a relation R before the deltas have been applied is referred to as before-image and denoted by R_{old}; the state of a relation R after the deltas have been applied is referred to as after-image and denoted by R_{new}. The after-image can be derived from the before-image and the delta sets as $R_{new} := R_{old} \cup \Delta R - \nabla R$. Similarly, the before-image can be derived from the after-image and the delta sets as $R_{old} := R_{new} \cup \nabla R - \Delta R$.

Given a relational expression $\mathcal{E}$ that defines a view, two expressions for incremental view maintenance are derived; one to compute the insertions into the view, and another to compute the deletions from the view. The derivation is driven by two mutually recursive functions $\Delta(\mathcal{E})$ and $\nabla(\mathcal{E})$ presented in Table 2.4. These functions essentially specify rewrite rules and are therefore also referred to as delta rules. Each of these functions is recursively applied to $\mathcal{E}$. In each step the outermost relational algebra operator is considered and the expression is transformed according to the delta rules. Note that R is used in these rules to denote a base relation while $\mathcal{S}$ and $\mathcal{T}$ may either denote base relations or relational sub-expressions. Such sub-expressions will be transformed in subsequent recursion steps. The derivation process terminates when $\mathcal{E}$ has

been transformed down to the base relations. At that point no other rules are applicable.

Intuitively, the delta rules in Table 2.4 can be understood as follows.

- Base relation: The inserted and deleted tuples of base relations are directly available in the associated delta relations.
- Selection: An inserted tuple is propagated through a selection, if it satisfies the filter predicate. A deletion is propagated through a selection, if the tuple used to satisfy the filter predicate.
- Projection: An inserted tuple is propagated through a projection, if no alternative derivation existed before the change. A deletion is propagated through the projection, if no alternative derivation remains after the change.
- Union: A new tuple appears in the union of two relations, if it is inserted into at least one relation and has not been in the other before. A tuple disappears from the union, if it has been deleted from at least one relation and is not in the other any longer.
- Intersection: A new tuple appears in the intersection of two relations, if it is inserted into at least one relation and also present in the other one. A tuple disappears from an intersection, if it is deleted from at least one relation and used to be in the other before the change.
- Difference: A new tuple appears in the difference of relations S and T, if it has been inserted to S and is not in T after the change, or if it has been deleted from T and is in S after the change. A tuple disappears from the difference of S and T, if it has been deleted from S and has not been in T before the change, or if it has been inserted to T and has been in S before the change.
- Join: New tuples appear in the join of two relations, if a tuple inserted into one relation joins to tuples in the other one. Tuples disappear from the join, if a tuple deleted from one relation used to join to tuples in the other one before the change.
- Cartesian product: New tuples appear in the cross-product of two relations, if a tuple is inserted into one relation. Tuples disappear from the cross-product, if a tuple is deleted from one relation.

Example 2.1 *We provide an example to illustrate an algebraic differentiation. Let* $R(a, b)$ *and* $S(b, c, d)$ *be base relations and* $\mathcal{E} = R \bowtie \sigma_{d>10}(\pi_{b,d}S)$ *be a relational view definition. A relational expression* $\Delta\mathcal{E}$ *to compute the insertions*

into the view is derived below. The numbers next to the equivalence sign (≡) indicate which delta rule from Table 2.4 is used in a derivation step.

$$
\begin{aligned}
& \Delta(\mathcal{E}) \\
= \quad & \Delta(R \bowtie \sigma_{d>10}(\pi_{b,d}S)) \\
\equiv^{7} \quad & [R_{new} \bowtie \Delta(\sigma_{d>10}(\pi_{b,d}S))] \cup [\Delta(R) \bowtie \sigma_{d>10}(\pi_{b,d}S_{new})] \\
\equiv^{2,1} & [R_{new} \bowtie \sigma_{d>10}(\Delta(\pi_{b,d}S))] \cup [\Delta R \bowtie \sigma_{d>10}(\pi_{b,d}S_{new})] \\
\equiv^{3,1} & [R_{new} \bowtie \sigma_{d>10}(\pi_{b,d}\Delta S - \pi_{b,d}S_{old})] \cup [\Delta R \bowtie \sigma_{d>10}(\pi_{b,d}S_{new})]
\end{aligned}
$$

An expression $\nabla\mathcal{E}$ that computes the deletions from the view is derived in a similar fashion.

As suggested before, the algebraic differencing technique has been extended to include aggregation in [Qua96, MQM97, LYC+00, GM06]. Techniques for incremental maintenance of aggregate views will be discussed in Chapter 7.

View self-maintainability

For view maintenance, access to the underlying base data may be required. Note that the derived expression shown in Example 2.1 contains references to both, the deltas and the base relations. However, there is a class of materialized views that can be maintained using only the content of the view and the deltas, i.e. without accessing the base relations. These views are referred to as *self-maintainable* views.[2]

The concept of self-maintainable views has been introduced in [GJM96]. A materialized view is called self-maintainable with respect to a modification type (insertion, deletion, or update) if for all database states, the view can be self-maintained in response to a modification of that type to the base relations. Conditions under which several types of Select-Project-Join (SPJ) views are self-maintainable are presented in [GJM96]. In the following, we summarize the most important results.

An SPJ view that joins two or more distinct base relations is never self-maintainable w.r.t. insertions. This is intuitively clear, because inserted tuples may join to tuples that are neither in the view nor in the deltas but only found in base relations. Views that do not involve joins (SP views) are always self-maintainable w.r.t. insertions.

[2] Self-maintainability is related to the concept of *autonomously computable* updates introduced in [BCL89]. Autonomously computability is defined with respect to specific update instances. Self-maintainability, in contrast, is defined with respect to view definitions considering all possible updates. That is, the concepts differ w.r.t. the instance dimension.

A sufficient condition for an SPJ view with one or more base relations $R_1, \ldots, R_n$ to be self-maintainable w.r.t. deletions in R_1 is that for some key candidate of R_1, each key attribute is either retained in the view, or is equated to a constant in the view definition. Then, given a deleted tuple in R_1, the view tuples to be deleted can be identified by the key attributes. Note that the same considerations hold for SP views.

Updates are often implicitly modeled as pairs of deletions and insertions. However, views do not need to be self-maintainable w.r.t. both, deletions and insertions to be self-maintainable w.r.t. updates. Hence, updates have been considered separately in [GJM96]. Before we present the conditions for self-maintainability, we introduce some additional terminology: An attribute is called *distinguished* if it appears in the SELECT clause of a view definition. An attribute is called *exposed* if it is involved in some predicate of the view definition, i.e. appears in the WHERE clause.

An SPJ view that joins two or more distinct base relations $R1, \ldots, R_n$ is self-maintainable w.r.t. updates to R_1 if and only if either the updated attributes are unexposed and not distinguished or the updated attributes are unexposed and the view is self-maintainable w.r.t. deletions. In the first case, the updated attributes are irrelevant w.r.t. the view definition, because they neither appear in the SELECT clause nor in the WHERE clause. In the second case, the updated attributes do appear in the view; given a updated tuple in R_1, the view tuples to be updated can be identified by the key attributes, just like in the case of deletions.

Summary on materialized views

To sum up, we characterize the concept of materialized views based on the set of desirable integration systems properties introduced in Section 2.2.3. Materialized views do not allow for source distribution. In fact, it is assumed that base relations and materialized views are managed by the same database system. Consequently, the property of source autonomy does not apply either. The data transformation capabilities are related to the language dimension of the problem space introduced before. As said, a large variety of maintenance algorithms for different view definition languages have been proposed. Algebraic differencing, in particular, has been considered for relational algebra expression and extended to include aggregation and recursion. The last desirable property, i.e. the incremental maintenance of integrated datasets (the views), is clearly supported.

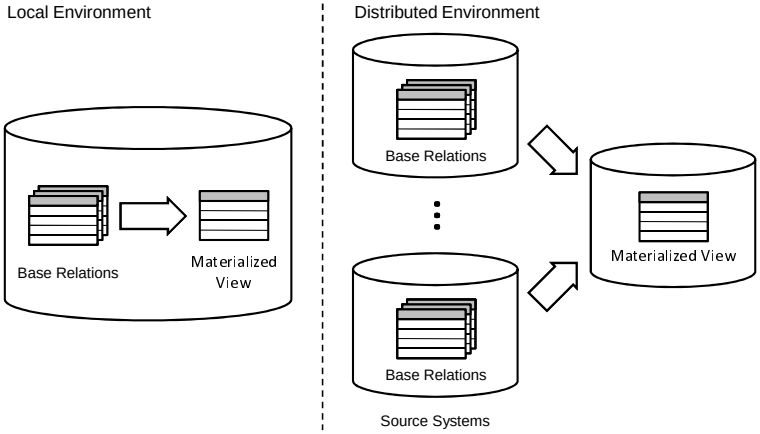


Figure 2.8: View maintenance in a local and distributed environment

2.3.4 Distributed materialized views

Traditional view maintenance techniques assume that both, the base relations and materialized views are under the control of a single database system as depicted on the left-hand side of Figure 2.8. The concept of materialized views has been generalized to a distributed environment[3] [AASY97, AAM+02, ZGMHW95, ZGMW98] as depicted on the right-hand side of Figure 2.8. In a distributed environment base relations and materialized views are managed by different database systems residing on different machines connected by a network.

Traditional view maintenance algorithms struggle in the distributed environment and may experience so-called *distributed incremental view maintenance anomalies* [ZGMHW95] or maintenance anomalies for short. The key difference in a distributed environment is that the sources do not understand view management and, hence, do not know about the view definitions. Consequently, sources can only be expected to report updates as they happen. The problem is that each arriving update may need to be integrated with data form the same or other remote sources before being stored in the materialized view. Re-

[3] In literature the distributed environment is sometimes referred to as *warehousing environment* in the context of view maintenance; furthermore a database managing materialized views over remote base relations is referred to as data warehouse. We prefer the term distributed environment, because the term data warehouse is more commonly used to mean a database system for decision-support having a multi-dimensional schema design and recording historical data.

call that join views are never self-maintainable w.r.t. insertions, i.e. base data access is generally inevitable for view maintenance. Thus, the view manager may have to issue queries back to some of the sources. Such queries are evaluated at the sources at a later point in time than the update that triggered view maintenance. Further updates may have happened in the meantime and thus, the source states may have changed. The queries sent back to the sources may hence deliver unexpected results causing the view to become inconsistent, i.e. maintenance anomalies may occur.

Several approaches to avoid maintenance anomalies have been proposed. They come with varying overheads and cost and differ in their assumptions w.r.t. the source systems and view definitions. The techniques can be classified into four basic approaches, namely distributed transactions, staging source data, timestamping updates, and compensating algorithms.

Distributed transactions In the local case, view maintenance is performed within the scope of the original update transactions. The isolation property of the transaction concept prevents "dirty" base data from being read during view maintenance. In a similar way, distributed transactions can be used to prevent maintenance anomalies in a distributed environment [ZGMW98]. However, the sources may be unsophisticated systems and unable to participate in global transactions. Even if they are not, distributed transactions require a global concurrency control mechanism. The involved overhead may be considerable. Source systems may need to be hold locks for the duration of view maintenance, for instance, which is most likely not acceptable.

Staging source data Another rather naive approach suggested in [ZGMW98] is storing copies of all source relations at the integration systems. In this way, queries do not need to be sent back to the sources but can be evaluated locally on a stable and consistent snapshot. Thus, the replicated base relations and the materialized view together are self-maintainable. However, this solution has several disadvantages. There is a considerable storage overhead, because the overall source data needs to be replicated. Furthermore the replicated data needs to be maintained in addition to the materialized view.

Hull and Zhou argue that for given view definitions the base data does not need to be replicated entirely but projections and local selections on base relations can be pushed down to the source systems [HZ96]. This is obvious considering that SP views are self-maintainable w.r.t. both insertions and deletions if the key attributes are retained in the view.

Quass et al. show that the base data to be replicated can further be reduced by exploiting knowledge about referential integrity constraint and non-updatable attributes [QGMW96]. They propose an algorithm to find a minimal

set of so-called auxiliary views so that the auxiliary views and the materialized view together are self-maintainable.

Timestamping updates Some algorithms for distributed view maintenance rely on timestamping updates at the sources [LHM+86, SP89, SF90]. However, the proposed algorithms are restricted to SP views and thus, maintenance anomalies are not considered. We will discuss the avoidance of maintenance anomalies by exploiting timestamped updates in Chapter 6 of this thesis.

Compensating algorithms Zhuge et al. were the first to recognized the possibility of maintenance anomalies [ZGMHW95]. To tackle this problem the authors proposed the Eager Compensating Algorithm (ECA) and later the Strobe family of algorithms [ZGMW96, ZGMW98]. The ECA algorithm applies to Select-Project-Join (SPJ) views with bag semantics over a single remote data source. The Strobe family of algorithms is designed for a multi-source environment but more restrictive in terms of the view definitions. The view definition is assumed to be an SPJ expression where the projection list contains the key attributes of each base relation and has thus set semantics. The basic idea behind both, the ECA algorithm and the Strobe family of algorithms is to keep track of source changes that interfere with view maintenance and perform compensation to avoid maintenance anomalies.

The major difference between the ECA algorithm and the Strobe family lies in the way compensation is performed. ECA relies on compensation queries that are sent back to the sources to offset the effect of changes that occurred concurrently to view maintenance. In contrast, Strobe performs compensation locally, exploiting the fact that the materialized view includes all key attributes of the source relations and is thus self-maintainable w.r.t. deletions.

An improved algorithm for view maintenance in a distributed environment referred to as SWEEP is proposed in [AASY97, AAM+02]. The SWEEP algorithm carries on the ideas of ECA and Strobe and improves them in several ways: Unlike ECA, SWEEP is not restricted to views over a single data source. Unlike Strobe, SWEEP works under bag semantics and does not require key attributes to be included in the view. SWEEP uses a technique referred to as on-line error correction. The idea behind online error correction is to process a single update at a time so that local knowledge suffices to offset any interference caused by concurrent updates.

The ECA algorithm, the Strobe family of algorithms, and the SWEEP algorithm have been designed for a specific class of data sources: It is assumed that the sources actively notify the view manager about local updates as soon as they occur. The communication is assumed to be reliable and order-preserving. Furthermore, the sources need to be able (and willing) to evaluate SPJ queries

issued by the view manager for maintenance and compensation purposes.

The Strobe family of algorithms have been implemented in the WHIPS research prototype (WareHousing Information Project at Stanford) [ZGMW98]. While many commercial database systems support traditional materialized views, we are not aware of any mature system for distributed view management. IBM DB2, for instance, does not allow for materialized views that reference so-called nicknames, i.e. local names of remote tables in a federated environment [DB2a].

We now characterize distributed materialized views based on the desirable properties defined in Section 2.2.3. As the name suggests, distributed materialized view techniques allow for source distribution. Physical distribution is fully supported, technical distribution at least to some extent. It is assumed that any source system is able and willing to cooperate in view maintenance. Depending on the approach, a source system must offer a relational query interface, participate in query compensation, communicate by messages-passing, and ensure the FIFO order of request processing and response messages. Obviously, these assumptions limit the degree of source autonomy. The data transformation capabilities of all approaches to distributed view maintenance we are aware of are limited to SPJ views. Support for the incremental maintenance, being the last desirable property we identified, is obviously provided.

2.3.5 Extract-Transform-Load

In the most general sense, the term Extract-Transform-Load (ETL) describes a style of materialized data integration. It is characterized by batch-oriented data processing and three consecutive integration phases, from which the name Extract-Transform-Load was derived. Data is first extracted from the sources, transformed, and finally loaded into a central repository as discussed in Section 2.2.2 before. The term ETL is also used to mean a class of tools for ETL-style data integration.

ETL has not received much attention in the database research community. There are few publications focusing on this specific area. However, ETL is of great practical relevance. The latest Gartner report "Magic Quadrant for Data Integration" released in November 2010, lists 50 vendors offering products with ETL-style integration capabilities [FBT10]. All major database vendors have one or more ETL solutions in their portfolio.

The primary application area of ETL is data warehousing [KC04]. In the early days, the ETL process was usually performed by combinations of shell scripts, custom programs, and database utilities. The development required considerable hand-coding and such solutions were hard to debug and maintain. This motivated the development of dedicated tools for developing and

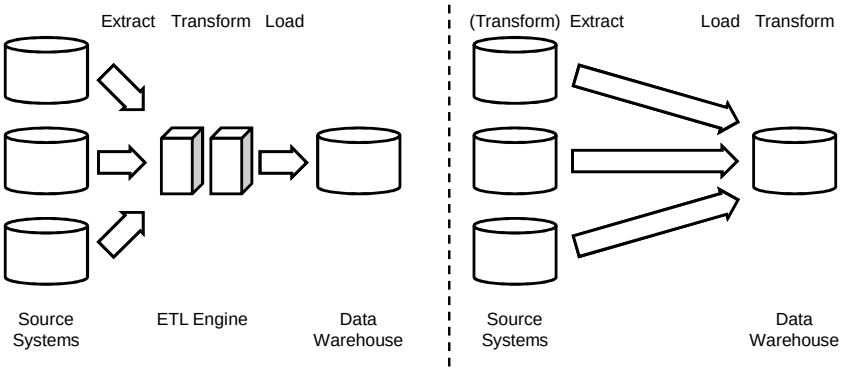


Figure 2.9: ETL (left-hand side) and ELT (right-hand side) architectures

operating ETL systems known as *ETL tools*. ETL tools typically provide an integrated development environment for ETL programs or *ETL jobs*, as they are more commonly called. They furthermore allow for scheduling ETL jobs and monitoring ETL job instances during execution. Additionally, some ETL tools offer analysis capabilities. Popular types of analysis are *data lineage analysis*, which allows to trace target data back to the source data it was derived from, and *change impact analysis* to understand the impact of changing ETL development artifacts on dependent artifacts.

ETL architectures

On an abstract level, ETL jobs are usually represented as data flow graphs, which are translated into an executable form for a specific platform later. ETL tools can thus be seen as code generators. Roughly, three types of platforms can be distinguished that are being used as execution engines by ETL tools.

- Dedicated ETL platforms: ETL tools such as IBM InfoSphere DataStage and Informatica PowerCenter rely on proprietary execution engines dedicated to ETL processing. Their engines offer native support for ETL transformation operations and are typically installed on dedicated server machines. The resulting architecture is depicted on the left-hand side in Figure 2.9.
- Database systems: More recently, an alternative architecture has gained popularity. ETL tools such as IBM DB2 Data Warehouse Edition or Oracle Warehouse Builder use standard database systems for ETL processing. For this purpose, the source data is extracted and immediately

loaded into the target database. Data transformations are performed at the target database. Because the loading phase occurs before the transformation phase, this approach is referred to as Extract-Load-Transform (ELT). Some ELT tools additionally allow to perform parts of the transformations at the source systems prior to the extraction phase. These tools are not referred to as TELT, though this acronym would be more accurate. An ELT architecture is depicted on the right-hand side in Figure 2.9.

ELT tools compile ETL jobs into SQL code and thus leverage the query engine of the data warehouse and possibly of the source databases to perform data transformations. ELT vendors claim that their approach has some advantages [Rau05]. They argue that using SQL instead of a proprietary language makes data integration solutions portable, flattens the learning curve for developers and administrators, and allows for the reuse of existing SQL stored procedures and functions. They furthermore argue that cost reductions are possible, because no dedicated ETL software and hardware infrastructure needs to be purchased and maintained.

Recently, the distinction between the ETL and ELT approaches has become somewhat blurred. Traditional ETL vendors such as Informatica, extended their tools to leverage the transformation capabilities of database systems. An add-on for Informatica PowerCenter called "Pushdown Optimization Option" is available, that allows data transformation processing to be pushed down into a relational engine, when both source and target data are co-resident in a relational database.

- General purpose programming languages: There are ETL tools that neither rely on proprietary ETL languages nor SQL but generate code in general purpose programming languages. The open source ETL tool Talend Open Studio, for instance, is able to translate ETL jobs into either Java applications or Perl scripts.

ETL programming model

It is common for ETL tools to use a graph representation for modeling and visualizing ETL jobs. As an example, screenshots of the user interfaces of two ETL tools, namely IBM InfoSphere DataStage and Informatica PowerCenter, are shown in Figures 2.10 and 2.11, respectively. These tools are considered to be the market leaders in a recent Gartner report [FBT10].

At the first glance, the user interfaces look very alike. We could have provided many more examples. In fact, it is very common for ETL tools to use directed, acyclic graphs as a programming abstraction. The nodes of an ETL graph represent ETL processing operators, which are also referred to as ETL stages.

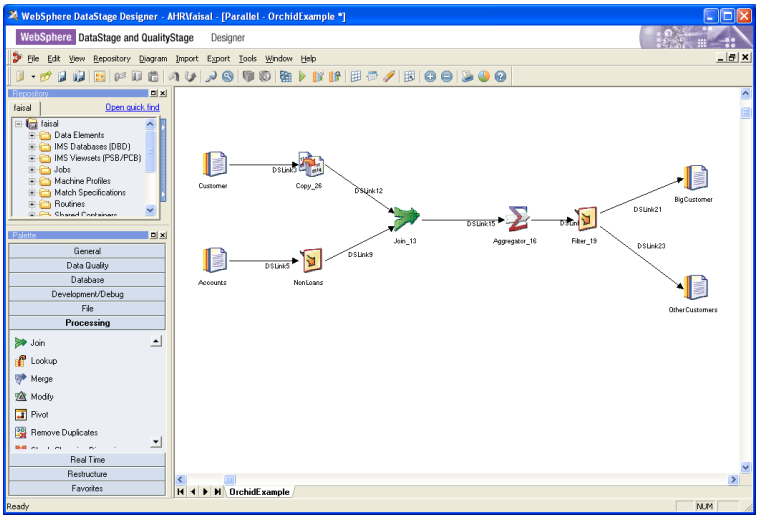


Figure 2.10: IBM InfoSphere DataStage user interface

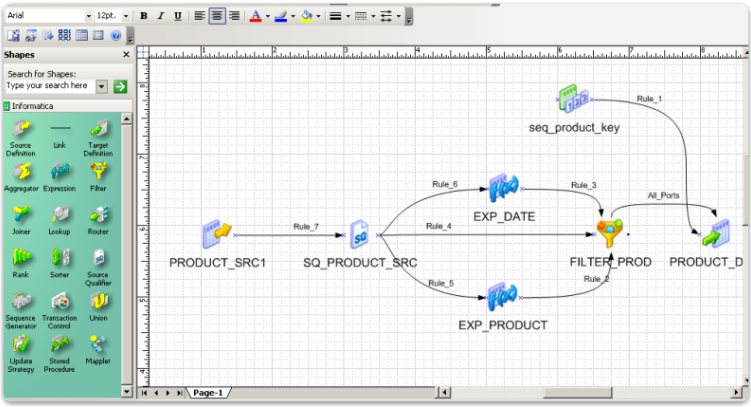


Figure 2.11: Informatica PowerCenter user interface

The edges of the graph indicate the flow of data. At the start nodes shown on the left-hand side[4] in Figures 2.10 and 2.11, data is extracted from the source systems. ETL tools typically provide extractors for a wide variety of source systems including general purpose and specialty databases, packaged applications, files in various formats, messages queues, and web services. The inner nodes represent data transformation operations. The target nodes shown on the right-hand side stand for data being loaded into target systems.

While ETL tools commonly use graphs to model ETL jobs, there is no common model for ETL operators, which act as nodes in these graph. A standardized language, such as SQL for relational database systems, does not exist in the ETL world. Instead, ETL tools typically provide a wide range of proprietary operators with complex and often overlapping transformation semantics. Even the ELT tools that rely on relational engines and generate standard SQL code differ in terms of their operator model. This situation complicates research in the ETL area. Focusing on the operator model of a specific tool will likely limit the applicability of any results. For this reason a platform-independent model for ETL is very desirable. In the sequel, we will discuss three independent efforts that provide a platform-independent view on ETL transformation semantics.

The first of these efforts is currently carried out by the Transaction Processing Council (TPC). The TPC has formed a subcommittee in 2008 to work on an industry standard benchmark for ETL called *TPC-ETL*. Such a benchmark needs to capture core transformation capabilities common to ETL tools of different vendors, much like the TPC-C and TPC-E benchmarks capture core OLTP processing capabilities and the TPC-H benchmark captures core OLAP processing capabilities.

Another interesting effort is the *Orchid project* carried out at IBM Almaden research lab in 2006. The Orchid system allows for translating schema mappings into ETL jobs automatically and translating ETL jobs back into schema mappings. In this process an intermediary model is used that captures the common transformation semantics of schema mappings and ETL jobs. This model is referred to as Operator Hub Model (OHM).

An extensive study on modeling ETL jobs is by Simitsis and Vassiliadis et al. [Sim03, Sim05, SVS05, TVS07, VSS02]. The authors propose a two layered approach using a conceptual model and a logical model for the representation of ETL jobs. We will discuss each of the mentioned efforts in greater detail in the sequel.

[4] By convention, the data flow direction in an ETL graph is left-to-right. This is in contrast to database query plans where the data flow is directed bottom-to-top most often.

TPC-ETL benchmark In 2008, the TPC formed an ETL benchmark development subcommittee under participation of all major ETL vendors that is currently working on a standard ETL benchmark called TPC-ETL.

The TPC subcommittee members feel that today's ETL market is in much the same situation as the relational database market was in the 1980s. At this time, vendors competed aggressively in an area that lacked a standard benchmark. The lack of common ground rules for evaluating the performance of products allowed vendors to make almost any claim they wished. This situation eventually led to the development of a series of TPC benchmarks for relational database systems.

Today, it is ETL vendors that compete aggressively in an area that lacks a standard benchmark. Virtually all vendors claim "world record" performance for their products [WCP09]. This motivated the development of TPC-ETL. This endeavor is very interesting for our work. For the first time major ETL vendors will agree on a set of core ETL transformation operations. It is the intention of the benchmark committee to include a diverse set of typical ETL operations in the benchmark. At the same time, however, the committee intents not to exclude any ETL tools by requiring specialized functionality that would not be common to most tools. TPC-ETL has not been released yet but preliminary results have been reported in [WCP09]. The benchmark committee expects the following types of transformations to be part of the benchmark.

- String, date, time, and numeric transformations: It is intended to create source datasets that are syntactically heterogeneous. The benchmark ETL job will require string, date, time, and numeric transformations as well as data type conversions to resolve syntactical heterogeneity.

- Lookups and joins: The benchmark will require data to be combined in a join fashion. Both, joins between two source datasets and between source and target datasets are intended to become part of the benchmark. The latter is relevant in dimensional modeling to lookup so-called *surrogate keys* from warehouse tables. We will discuss dimensional modeling later in this section.

- Conditional processing: Some aspect of the source-to-target transformation in the benchmark will vary depending on the data. Specific transformations will only be applied to data of a certain kind (such as customers from a specific country) or when certain errors in the data are to be handled.

- Aggregations: In the early days, data warehouses typically stored data in an aggregated form to keep its size manageable. Today, data is usually stored at the finest level of granularity to allow for more detailed

analysis [KR02]. However, ETL jobs may aggregate data to pre-compute results for common warehouse queries, which is stored redundantly at the warehouse.

- Data cleansing: The benchmark committee recognizes that data cleansing includes a broad range of complex transformations. Data cleansing is the area where today's ETL tools differ most widely. The benchmark committee feels that including aspects of data quality would complicate the benchmark. It is therefore intended to include only straightforward data cleansing tasks that essentially resolve simple syntactic heterogeneity in the source data.

The TPC-ETL draft provides an vendor-independent (and therefore platform-independent) view on what is believed to be core ETL transformation semantics. Before we discuss a platform-independent model for ETL, we review two more efforts touching on this aspect.

Orchid The Orchid system [DHW+08] is meant to bridge a gap in the tool chain used to develop data integration solutions. Business analysts usually start off with defining correspondences between source and target schemas at a high level of abstraction. For this task, so-called schema mapping tools are commonly used. Schema mapping tools allow to enter source-to-target schema correspondences in a declarative way, usually by drawing lines across the two schemas. Such lines may be further annotated with functions or predicate conditions to refine the transformation semantics of the mapping. Schema mappings provide a high-level transformation specification for the implementation of executable ETL jobs, which is done by ETL developers with the help of ETL tools.

Even though business analysts and ETL developers work as teams, their tools, i.e. mapping tools and ETL tools, did not directly interoperate. This gap in the tool chain motivated the development of the Orchid system. Orchid essentially automates three tasks that used to be done manually. First, based on schema mappings, ETL jobs are developed that implement the intended transformation semantics. Second, schema mappings are derived from ETL jobs to understand the transformation semantics at a declarative level. Third, when either ETL jobs or schema mappings are refined in the development process, the changes need to be reflected in the respective other representation.

The main challenge in translating schema mappings into ETL jobs is the difference in the expressive power of the underlying data transformation languages. Schema mappings, by design, do not capture the exact method of data transformations but simply indicate correspondences. In general, the transformation semantics of ETL jobs is too rich to be fully expressed at the schema

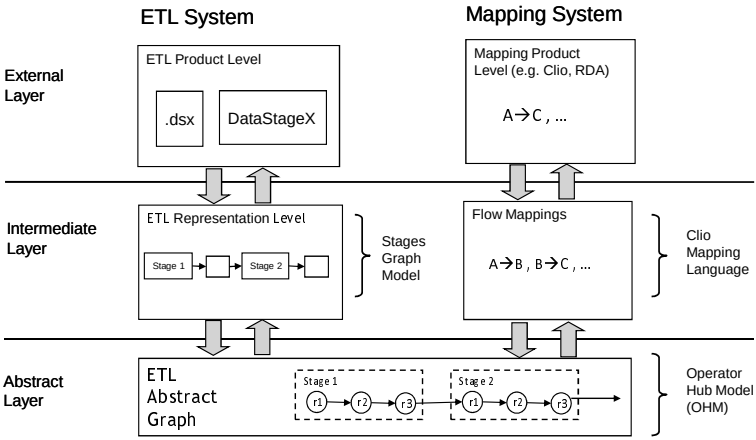


Figure 2.12: Orchid components and representation layers [DHW+08]

mapping level. Orchid tackles the problem of non-expressible transformation semantics at the mapping level by allowing business analysts to annotate mappings in natural language. Orchid generates ETL job skeletons that may contain some unresolved place-holders. ETL developers may then fully implement such place-holders with the help of the analysts' annotations.

The developers of Orchid saw two principal ways to facilitate interoperability between schema mapping and ETL tools. First, for a given pair of tools, a subset of vendor-specific ETL operators can be chosen that covers the expressible transformation semantics of the respective mapping language. The authors argue that this approach requires implementing an ad-hoc conversion between every possible pair of mapping and ETL tool and is thus undesirable. Instead, the authors advocate a more general solution. The basic idea is to develop a common model that captures the transformation semantics of both, schema mappings and ETL jobs in a platform-independent manner. This model serves as a "hub" for the conversion between these representations and is hence referred to as Operator Hub Model (OHM).

Orchid distinguishes models at three layers of representation as depicted in Figure 2.12. OHM is used at the so-called abstract layer. As the name suggest, models at this layer abstract from product-specific details, whereas at the external layer, mappings and ETL jobs are represented in their native, product-specific formats. Models at the intermediate layer help with the translation from mappings to ETL jobs or vice versa. ETL jobs are represented as graphs at this layer. Intermediate ETL models are, however, still product-specific in

the sense that product-specific ETL operator types are used as nodes in the graph. Mappings are represented in the Clio mapping language [HHH+05] at the intermediate layer.

The translation process performed by Orchid is indicated by Figure 2.12. Starting with an ETL job, its external representation is first translated into an intermediate model which is subsequently translated into an OHM model instance. Orchid proceeds by translating OHM instance into an intermediate mapping representation that is finally translated into an external, product-specific format. The translation steps can be performed in reverse order, to convert a schema mapping into an ETL job.

We now take a closer look at OHM, which proved very useful in our work, because it facilitates reasoning on and rewriting of ETL jobs in a product-independent manner. It is very common for the set of operators provided by a specific ETL tool to be non-minimal in the sense that there is an overlap in the transformation capabilities of the individual operators. Hence, the same transformation semantics can often be achieved by combinations of different operators. IBM InfoSphere DataStage, for instance, provides five different operators (or stages) with the ability to evaluate predicate conditions, besides doing other transformations. One reason for operator sets to be non-minimal is probably usability; by allowing individual operators to perform several kinds of transformations, the size of ETL jobs remains manageable. Another possible reason is that operators sets have evolved over time.

OHM was derived by decomposing ETL operators into their atomic constituents and can be characterized as an extension and generalization of relational algebra. In [DHW+08] a set of basic OHM operators is proposed, some of them having refined variants. The set of operators currently defined in OHM are PROJECT, FILTER, JOIN, UNION, GROUP, SPLIT, NEST, and UNNEST. PROJECT, FILTER, JOIN, and UNION roughly correspond to the traditional relational algebra operators project, select, join, and union, respectively. OHM uses a generalized notion of projection that allows for the creation of additional output columns and value transformations based on expressions similar to those used in the an SQL select clause.

OHM allows for refined operator variants through a notion of subtyping. Four subtypes of the PROJECT operator are part of OHM. BASIC PROJECT permits only renaming and dropping columns but does not support any value transformations, much like a traditional relational project operator. The KEYGEN operator generates surrogate keys in the output dataset. COLUMN SPLIT and COLUMN MERGE split the value of a single column into multiple output columns or vice versa.

The GROUP operator performs such transformations that merge a group of input tuples into a single output tuple such as aggregation or duplicate elimination. Note that the NEST operator is defined as a subtype of GROUP.

Since ETL jobs are graph-shaped in general, OHM includes the SPLIT operator that allows for branching the data flow and feeding the output of a single operator into multiple down-stream operators. Furthermore, Orchid considers NEST and UNNEST operators to deal with nested-relational data structures. However, the initial design of Orchid is restricted to flat schemas and does not elaborate much on this aspect.

In its current form, OHM has some limitations in expressing advanced data cleansing operations, such as address standardization. One reasons for this limitation is that such operations cannot be expressed at the mapping level. Another reason is that OHM is meant to be platform-independent. While relational algebra-like operations will produce the same result on any transformation platform, data cleansing is inherently fuzzy and the results may vary greatly depending on the platform of choice. The same argument holds for fuzzy duplicate elimination.

Simitsis and Vassiliadis et al. An extensive study on modeling ETL jobs is by Simitsis and Vassiliadis et al. The authors propose to model ETL jobs at two layers – the conceptional layer and the logical layer. The conceptional model specifies inter-attribute relationships between source and target schemas [VSS02] and can thus be seen as a special kind of schema mapping. The logical model describes the flow of data from the sources towards the data warehouse through the composition of transformation activities and data stores [Sim03]. The logical model thus serves a similar purpose as OHM in the Orchid system. The translation of conceptual model instances into logical model instances is discussed in [Sim05]. Rewriting logical model instances for the purpose of optimization is discussed in [SVS05]. Optimization at the physical level is discussed in [TVS07]; the idea is to introduce sorters into the ETL data flow to achieve "interesing orders" that can be exploited by down-stream operators.

Simitsis et al. do not propose a set of logical operators with fixed transformation semantics. Their approach is rather generic. Operators are characterized by a unique name, input and output schema, the set of attributes that the transformation relies on (functionality schema), the set of attributes resulting from the transformation (generated schema), the set of attributes that are not further propagated (projected-out schema), and a description of the transformation semantics. In [Sim05] it is proposed to describe the transformation semantics of logical operators in a functional programming language called LDL++. However, in [SVS05] the authors propose to use the relational algebra extended with functions as an alternative.

We reviewed the TPC-ETL benchmark draft, the Orchid system, and work on ETL modeling by Simitsis and Vassiliadis et al., because these efforts pro-

vide a platform- and vendor-independent view on core ETL transformation semantics. This survey provides the basis for an appropriate ETL transformation model to be used throughout this thesis, which will be introduced in Chapter 3.

To conclude, each of the three efforts suggests that the transformation capabilities of ETL tools have a "relational core". The TPC-ETL draft mentions joins, conditional processing, and aggregation, similarly OHM includes the operators JOIN, FILTER and GROUP, which correspondent to the relational join, selection, and aggregation, respectively. Furthermore, the TPC-ETL benchmark is expected to require string, date, time, and numeric transformation and data type conversions. Such value transformations are captured in OHM through a notion of a generalized projection operator. Similarly, Simitsis et al. propose to extend the relational algebra with functions to express value transformations.

Data cleansing is not fully covered in any of the three efforts. The TPC-ETL committee refrains from complicating the benchmark by including data quality-related aspects. It is intended to restrict the benchmark requirements to data standardization, i.e. heterogeneity is resolved at the syntactical level only. This can already be achieved using value transformation functions. The Orchid system has similar limitations. OHM does not include specific operators to capture data cleansing. While the generalized projection is appropriate to express data standardization operations, more complex data cleansing tasks such as address standardization or fuzzy duplicate elimination cannot be expressed in OHM. Simitsis et al. do not elaborate much on the aspect of data cleansing either.

Summary

To sum up, we characterize ETL tools based on the set of desirable integration system properties introduced in Section 2.2.3. The primary application area of ETL tools is data warehousing and one main strength is their ability to extract data from a large range of different source systems. That is, technical distribution can be resolved and source autonomy can be preserved to a very large extent. Another strength of ETL tools is their ability to perform complex transformations such as data standardization and cleansing, fuzzy duplicate elimination and joins across system boundaries.

However, ETL tools generally lack the ability to maintain integrated data in an incremental fashion. Unlike materialized view or database replication systems, ETL tools do not update integrated data in response to base data changes automatically. It is common practice to simply rerun ETL jobs from time to time to incorporate the latest changes in the base data into the integrated data that became stale in the meantime.

	Degree of source distribution	Degree of source autonomy	Data Transformation capabilities	Incremental Maintenance
Database Replication	Physical but no technical distribution	Very limited	Project, select	supported
Advanced Database Replication	Limited technical distribution (relational DBMSs only)	Limited (e.g. archive logging is required)	Project, select, local equi-join, data standardization	supported
Materialized views	Neither physical nor technical distribution	Not applicable	Project, select, join, aggregation, recursion, ...	supported
Distributed Materialized views	Limited technical distribution (relational DBMSs only)	Limited (sources must provide SQL interface, ensure FIFO message order, and participate in query compensation)	Project, select, join	supported
ETL	Technical distribution (Large range of extractors provided)	Autonomy is preserved	Project, select, (cross-system) join, aggregation, data standardization and cleansing, fuzzy duplicate elimination	unsupported

Table 2.5: Characterization of Data Integration Systems

2.4 Goals of this work

In the previous section, we presented a survey of different types of data integration systems and found them to differ in the degree of tolerable source distribution and autonomy, their data transformation capabilities, and their ability to maintain integrated data incrementally. The characteristics of each type of integration system are summarized in Table 2.5.

ETL tools have always had a strong emphasis on the integration of data from heterogeneous sources. Materialized views and database replication systems have different roots but have recently been extended with data integration capabilities. Nevertheless, ETL tools are still the first choice for large integration solutions that need to deal with a great degree of technical distribution, source autonomy, and data quality-related problems. The main strength of materialized view and database replication systems is their ability to maintain integrated data incrementally. Today's ETL tools lack this ability. In fact, "many companies [...] have data marts and cubes they rebuild every time they

use them" [She08].

We argue that incremental recomputation techniques can advantageously be applied in the ETL environment to improve the efficiency of recurrently executed ETL jobs. Doing so allows to shrink the data warehouse update window, to improve the timeliness of warehouse data by maintaining it more frequently, or to maintain the warehouse with less computing resources.

In a nutshell, this thesis shows ways to combine the efficiency advantage of incremental view maintenance with the advanced integration capabilities of ETL tools. We will propose an approach which leverages existing ETL tools without modifications to incrementally recompute previously integrated datasets in response to base data changes in Chapters 4 to 6.

Hereafter, we will shift our focus from traditional data integration scenarios supported by ETL middleware to web-scale data management systems. This class of systems emerged only recently and has been built to scale up to very large data volumes available on the web. A key concept in web-scale data management is the MapReduce paradigm for parallel data processing on large clusters of shared-nothing commodity machines introduced by Google [DG04]. MapReduce was primarily built for processing large amounts of weakly structured data, such as web request logs or crawled web documents. From an abstract point of view, a MapReduce program can be seen as a definition of a materialized view. MapReduce programs essentially specify transformations to be applied to base data and their result is stored in a materialized form. However, incremental view maintenance techniques have not been used in this environment so far. The benefits and challenges of this approach will be discussed in Chapter 7.

2.5 Use cases

In this section, we will discuss typical use cases for ETL systems. We will point out current problems and suggest how they could be solved by incremental recomputation techniques. ETL systems are primarily used for data warehouse loading and data migration. We will discuss these use cases in Section 2.5.1 and 2.5.2, respectively. Use cases for incremental recomputations in web-scale data management systems will be discussed in Section 7.4.

2.5.1 Use case: Data warehousing

Before we discuss how incremental recomputation techniques can be applied in data warehouse maintenance, we present some data warehousing preliminaries, namely the dimensional modeling methodology.

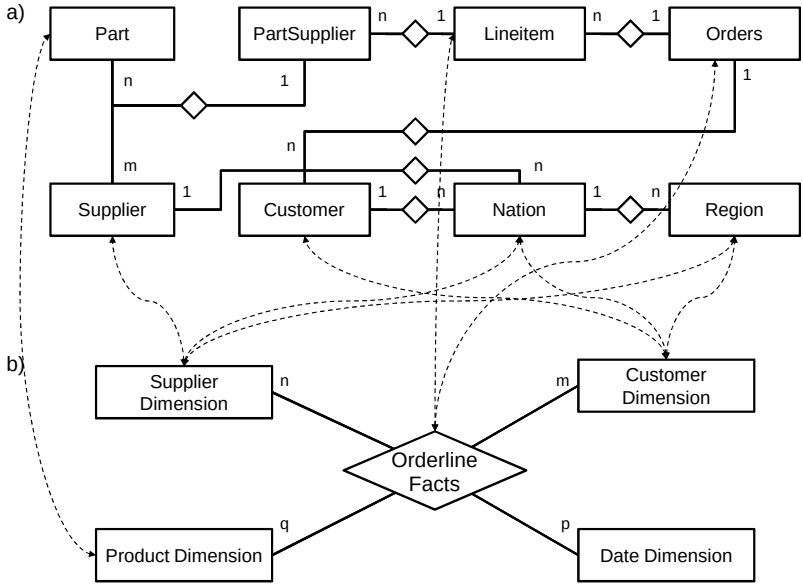


Figure 2.13: TPC-H benchmark schema (a) and SSB benchmark schema (b)

Dimensional modeling

Dimensional modeling is an established methodology for data warehouse design and is widely used in practice [KC04, KR02]. The dimensional modeling methodology covers both, data warehouse schema design and strategies for keeping historical data. Both parts are relevant to the ETL system, because it is its responsibility to bring data in line with the dimensional schema and update the warehouse.

A relational database schema designed according to the rules of dimensional modeling is referred to as *star schema*. Star schemas differ significantly from schemas in third normal-form. Note that the latter are commonly used in operational systems, which may act as data sources for the ETL system. The upper half of Figure 2.13 depicts an entity-relationship (ER) model of the TPC-H benchmark schema [Tra09], which is in third normal-form. The lower half of the same figure shows an ER model of a star schema proposed in the Star Schema benchmark (SSB) [OOCR09]. The SSB schema was derived from the TPC-H schema, i.e. it roughly stores the same information but uses a different schema structure. The dashes lines between the schema suggest the correlation

between the tables in the third normal-form schema and the star schema.

A star schema consists of so-called *fact tables* and *dimension tables*. Fact tables store measures of business processes referred to as *facts*. Facts are usually numeric values that can be aggregated. Dimension tables contain rich textual descriptions of the business entities. The TPC-H benchmark is inspired by the retail business. In this scenario, facts represent order lines of purchase orders and provide measures like the sales quantity and dollar sales amount. Dimensions describe the product being sold, the customers placing orders, the supplier of the products, and the date of the sales transaction.

Data warehouse queries typically use dimension attributes to select, group, and aggregate facts of interest. We emphasize that star schemas are not in third normal-form. Dimensions rather represent multiple hierarchical relationships in a single table. Customer addresses roll up into nations and then into regions, for example. In a star schema, data about (customer) addresses, nations, and regions is stored at the same dimension table. Hence, dimension tables are typically denormalized. The goals of dimensional modeling are improved query performance by reducing the number of joins, user understandability, and resilience to changes that come at the cost of data redundancy. The goal of third normal-form schemas, in contrast, is the avoidance of update anomalies.

Besides the table structure, third normal-form schemas used in operational systems and star schemas used in data warehouses typically differ in the way data is represented at the column level [KR02]. Consider the three tables depicted on the left-hand side in Figure 2.14 taken from the TPC-W benchmark schema [Tra02] that store customer-related data. As it is common for operational systems, they include some "catchall" columns such as ADDR_STREET1, ADDR_STREET2, or C_PHONE. Such columns are often appropriate for operational processing but fail to support analytical queries to better understand and segment the customer base.

A typical warehouse customer dimension [KR02] is shown at the right-hand side of Figure 2.14. Instead of using catchall columns, the information is broken down into as many elemental parts as possible. The catchall street columns in the source schema are split up into street name, number, direction, type, post box, and suite number, for instance. The base data is typically enriched with additional information, such as the gender of a customer or the postal district. Such additional data can be derived from the customer name or postal code and is useful for data analysis. Furthermore, inconsistent formats and abbreviations are standardized. It is the ETL systems that is responsible for standardizing, cleansing, and enriching base data before it is delivered to the warehouse.

Apart from the schema design, the dimensional modeling methodology includes techniques for keeping a history of data changes referred to as *Slowly Changing Dimensions* [KC04, KR02]. For this purpose, so-called *surrogate key columns* are added to dimension tables. Surrogate keys uniquely identify

Customer (shortened)	
C_ID	123456
C_FNAME	Dr. Jane R.
C_LNAME	Smith
C_ADDR_ID	234567
C_PHONE	888-555-3333 x776
C_EMAIL	RJSmith@mail.com
C_BIRTHDATE	14-Sep-1980
...	

Address	
ADDR_ID	234567
ADDR_STREET1	123 Main Rd, North West, Ste 100A
ADDR_STREET2	P.O. Box 2348
ADDR_CITY	Kensington
ADDR_STATE	Ark.
ADDR_ZIP	88887-2348
ADDR_CO_ID	100

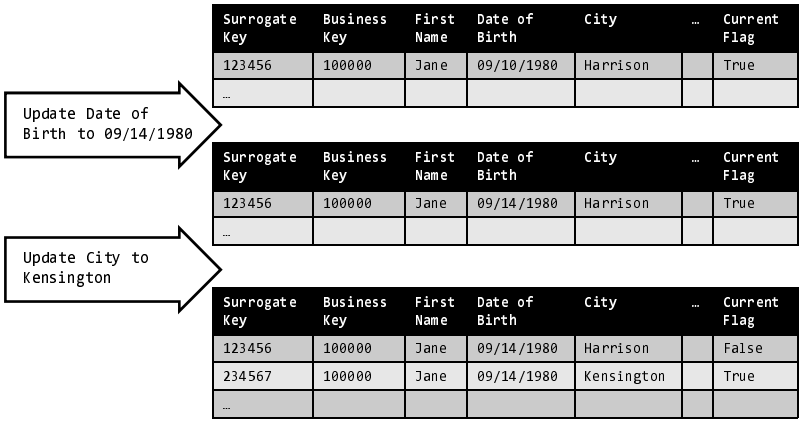
Country	
CO_ID	100
CO_NAME	USA
CO_CURRENCY	US Dollar
CO_EXCHANGE	0,691658597

a) TPC-W customer tables

Customer Dimension	
Surrogate Key	100000
Business Key	123456
Salutation	Ms.
Title	Dr.
Surname	Smith
Middle Initial	R
First name	Jane
Formal Greetings	Ms. Smith
Gender	Female
Email Address	RJSmith@mail.com
Date of birth	09/14/1980
Street Number	123
Street Name	Main
Street Type	Road
Street Direction	North West
Post Box	2348
Suite	100A
City	Kensington
District	Cornwall
State	Arkansas
Country	United States of America
Continent	North America
Primary Postal ZIP Code	88887
Secondary Postal ZIP Code	2348
Telephone Country Code	1
Telephone Area Code	888
Telephone Number	5553333
Telephone Extension	776

b) Typical customer dimension table

Figure 2.14: Operational customer data vs. warehouse customer dimension



Surrogate Key	Business Key	First Name	Date of Birth	City	…	Current Flag
123456	100000	Jane	09/10/1980	Harrison		True
…						

Surrogate Key	Business Key	First Name	Date of Birth	City	…	Current Flag
123456	100000	Jane	09/14/1980	Harrison		True
…						

Surrogate Key	Business Key	First Name	Date of Birth	City	…	Current Flag
123456	100000	Jane	09/14/1980	Harrison		False
234567	100000	Jane	09/14/1980	Kensington		True
…						

Figure 2.15: Slowly Changing Dimension example

dimension tuples and are controlled by the ETL system. Facts reference dimensions by their surrogate keys to establish foreign key relationships. Operational data sources typically manage primary keys, which are referred to as *business keys* in the warehousing context. It is common to assign a unique number to each product in stock, for instance. Business keys are not replaced by surrogate keys. Rather both, the business key and the surrogate key are included in the dimension table as indicated in Figure 2.14.

Surrogate keys permit to keep historical data in warehouse dimension tables that has been overwritten or deleted at the source systems. In the following we will discuss how insertions, updates, and deletions at the source systems are reflected in warehouse dimension tables using the Slowly Changing Dimension technique. Say a new customer is entered at an operational system. In this case a new tuple is created in the dimension table and a fresh surrogate key is assigned. Say a customer tuple is updated at an operational system. How this is reflected at the dimension table typically depends on what columns have been updated. Changing a customers date of birth, for example, can be assumed to be an error correction and hence, the corresponding column at the dimension table is simply overwritten as indicated in Figure 2.15. This case is referred to as Slowly Changing Dimensions type 1.

When the address of a customer is updated, however, it is most likely not an error correction but simply means that she has moved. The former address is kept in the warehouse, because it may still be relevant for any previous orders. This case is referred to as Slowly Changing Dimensions type 2. The basic idea

is to leave the outdated tuple in place and issue a new customer dimension tuple to reflect the new address as shown in Figure 2.15. The new tuple is assigned a fresh surrogate key value, by which it can be distinguished from any expired versions. Each surrogate key value thus identifies a unique customer profile that was true for a span of time. All orders in the fact table refer to such customer dimension tuples that were current at the time the order was placed. For this reason, the type 2 technique is said to partition history in the fact table. Note that the complete customer history can be retrieved by means of the business key that remains constant throughout time. Besides the surrogate key, further "special purpose" columns may be used to reflect type 2 changes. Dimension tuples may be assigned with two timestamps to indicate when the they became effective and when they expired. An additional flag may be used to mark the most current version as shown in Figure 2.14.

When a customer dataset is deleted at the operational system, it typically has continuing presence in the data warehouse. However, it may be marked as having been deleted by setting a flag or an expiration timestamp in the dimension table.

Incremental warehouse maintenance

In this section, we sketch the state-of-the-art in data warehouse maintenance and suggest that there is room for improvement by incremental recomputation techniques.

Facts tables are typically sourced from append-only datasets such as cash register records, telephone call records, or web logs, for instance, and usually only simple data transformations are required to prepare fact data, mainly to resolve syntactical heterogeneity. For these reasons loading fact tables in an incremental fashion is straightforward. The situation is more complex when it comes to dimension tables. Dimension tables are usually sourced from multiple datasets that may be arbitrarily modified (insert, update, and delete) and often reside on different source systems. Furthermore, the preparation of dimension data usually involves complex data transformations and cleansing.

It is common practice "to pull all the dimension data into the ETL system on each incremental load and compare with the existing dimension" [MTK06]. That is, dimension tables are usually recomputed from scratch in each loading cycle. The existing warehouse dimensions cannot simply be replaced with the recomputed dimensions, because any historical data, which is no longer available at the sources, would be lost. Rather, the recomputed dimensions are compared to the existing dimensions to identify changes, which are then incorporated according to the Slowly Changing Dimension technique. A naive field-by-field comparison is discouraged in [KC04], because of efficiency issues and a comparison approach based on cyclic redundancy checksums is proposed

instead. While this technique brings some improvements, Ralph Kimball, a pioneer in data warehousing, notes in this context that "it would be wonderful if only the changes since the last extract, or deltas, were delivered to the staging area, but more typically, the staging application has to find the changed dimensions" [KC04]. He thus advocates a "true" incremental approach to warehouse maintenance. This approach is subject of this thesis.

An incremental approach has two main advantages. First, the repeated computation of unchanged (intermediate) data can be avoided. Second, a costly comparison with existing warehouse data to obtain deltas is no longer necessary, because deltas are directly computed. Both allows for improving the efficiency of warehouse maintenance. According to a study by the International DB2 Users Group (IDUG) roughly 90% of today's production data warehouses are maintained once a day or less frequently [IDU]. Making warehouse maintenance more efficient will allow for refreshing data warehouses more often and hence, provide warehouse users with more timely data. Alternatively, the (nightly) maintenance window can be shrunken, making the warehouse more available to its users. Another option is to perform warehouse maintenance at the same pace but spent less resources (and energy) on this task.

2.5.2 Use case: Data migration

Besides data warehousing, data migration is an area where ETL tools are being widely used. Data migration is crucial when legacy systems are retired or consolidated. The data managed by legacy applications is a valuable asset to companies and needs to be migrated to their modern replacement. Data migration inevitably involves substantial data transformation as virtually every enterprise application system operates on its own information model. Furthermore, data quality in legacy systems is often not sufficient for modern enterprise applications [OMSV11]. Thus, data migration involves substantial data transformation, standardization, cleansing, and possibly deduplication.

An incremental approach to data migration offers interesting opportunities. It allows for an migration scheme in which the legacy systems continue to operate while data migration is in process. When the target system is completely loaded for the first time, the legacy data has been updated by concurrent business transactions. The resulting deltas may be used to incrementally maintain the target system in the next step. Again, the legacy data may be updated concurrently. However the delta set is likely much smaller, because incremental loading generally proceeds faster than the initial load. The process is repeated until the delta set at the legacy system is small enough for it to be quiesced for the final incremental load after which the target system is ready to go into production. That is, an incremental scheme may considerably reduce the downtime of legacy systems during data migration.

2.6 Related work

In this section, we will discuss related work in industry and academia. We divided the related work into four broad categories. We will first discuss techniques for loading data warehouses continuously, which are complementary to our work. We will proceed with a discussion of recent data warehouse architectures known as real-time data warehouses and so-called *one size fits all* systems that do not rely on ETL-style data integration. Finally, we will discuss related work on incremental methods for data integration.

Note that other related work has been discussed within this chapter before. Integration systems offering an alternative to ETL such as (advanced) replication systems and (distributed) materialized view systems have been discussed in Sections 2.3.1 to 2.3.4. The Orchid system, which provided an important basis for our work, has been discussed in Section 2.3.5. Related work on incremental recomputations in MapReduce will be discussed in Section 7.2.

Continuous data warehouse loading Traditionally, data warehouses have been loaded at regular time intervals during nightly batch windows using fast bulk loading techniques. To improve data timeliness in the warehouse, the loading intervals may be shortened. However, bulk loading is not efficient for small batch sizes and the warehouse accessibility may be restricted while loading is in progress. Alternatively, the warehouse may be loaded on a row-by-row basis by issuing INSERT statements through the SQL interface. This approach is sometimes referred as trickle feeding. The problem with trickle feeding is that it interferes with concurrent OLAP queries and hence causes the warehouse performance to degrade.

As a pragmatic solution, it has been proposed to add a so-called real-time partition to a data warehouse [KC04]. A real-time partition consists of additional tables whose structure matches that of the fact tables. Facts are trickle fed into the real-time fact tables throughout the day. To minimize the insert overhead, the real-time fact tables are not indexed. On a regular basis the contents of the real-time fact tables are loaded to the static fact tables. Hence, the real-time fact tables remain relatively small and can be kept in memory. Even though the real-time partition is not indexed, it can thus be queried efficiently.

A technique similar to real-time partition approach is proposed in [SB08]. The basic approach is extended to trickle feed not only fact tables but also dimension tables. Furthermore rules are given to rewrite OLAP queries such that both static and real-time tables are transparently accessed together.

A middleware approach to support continuous data warehouse loading called RiTE is proposed in [TPL08]. The RiTE system consists of specialized JDBC drivers for warehouse writers and readers and a so-called catalyst. The catalyst is an in-memory storage system that buffers warehouse insertions. It allows for

trickle feeding the warehouse at bulk-load speed and offers concurrency control. The bulk movement of data from the catalyst to the warehouse is controlled by user-defined policies. Data stored in the catalyst can be transparently accessed by OLAP queries through the RiTE JDBC driver.

In [TBL09, TL09], the scheduling of queries and updates at a data warehouse is discussed. Scheduling updates before queries generally improves the data freshness but increases the query response time and vice versa. The authors propose a multi-objective scheduling approach that finds the optimal schedule for given user requirements with regard to both criteria.

Related work on continuous warehouse loading is complementary to our work on incremental ETL processing described in this thesis. Incremental ETL processing allows to efficiently propagate updates from the operational source systems to the data warehouse. Techniques for continuous loading allow for applying these updates to the warehouse in a timely and efficient manner.

Real-time data warehousing There is a recent trend towards *real-time data warehousing* in the data integration industry. The promise of real-time data warehousing is a low-latency propagation of updates from the operational systems to the data warehouse. Current real-time data warehousing offerings roughly fall into two categories.

First, there are solutions built on top of a specific ERP or CRM system. These systems use custom extractors and hard-wired change propagation logic tailored to the respective operational source system. A prominent representative of this category of real-time warehousing solutions is SAP NetWeaver Business Intelligence, which is closely coupled with SAP ERP. The problem we study in this thesis is somewhat more general. In particular, we will make minimum assumptions regarding the operational source systems, their schemata, and their change data capture capabilities.

The second category of real-time warehousing solutions has its roots in database replication. We introduced this class of integration systems in Section 2.3.2 and referred to them as advanced replication systems. As said, advanced database replication systems have a strong emphasis on fast data propagation but are restricted in terms of their data integration capabilities. This is quite contrary to ETL tools that have a strong emphasize on data integration but rely on periodic, batch-style data processing. In this thesis we propose to leverage ETL tools for incremental update propagation. While our solution will not beat the latency of replication-style update propagation, it inherits the advanced data integration capabilities of ETL tools and is thus applicable in heterogeneous environments where advanced replication is not.

One size fits all database systems Today transactional workloads (OLTP) and analytical workloads (OLAP) are typically processed by separate database systems. That way, each system can be optimized to its respective workload pattern and data contention problems are avoided. The term "one size fits all" is used for systems that address mixed OLTP and OLAP workloads. The promise of one size fits all databases is to make a separated data warehouse and its ETL system superfluous.

A one size fits all architecture has obvious advantages. There is no need to move data from operational systems to a separate warehouse system. Thus OLAP queries are always evaluated on the most current data. Furthermore, the resource consumption for maintaining two separate systems may be reduced.

The HyPer database system is a one size fits all research prototype [KN11]. HyPer stores all transactional data in main memory and guarantees the ACID properties by processing OLTP transaction serially. OLAP queries are evaluated on virtual memory snapshots that are created through calling the fork function provided by the operating system. Forking creates a new operating system process with shared memory. The consistency of OLAP memory snapshots is ensured by the hardware-supported copy-on-write technique. When a memory page is modified by the OLTP process, the copy-on-write technique creates a separate copy of that page and redirects all future references of the OLTP process to it. In this way, the OLAP memory snapshot remains stable and consistent.

The OctopusDB concept suggests a unified, one size fits all data processing architecture for OLTP, OLAP, and data streaming systems [DJ11]. OctopusDB is not built around a central store but uses a logical event log as its primary storage structure. Different types of so-called storage views are defined on that log to represent (parts of) it in different physical layouts. Storage views provide a unified way to model database concepts such as point-in-time and continuous queries, row- and column stores, materialized views, and index structures. Thus, OctopusDB may emulate different types of data management systems.

One size fits all databases are still in their early stages. However, it is an interesting question whether ETL systems and separated data warehouses will be obsolete once these systems mature. We doubt that this will be the case, because the one size fits all concept does not cover all aspects of data warehousing. Data warehousing goes beyond OLAP query processing and, most importantly, addresses data integration needs. It is used to integrate data from multiple heterogeneous source systems and ensure data quality standards. According to [RL11], data integration was the main driver to introduce the concept of data warehousing and this requirement is still valid today. We thus believe that ETL and data warehouses are here to stay.

Incremental methods for data integration The Stuttgart Information and Exploration System (SIES) has been proposed for propagating changed data between autonomous and heterogeneous systems [CHRM02]. SIES allows for the specification of so-called propagation dependencies between related data objects in different systems. Propagation dependencies are associated with transformation scripts to translate between heterogeneous data representations. Based on propagation dependencies, SIES automatically tracks changes at source systems, transforms and filters changed data, and propagates it to the target system.

An important difference between SIES and our work on incremental ETL processing is that SIES uses message-oriented data exchange and propagation dependencies establish pairwise relationships between systems. Our work leverages the data integration capabilities of ETL tools. It furthermore allows for many-to-one source to target dependencies (such as cross-system joins) that are common in data warehouse scenarios.

The approach to incremental ETL processing described in this thesis has been patented by IBM [JMS09]. An intensive search for similar techniques was conducted by a patent attorney and discovered a single directly related paper with the title "Generating Incremental ETL Processes Automatically" [ZSW+06]. Like this thesis, it proposes the automatic derivation of incremental ETL jobs based on initial ETL jobs. The authors focus on the relational difference operation and claim that this operation, while commonly used in ETL processing, had not been considered in incremental view maintenance before. We do not agree with them in this point. IBM InfoSphere DataStage, which is a leading ETL tool, does not provide a difference operator, which suggests a rather minor importance in ETL processing. Unlike suggested by the authors, previous work on view maintenance such as [GLT97] did cover difference views. While [ZSW+06] presents some new ideas on the self-maintainability of difference views, we feel that it does not align well with the ETL environment. It neglects important characteristics such as the importance of data standardization and cleansing, the autonomy of data sources, and maintenance anomaly issues that will be discussed in the remainder of this thesis.

3 An approach to incremental ETL processing

In this section, we will sketch the basics of our approach to facilitate incremental recomputations in an ETL environment. Abstractly speaking, we follow the algebraic differencing approach, which has been discussed in Section 2.3.3. Algebraic differencing was originally developed for the maintenance of materialized views. The basic idea is to differentiate a view definition given as a relational algebra expression, to derive incremental relational algebra expressions that compute the change to the materialized view. We propose to differentiate ETL jobs to obtain incremental ETL job variants that compute the change to the warehouse tables. Our approach has been patented in collaboration with the IBM Research and Development Lab, Böblingen in 2009 [JMS09].

The differentiation of an ETL job is illustrated in Figure 3.1. The top of the figure shows a sample ETL job for loading a customer dimension at a data warehouse. The ETL job extracts source data exhaustively from two source tables, which store customer and address data and may reside on different systems. Without going into much detail, the data is standardized, cleansed, and joined together hereafter, before it is loaded to the warehouse dimension. The ETL job computes the dimension from scratch and can hence be used to perform the initial computation and subsequent full recomputations.

At the bottom of Figure 3.1 another ETL job is shown that has been obtained by differencing the initial ETL job. It is an incremental variant that consumes deltas captured at the sources and computes deltas to be applied to the dimension table. We will discuss both ETL jobs and the process of differencing in greater detail in the remainder of this section. However, one can make two important observations at first glance.

First, the incremental ETL job variant is composed of standard ETL processing operators of the respective ETL tool. In the example, the ETL tool IBM InfoSphere DataStage [Inf] is used, which is referred to as DataStage for short in the following. Note that the incremental job variant contains additional ETL operators, or *stages* as they are referred to in DataStage, that do not appear in the initial ETL job. These operators are used to perform basic set operations. The Funnel stage combines several input datasets into a single output dataset and thus, essentially computes a set union. The Change Capture stage can be used to compute a set difference. The Copy stage is used to fork the ETL data flow and feed the output of a single stage into multiple down-stream stages. The Remove Duplicates stage performs duplicate elimination.

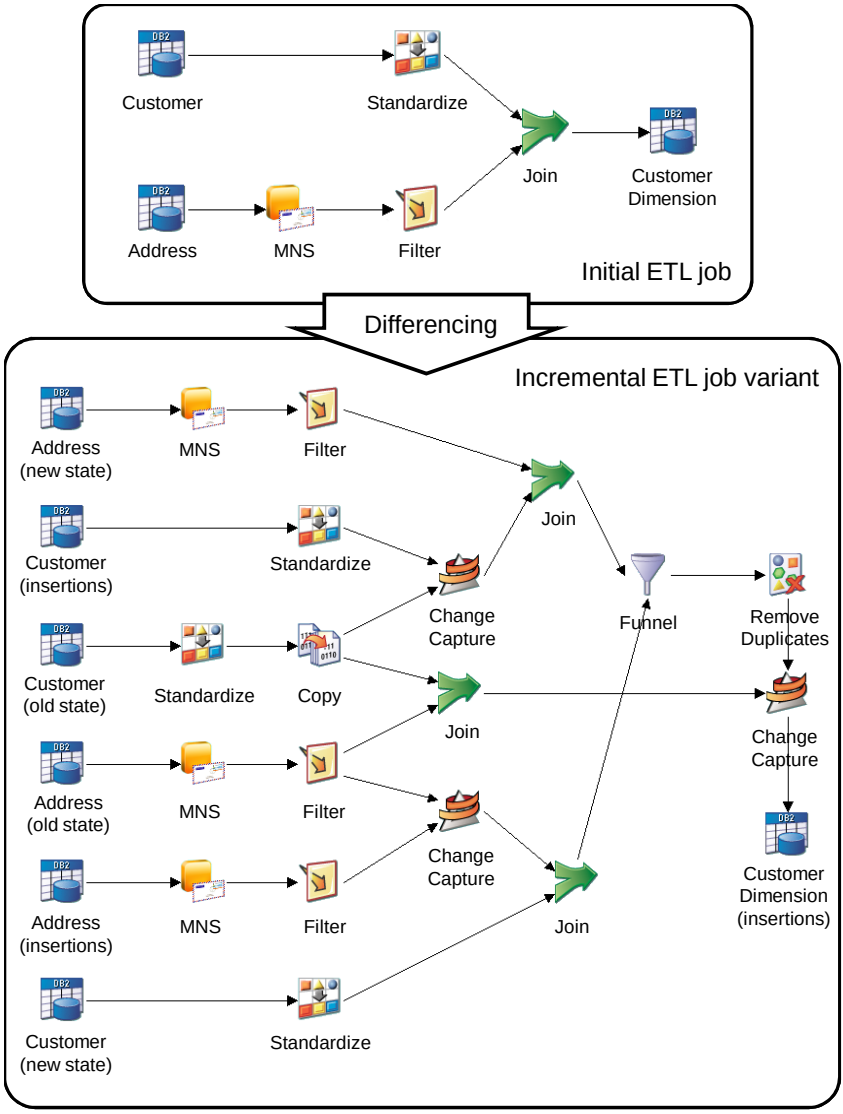


Figure 3.1: Differencing an ETL job

Funnel, Change capture, Copy, and Remove Duplicates are standard stages in DataStage. That is both, initial and incremental ETL job are expressed in the same "language" or, in other words, composed from the same processing primitives. This is analogous to algebraic differencing, by which view definitions in relational algebra are rewritten to incremental expressions in, again, relational algebra. An approach that is closed with respect to the set of transformation operators has a couple of advantages: First and foremost, incremental ETL jobs can be executed by standard ETL tools without any modifications. It is hence easily possible to "incrementalize" existing ETL jobs. Apart from the execution engine, it is furthermore possible to continue to use any additional ETL tooling. As will be discussed in the following, we were able to leverage the Orchid framework for ETL job deployment.

The second observation that can be made from Figure 3.1 is that the incremental ETL job variant is more complex than the initial ETL job. The aim is an efficiency improvement through the avoidance of redundant computations, which often requires a more complex job design. However, our approach allows to automatically derive incremental ETL job variants and thus, hides the complexity from ETL developers. To facilitate the differentiation of ETL jobs, we need to establish an abstract ETL operator model. We will propose a suitable model in Section 3.1 and discuss the process of differencing in Section 3.2.

3.1 An ETL operator model

We reviewed three independent efforts that provide a platform- and vendor-independent view on ETL transformation semantics in Section 2.3.5, namely the TPC-ETL benchmark draft, the Orchid system, and work on ETL modeling by Simitsis and Vassiliadis et al. These efforts suggest that ETL transformations have a "relational core" in the sense that ETL transformations can, at least partly, be described by relational algebra operators, though all ETL tools we are aware of use different proprietary operator models.

Out of the three efforts, the Orchid project is the only attempt towards establishing a concrete platform-independent model for ETL transformation semantics. As said, the proposed model extends and generalizes the relational algebra and is referred to as Operator Hub Model (OHM). OHM provided the basis for the ETL operator model that will be used for algebraic differencing in this thesis.

However, OHM has some limitations in expressing data quality-related transformation. Recall the phases of materialized data integration that have been described in Section 2.2.2. Figure 3.2 shows the phases that are covered by OHM. Note that technical heterogeneity is resolved in the extraction phase and data model heterogeneity may also be resolved in this phase. In its current

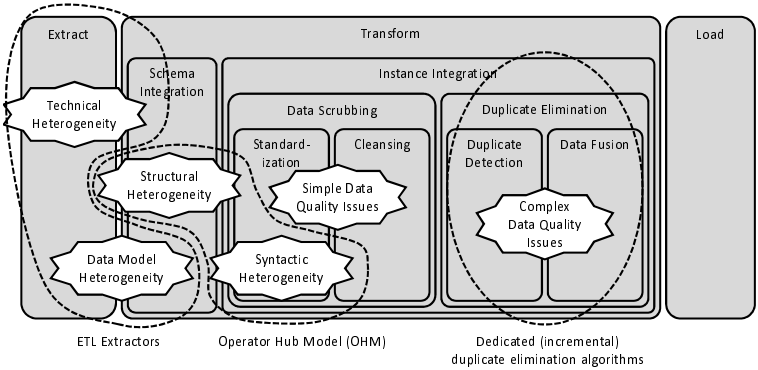


Figure 3.2: Phases of materialized data integration (revisited)

form, OHM covers the transformation phase only partly. It allows for expressing transformations to resolve heterogeneity at the schema level, i.e. structural heterogeneity. Furthermore, the generalized notion of the projection operator allows for expressing value transformations to resolve syntactical heterogeneity, e.g. date format conversions or currency translations.

The other phases of instance integration, i.e. data cleansing, duplicate detection, and data fusion, cannot be expressed by means of OHM. These data quality-related transformations are typically very domain-specific and inherently fuzzy. The algorithms are often complex and differ largely from tool to tool, as do the domain ontologies that are being used. Note that the TPC-ETL committee intends not to include any data quality-related transformations in the benchmark for these reasons. Furthermore, data cleansing and duplicate elimination algorithms can typically be configured in many ways, e.g. by adjusting threshold values or choosing from various distance functions. For all these reasons, it is challenging to capture data quality-related transformations in an operator model such as OHM without making it overly complex. Moreover, there is no point in capturing transformation semantics that is not common to most ETL tools in a model that is intended to be platform-independent.

We aim at facilitating incremental recomputations in the overall ETL process, i.e. we need to consider incremental processing in each phase of materialized data integration. The question is thus which of these phases needs to be covered by an abstract ETL operator model for algebraic differencing. We start our discussion with the duplicate elimination phase. Recall that this phase is typically the very last step of instance integration, because it requires its input data to be standardized and cleansed first.

Duplicate elimination algorithms perform pairwise comparisons of tuples to identify groups that correspond to the same real-world object. To reduce the number of comparisons, the base data is typically segmented into disjoint partitions or a sliding window is applied. Sometimes multiple passes are performed and the individual results are combined by computing the transitive closure. Duplicate elimination algorithms are thus specialized algorithms to solve a very particular problem. Dedicated *incremental* duplicate elimination algorithms have been proposed in literature, e.g. [HS98] presents an incremental version of the so-called sorted-neighborhood method.

Incremental duplicate elimination algorithms merge deltas with an existing, previously de-duplicated dataset. Similar to initial duplicate elimination, incremental duplicate elimination requires its input deltas to be standardized and cleansed and structural heterogeneity to be resolved in the upstream ETL process. In this work, we are thus interested in "incrementalizing" ETL jobs up to the duplicate elimination to provide it with appropriate input deltas. We hence focus on the schema integration phase, the data standardization phase, and the data cleansing phase.

Recall that OHM just covers the schema integration and the data standardization phases but does not cover the data cleansing phase. Thus, OHM leaves a gap in the overall ETL process model. As said, extending OHM to close this gap is challenging, because data cleansing operators are inherently fuzzy and rather diverse. However, the goals of our work are somewhat different from the goals of the Orchid project. OHM is intended to be a truly platform-independent model in the sense that the entire transformation semantics are captured in an abstract way. In particular, OHM instances contain all the information needed to generate and deploy executable ETL jobs to different ETL execution platforms.

In this work, however, we aim at differencing ETL jobs to obtain an incremental variant and need an abstract ETL operator model to support this task. We stress that such a model does not need to capture any details that are not relevant for algebraic differencing. Thus, in contrast to the use cases addressed by Orchid, it is often not necessary to capture an operator's transformation semantics entirely. We rather need to understand, how to difference ETL jobs to obtain an incremental variant. An incremental variant is essentially composed from the operators that appear in the original ETL job. Roughly speaking, these operators are reassembled according to certain rules, which are referred to as delta rules. We will describe the process of differencing in more detail in the next section and give an example.

Recall the delta rules by Griffin et al. for algebraic view maintenance discussed in Section 2.3.3. Different delta rules apply to different types of relational operators, e.g. there are dedicated delta rules for projection, selection, and join operators. The formulation of delta rules obviously requires an un-

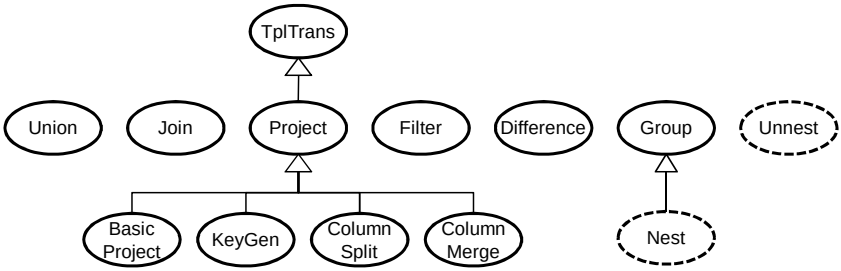


Figure 3.3: Generalized OHM operator set

derstanding of the transformation semantics of the operators. However, the transformation semantics need to be understood only to a certain extent. Consider the delta rules for relational selection. Note that it is not required to "understand" the selection predicate. The predicate is simply copied to the corresponding selection operators in the derived incremental expression. What matters is that the predicate is evaluated in the same way in both, the original expression and the incremental expression.

In much the same way, it is possible to formulate delta rules for data cleansing operators without understanding the very detail of their transformation semantics. Recall that data cleansing operators deal with so-called simple data quality issues such as missing values, misfielded values, misspellings, or embedded values. Simple data quality issues can be identified and resolved by examining a single tuple at a time. From an abstract point of view, data cleansing operators transform the attribute values of the input tuples in arbitrary ways, potentially change the attribute schema, and output exactly one result tuple per input tuple. Note that data cleansing operators behave much like relational projection operators in this respect.

Understanding the transformation semantics at this rather high level of abstraction is sufficient to formulate appropriate delta rules for algebraic differencing. In fact, data cleansing operators can be treated similar to relational projection operators in the differencing process, i.e. the delta rules for relational projection are directly applicable. However, while incremental ETL job variants derived using these delta rules are correct, they may not be efficient. We will see that data cleansing operators have a distinct runtime behavior, which cause traditional optimization rules to fail. We will discuss this issue in detail in Chapter 4 and propose advanced delta rules and optimization strategies.

The abstract ETL operator model used in this thesis is a generalization of OHM. It includes an additional operator referred to as TPLTRANS (tuple

transformer) that subsumes all operators that transform a single tuple at a time and emit exactly one output tuple per input tuple. It can thus be regarded as a further generalization of the OHM PROJECT operator. In contrast to the original OHM operators, its exact transformation semantics remain opaque. Nevertheless, such an operator is useful for our purposes, because the transformation semantics captured are sufficient for algebraic differencing. We thus trade cross-platform portability for a more complete coverage of ETL operators.

The operators of the generalized OHM are depicted in Figure 3.3. The generalization hierarchy among the operators are indicated by UML-like inheritance arrows. Note that OHM as defined in [DHW+08] did not include a set difference operator, which has been added. The OHM operators NEST and UNNEST have been included for the sake of completeness. However, non-first normal-form extensions to the relational data model will not be considered in this thesis.

3.2 Algebraic differencing of ETL jobs

In this section, we will illustrate the basic process of differencing ETL jobs to obtain an incremental variant. We will use standard delta rules discussed in Section 2.3.3 that have been proposed for the incremental maintenance of materialized views in database systems. Let us point out already here that these rules make some silent assumptions that are valid for database views but do not hold in an ETL environment. Thus, there are some flaws in the incremental ETL job we obtain. We will examine these issues in the subsequent chapters and propose solutions that work in an ETL environment.

Reconsider the sample ETL job depicted at the top of Figure 3.1. Assume that the source and target schemas resemble the schemas shown in Figure 2.14 in Section 2.5 on dimensional modeling. The ETL job extracts data from two source tables storing information on customers and postal addresses. It computes the customer dimension using several ETL transformation stages.

After the address data is extracted, it is passed to a so-called Multinational Address Standardization (MNS) stage, which is a domain-specific data cleansing operator provided by DataStage. Recall that the data warehouse has typically higher data quality requirements than the operational source systems. Furthermore, the address data is typically structured differently in the warehouse dimension. In operational source schemas, address data is commonly represented using "catchall columns" such as address line 1, address line 2, etc. and we thus face the data quality issue of embedded values. In the warehouse dimension schema, embedded values are broken down into their elemental parts. Catchall address lines would be split up into street name, street number, post box, and suite number, etc. for instance. Apart from separating embed-

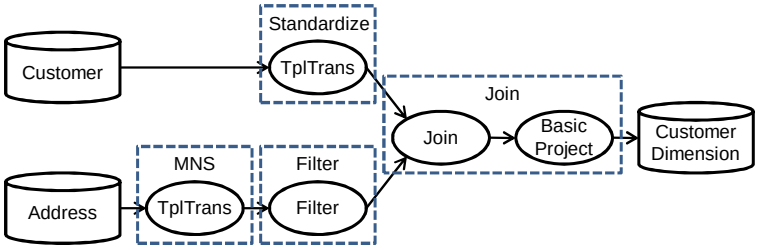


Figure 3.4: Abstract representation of the sample ETL job

ded values, the address cleansing operator does spelling corrections and adds missing information if possible, such as city names for given zip codes.

After the address data has been cleansed, it is fed into a so-called Filter stage that selects addresses from European countries and discards all others. The rationale behind this is that only European customers should be included in the warehouse dimension table.

Consider the upper branch of the sample ETL job. Customer data is extracted from the source table and passed to a so-called Standardize stage, which is another data cleansing operator provided by DataStage. The Standardize stage can be configured with domain-specific cleansing rule sets that are shipped with the ETL tool. In the sample job it is used to address several data quality issues and forms of heterogeneity typically found in customer data. First, the formatting of dates and email addresses is standardized, i.e. syntactical heterogeneity in the source data is resolved. Furthermore, the formatting of customer telephone numbers is standardized. Note that the latter is not a form of syntactical heterogeneity, because the source columns do not have a common format. The formatting of telephone number is rather likely to differ from row to row. The Standardize stage furthermore compensates for missing information by guessing the gender of customers from their first name. Finally, it parses out embedded values in the name columns, such as a title and the middle initial.

After the address data and the customer data has been cleansed, it is joined together along a foreign-key relationship using a Join stage. The Join stage discards the address key column from its output schema, because this column is not included in the customer dimension schema. Dimension schemas typically do not contain any source key columns apart from the business key column, which is the customer key in this example. The output data of the Join stage is loaded into the customer dimension table at the data warehouse. Alternatively, a final duplicate elimination step may be performed at this point.

	$\mathcal{E}$	$\Delta(\mathcal{E})$	$\nabla(\mathcal{E})$
1	R	ΔR	∇R
2	$\sigma_p(\mathcal{S})$	$\sigma_p(\Delta(\mathcal{S}))$	$\sigma_p(\nabla(\mathcal{S}))$
3	$\pi_A(\mathcal{S})$	$\pi_A(\Delta(\mathcal{S})) - \pi_A(\mathcal{S}_{old})$	$\pi_A(\nabla(\mathcal{S})) - \pi_A(\mathcal{S}_{new})$
7	$\mathcal{S} \bowtie \mathcal{T}$	$[\mathcal{S}_{new} \bowtie \Delta(\mathcal{T})] \cup [\Delta(\mathcal{S}) \bowtie \mathcal{T}_{new}]$	$[\mathcal{S}_{old} \bowtie \nabla(\mathcal{T})] \cup [\nabla(\mathcal{S}) \bowtie \mathcal{T}_{old}]$

Table 3.1: Delta rules by Griffin et al. (not complete)

To derive an incremental variant from the sample ETL job, it is first translated into an abstract model representation. As discussed before, our abstract ETL operator model generalizes OHM. Thus, translating ETL jobs into abstract model instances is done in a way similar to the Orchid approach. The Orchid system converts ETL jobs into OHM instances by compiling each vendor-specific ETL stage into one or more OHM operators. The result of this compilation is a sequence of OHM subgraphs, which are connected together to form the OHM representation of the job [DHW+08].

Figure 3.4 depicts the abstract model instance that is derived from the sample ETL job. Note that the Join stage is compiled into more than one abstract ETL operators. It is expressed in terms of an abstract JOIN operator followed by a BASIC PROJECT operator. The latter is used to discard the address key column, which is not included in the customer dimension schema. The ETL cleansing operators used in the sample job are represented by an abstract TPLTRANS operator. As said, the TPLTRANS operator further generalizes the OHM PROJECT operator and subsumes all operators that transform a single tuple at a time and emit exactly one output tuple per input tuple.

The abstract operator graph depicted in Figure 3.4 provides the input for algebraic differencing. Alternatively, it can be represented by an algebra expression, which is a more concise representation and therefore preferred in this thesis. The sample ETL job is represented by the following algebra expression. Recall that by convention we use calligraphic letters to denote relational expressions while both base relations and derived relations are denoted by ordinary letters.

$$\mathcal{D} := \pi_{DS}(\sigma_p(\Pi_{Addr}(A)) \bowtie \Pi_{Cust}(C)$$

In the above expression, A denotes the address table, C denotes the customer table, DS denotes the schema of the customer dimension, and p denotes a predicate that evaluates to true for European addresses. Note that the TPLTRANS operator is denoted by a capital Pi (Π), while the basic projection is denoted by a lowercase Pi (π).

Algebraic differencing is done by recursively applying delta rules to the original algebra expression. We introduced a set of standard delta rules by Griffin et al. in Section 2.3.3. For the reader's convenience the delta rules required in the example are repeated here in Table 3.1.

$$
\begin{aligned}
\Delta\mathcal{D} = \quad & \Delta(\pi_{DS}(\sigma_p(\Pi_{Addr}(A)) \bowtie \Pi_{Cust}(C))) \\
& \qquad\qquad\qquad\qquad :=\mathcal{H} \\
\equiv^3 \quad & \pi_{DS}(\Delta\overbrace{(\sigma_p(\Pi_{Addr}(A)) \bowtie \Pi_{Cust}(C))}) - \\
& \pi_{DS}(\sigma_p(\Pi_{Addr}(A_{old})) \bowtie \Pi_{Cust}(C_{old}))
\end{aligned}
$$

$$
\begin{aligned}
\Delta\mathcal{H} = \quad & \Delta(\sigma_p(\Pi_{Addr}(A)) \bowtie \Pi_{Cust}(C)) \\
\equiv^7 \quad & [(\sigma_p(\Pi_{Addr}(A_{new}))) \bowtie \Delta(\Pi_{Cust}(C))] \cup \\
& [\Delta(\sigma_p(\Pi_{Addr}(A))) \bowtie (\Pi_{Cust}(C_{new}))] \\
\equiv^{3,2} & [(\sigma_p(\Pi_{Addr}(A_{new}))) \bowtie (\Pi_{Cust}(\Delta C) - \Pi_{Cust}(C_{old}))] \cup \\
& [(\sigma_p(\Delta(\Pi_{Addr}(A)))) \bowtie (\Pi_{Cust}(C_{new}))] \\
\equiv^2 \quad & [(\sigma_p(\Pi_{Addr}(A_{new}))) \bowtie (\Pi_{Cust}(\Delta C) - \Pi_{Cust}(C_{old}))] \cup \\
& [\underbrace{(\sigma_p(\Pi_{Addr}(\Delta A) - \Pi_{Addr}(A_{old})))}_{\text{selection pushdown possible}} \bowtie (\Pi_{Cust}(C_{new}))]
\end{aligned}
$$

The incremental expression obtained by algebraic differencing can be further optimized by applying standard rewrite optimization rules. In the example, a selection pushdown through a difference operator is possible, as indicated. We finally obtain the following expression.

$$
\begin{aligned}
\Delta\mathcal{D} = & [(\sigma_p(\Pi_{Addr}(A_{new}))) \bowtie (\Pi_{Cust}(\Delta C) - \Pi_{Cust}(C_{old}))] \cup \\
& [(\sigma_p(\Pi_{Addr}(\Delta A)) - \sigma_p(\Pi_{Addr}(A_{old}))) \bowtie (\Pi_{Cust}(C_{new}))] - \\
& \pi_{DS}(\sigma_p(\Pi_{Addr}(A_{old})) \bowtie \Pi_{Cust}(C_{old}))
\end{aligned}
$$

The expression $\Delta\mathcal{D}$ computes the insertions into the customer dimension. An expression $\nabla\mathcal{D}$ to compute the deletions from the customer dimension can be derived in much the same way. A graph representation of the incremental expression $\Delta\mathcal{D}$ is shown in Figure 3.5. Note that common subexpressions in $\Delta\mathcal{D}$ have been eliminated in the abstract operator graph. OHM includes a so-called SPLIT operator that allows for passing input data to multiple upstream operators. There are two common subexpression in $\Delta\mathcal{D}$ and thus two SPLIT operators are introduced to duplicate the result data.

In the deployment step, an executable ETL job is obtained from the abstract operator graph. Because our abstract ETL operator model generalizes OHM, the deployment of model instances into native ETL jobs can be done in a way

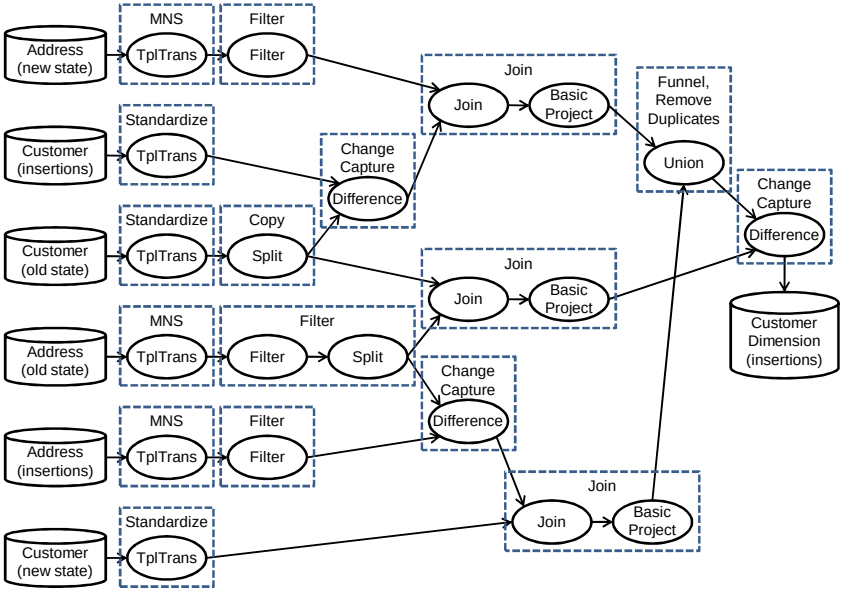


Figure 3.5: Abstract representation of the sample ETL job

similar to the Orchid approach. The Orchid algorithm finds a combination of native ETL operators that covers the abstract operator graph entirely. Orchid uses so-called OHM graph patterns to describe the transformation capabilities of each native operator. Orchid matches these graph patterns against the abstract operator graph. To find a "good" ETL job design, Orchid uses a simple heuristic. It prefers jobs that use as few native operators as possible. The intention is to exploit the transformation capabilities of each operator as much as possible, which is expected to result in better performance characteristics. A detailed description of the Orchid deployment algorithm can be found in [DHW+08].

In the example at hand, Orchid decides to implement multiple abstract operators by a single native operator in four cases as indicated in Figure 3.5. All three JOIN operators are followed by a BASIC PROJECT operator and can thus be expressed by a Join stage in DataStage. Furthermore the Filter stage is capable of passing its output data to multiple upstream operators and can thus be used to implement a FILTER operator together with a subsequent SPLIT operator. Note that the Standardize stage cannot pass its output data to more

than one upstream operator. For this reason Orchid chooses to implement the subsequent abstract SPLIT operator by a separate native Copy stage.

Recall that OHM does not include the TPLTRANS operator. In contrast to the other OHM operators, the transformation semantics of this operator are not fully expressed at an abstract level. However, in the deployment process each TPLTRANS operator can be replaced with an exact copy of the corresponding native cleansing operator in the original ETL job. It is thus required to maintain the link between each TPLTRANS operator and its native counterpart when the original ETL job is translated into an abstract model instance in the first place, but this is straightforward. The deployed ETL job is depicted at the bottom of Figure 3.1.

Note that only a subset of the abstract ETL operators proposed in Section 3.1 appeared in the above example. Nevertheless, algebraic differencing works for ETL jobs including the remaining relational algebra-like OHM operators UNION and DIFFERENCE using the delta rules presented in Section 2.3.3. Similarly, ETL jobs that aggregate data and therefore include GROUP operators can be differenced using delta rules proposed in [Qua96, GM06]. We refrain from presenting these delta rules here, because we will not elaborate much on aggregations in the ETL context in the following chapters. While data aggregation is undoubtedly an important operation in ETL processing, we will focus on challenges that are unique to the ETL environment. Recomputing aggregates incrementally has been researched in the context of materialized view maintenance and, as we experienced, can be done in much the same in the ETL environment. Since we do not contribute to this area, we refrain from repeating previous work.

3.3 Research challenges in incremental ETL processing

The incremental ETL job obtained by algebraic differencing is *correct*, i.e. the customer dimension table reaches the exact same state after incremental maintenance that a full recomputation would have produced. A proof of correctness for the set of delta rules is given in [GLT97]. However, as suggested at the beginning of this section there are some flaws in the incremental ETL job that will be addressed in the following chapters of this thesis.

- Unlike traditional database views, ETL solutions typically have a strong emphasis on data integration and thus, standardizing heterogeneous data and improving data quality are key tasks. For this purpose, ETL tools provide rich sets of data cleansing operators. These operators perform complex data transformations that are usually computationally expensive.

In Chapter 4 we will show that traditional relational optimization techniques often struggle in the presence of data cleansing operators. Surprisingly, the incremental variant derived from the sample ETL job in this section, does not achieve an efficiency improvement compared to a full recomputation. The reason for the sample incremental job to perform badly is that it performs excessive data cleansing. We will propose optimization techniques that effectively reduce the data cleansing effort during incremental recomputations.

- In the algebraic view maintenance approach, deltas are modeled as two sets, i.e. the set of inserted tuples and the set of deleted tuples. Updates are modeled implicitly as delete-insert pairs. In particular, it is assumed that the deltas are complete, e.g. it is assumed that both, the new and the old state of updated tuples are known. However, in a warehousing environment, so-called Change Data Capture (CDC) techniques are used to gather deltas at the source systems and an important difference to traditional view maintenance is that these deltas are often incomplete (or partial). This is for two reasons. First, a number of different CDC approaches is used in practice – a survey will be presented in the beginning of Chapter 5 – and some CDC techniques are simply incapable of capturing complete deltas. Second, it is often more efficient to capture deltas only partially instead of completely. In this way, the load on the operational source systems may be decreased.

  Standard delta rules proposed for algebraic view maintenance are built on the assumption that deltas are completely available. We will show that these rules fail for partial deltas in Chapter 5 and propose a generalized delta model and adapted delta rules hereafter.

- The incremental expressions obtained by the standard delta rules rely on transactional guarantees. Several delta rules, such as the rule for relational joins, require access to both, the deltas and the base data itself. It is assumed that these datasets are accessed in the scope of a transaction and thus synchronized with concurrent updates. Unsynchronized access may cause so-called maintenance anomalies that have the potential to corrupt the materialized view irrecoverably.

  In the ETL environment, the operational source systems and the warehouse are distributed. In a distributed environment, maintenance anomalies can be avoided using distributed transactions. However, distributed transactions are most often prohibitively expensive and holding locks on base data for the period of warehouse maintenance is usually unacceptable. Moreover, the source systems in an ETL environment may be unsophisticated and unable to participate in distributed transactions. We

discuss maintenance anomalies in the context of data warehousing in Chapter 6 and propose approaches to avoid such anomalies that do not rely on distributed transactions.

4 Optimization of incremental ETL jobs

We proposed an approach for generating incremental ETL jobs in Chapter 3 that is inspired by algebraic differencing for incremental view maintenance. As suggested earlier, incremental ETL jobs obtained in the way described in this chapter are not without issues. The optimization of incremental ETL jobs will be discussed in this chapter, which is based on our work published in [BJ10].

As we will show in Section 4.1, incremental jobs obtained by algebraic differencing are often inefficient and fail to outperform a naive full recomputation approach. To fix this issue, we will propose a refined set of delta rules for algebraic differencing tailored to the specifics of ETL in Section 4.2. After a discussion of some remaining shortcomings in Section 4.3, we will show in Section 4.4 that the Magic Sets method, which was originally proposed in the context of recursive logical programs, is effective for the optimization of incremental ETL jobs that include data cleansing. We will conclude the chapter in Section 4.5.

4.1 Incremental recomputation issues in ETL

Throughout this chapter, we will use a running example to illustrate the ETL-specific optimization techniques proposed in the following. Consider the sample ETL job depicted in Figure 4.1. It is a slight variation of the sample job presented in Chapter 3. In particular, the underlying base tables are taken from the TPC-W benchmark schema used in our experiments.[1].

The sample job extracts data from the TPC-W customer, address, and country tables, which are denoted as C, A, and N (as in nation), respectively, in the sequel. A is joined with N along a foreign-key relationship using a so-called Lookup stage, which is a hash join implementation provided by DataStage. The result data is cleansed hereafter, using an address standardization stage. Another data cleansing stage is used to standardize the formatting of telephone numbers in the customer data extracted from C. The cleansed datasets are joined together and finally loaded into the warehouse customer dimension

[1]The TPC-W benchmark simulates a web-based e-commerce system. Since operational systems of this kind often serve as data sources for warehouse systems, we chose the TPC-W benchmark schema for our experiments. Another reason for our decision was the availability of a data generation utility.

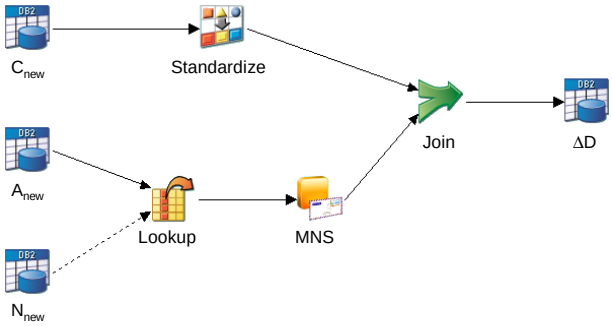


Figure 4.1: ETL job computing a customer dimension table from TPC-W benchmark tables

table. The sample ETL job is represented by the following algebra expression.

$$\mathcal{D} := \Pi_{Addr}(A \bowtie N) \bowtie \Pi_{Tel}(C)$$

As discussed earlier in Section 2.2.3, there are two fundamental approaches to maintain materialized integrated data such as the customer dimension in the example. It may either be fully recomputed or maintained incrementally. Recall that warehouse tables cannot be simply dropped and rebuilt, because any historical data would be lost, since it is usually no longer available at the sources. To correctly keep the data history, the recomputed data rather needs to be compared to the current warehouse content to determine the set of changes. For this reason, ETL jobs for the initial computation (such as the sample job depicted in Figure 4.1) cannot be reused for full recomputations as they are. In fact, an additional final step is required, in which the recomputed data is compared to the warehouse content and the data history is updated. Commercial ETL tools usually provide dedicated operators for this purpose. IBM DataStage, for instance, offers the so-called Slowly Changing Dimension (SCD) stage.

A variant of the initial ETL job that uses a Slowly Changing Dimension stage to perform a full recomputation is shown in Figure 4.2. The SCD stage is added as a final processing stage right before the database adapter. It consumes the output of the original ETL job and, additionally, the warehouse customer dimension table in its current state. These datasets are compared in a process similar to snapshot differentials (see Section 5.1). From the differences found in these datasets, the SCD stage generates a set of SQL insert and update statements to update the dimension table according to the SCD method (see 2.5.1).

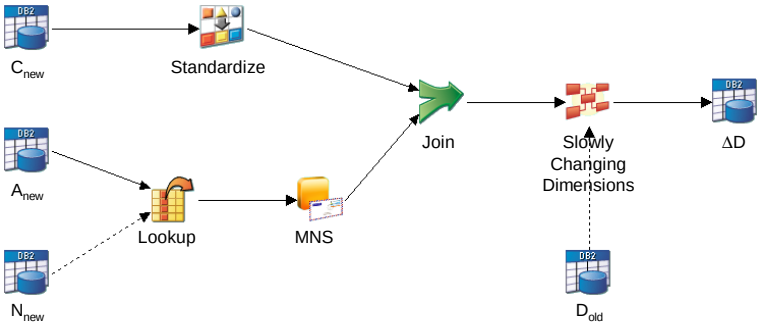


Figure 4.2: ETL job fully recomputing the customer dimension table

The alternative approach to a full recomputations is an incremental recomputation. An incremental ETL job for this purpose can be obtained by algebraic differencing as shown in Chapter 3. The implementation of the incremental ETL job is depicted in Figure 4.3.

$$\Delta\mathcal{D} = [\Pi_{Addr}(A_{new} \bowtie N_{new}) \bowtie (\Pi_{Tel}(\Delta C) \underbrace{-\Pi_{Tel}(C_{old})}_{\text{effectiveness test}})] \cup$$

$$[(\Pi_{Addr}(\Delta A \bowtie N_{new} \cup A_{new} \bowtie \Delta N) \underbrace{-\Pi_{Addr}(A_{old} \bowtie N_{old})}_{\text{effectiveness test}}) \bowtie$$

$$(\Pi_{Tel}(C_{new}))]$$

Note that the incremental expression above contains two so-called effectiveness tests. Effectiveness tests are generated by the delta rules for relational projection, union, and difference to prevent the propagation of redundant deltas. Redundant means that the exact same delta tuple has been computed in an earlier maintenance cycle and thus, already exists in the warehouse. The effectiveness test ensures that each delta tuple in ΔD is truly an insertion, i.e. no such tuple exists in D yet, and that each delta tuple in ∇D is truly a deletion, i.e. the exact same tuple does exist in D. Technically, effectiveness test for insertions (deletions) are realized through relational difference with the minuend being the insert delta tuples (delete delta tuples) and the subtrahend being source tuples from the old database state (the new database state). The difference ensures that those delta tuples that were already derivable from the old database state (are still derivable in the new database state) are eliminated from the set of insertions (deletions).

We did experiments to compare the performance of the naive full recomputation approach and the incremental recomputation approach. We executed

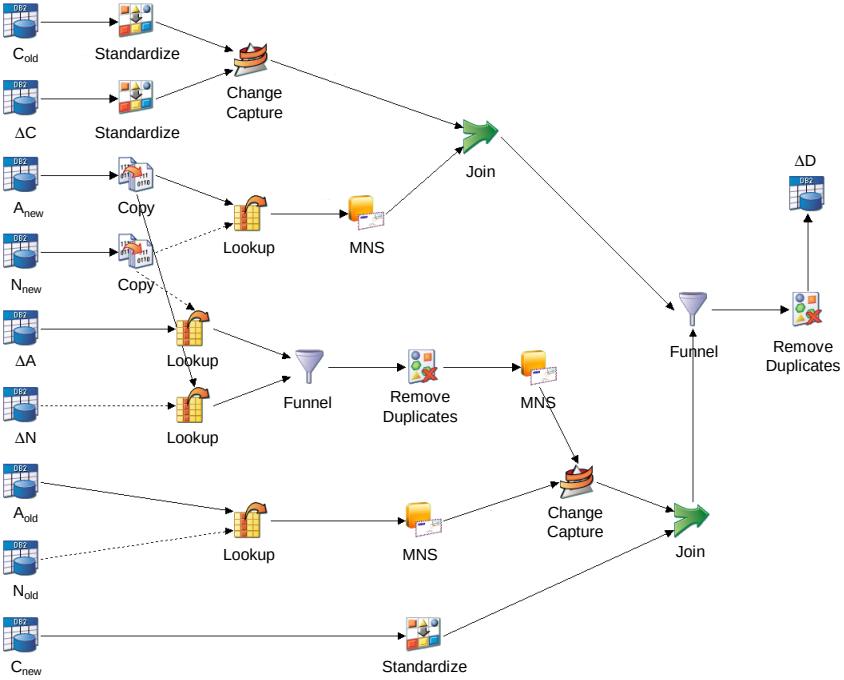


Figure 4.3: ETL job incrementally recomputing the customer dimension table

either variant of the sample ETL job using IBM DataStage as ETL runtime platform. We performed the experiments on an SMP system equipped with two quad-core processor at 2.0 GHz and 8GB RAM. The test data was generated in a way such that the customer relation and the address relation join losslessly and that addresses are uniformly distributed on customers.

The measurements taken are shown in Figure 4.4. The runtime of the different ETL job variants is shown on the y-axis. Note that the measurements are normalized as percent of the initial loading time, i.e. the runtime of the ETL job depicted in Figure 4.1. During the experiments, we stepwise increased the percentage of updates at the source tables, which is denoted on the x-axis.

The time required for the initial computation is of course independent of the number of updates at the sources. The full recomputation works much like the initial computation but additionally compares the recomputed dimension to the current dimension to determine the changes and update the dimension

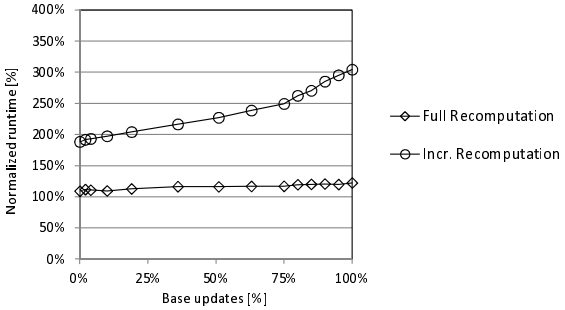


Figure 4.4: Runtime comparison of full recomputation and incremental recomputation ETL jobs

according to the SCD method. Hence, a full recomputation is more expensive than the initial one. The runtime increases slightly for larger numbers of updates at the sources. This is because more and more updates at the warehouse dimension are required, i.e. the size of the output deltas increases.

Surprisingly, the incremental ETL job, which we obtained using the standard delta rules proposed by Griffin et al., performs worse than the full recomputation approach. It takes almost twice as much time for very small numbers of updates at the sources. Unlike the full recomputation that has an almost constant runtime, the runtime of the incremental recomputation increases noticeably with the number of source updates. If the update percentage gets close to a hundred percent, the incremental recomputation takes almost three times as long as the full recomputation. Considering that the whole purpose of incremental recomputation techniques is to improve efficiency, this result seems unexpected. It clearly indicates that the delta rules proposed for the maintenance of materialized views in relational database systems may struggle in the ETL environment. We will explore the reasons in the next section and propose a modified set of delta rules.

4.2 ETL-adapted delta rules

In this section, we will first look into the reasons for the efficiency issues with incremental ETL jobs created by standard delta rules for algebraic differencing and proceed by reworking these rules. The root cause for the poor efficiency of the sample incremental ETL job is that it performs excessive data cleansing. This kind of data transformations does not have a real counterpart in the relational world. Hence, standard delta rules that were proposed for database

view maintenance did not take data cleansing into account. In Chapter 3, we proposed to extend OHM with the TPLTRANS operator that generalizes the PROJECT operator and covers ETL data cleansing operators.

Relational projection and tuple-wise data cleansing have similar operational properties, i.e. tuples are transformed one at a time and a single output tuple is produced per input tuple. Consequently, we treated the TPLTRANS operator just like a relational projection for algebraic differencing, i.e. we used the delta rule for projection in either case. This is justifiable form a correctness point of view, because any incremental ETL job derived this way produces the same result as a naive full recomputation. However, the execution cost of simple projection operators and data cleansing operators are very different and, from a performance point of view, it is problematic to treat them alike.

As discussed in Section 2.1.4, data cleansing often requires sophisticated data parsing and frequent dictionary lookups and is thus a CPU intensive task. In our experiments, address cleansing required about 1 ms of CPU time per tuple and telephone number standardization about 0.2 ms per tuple. Hence, data cleansing is a major cost factor in ETL processing and requires special attention regarding the optimization of incremental ETL jobs. The optimization goal is to reduce the number of cleansing operations done during incremental recomputations. Reconsider the sample incremental ETL job, which is repeated here for the reader's convenience.

$$\begin{aligned}\Delta\mathcal{D} = [&\Pi_{Addr}(A_{new} \bowtie N_{new}) \bowtie (\Pi_{Tel}(\Delta C) \underbrace{-\Pi_{Tel}(C_{old})}_{\text{effectiveness test}})]\cup \\ &[(\Pi_{Addr}(\Delta A \bowtie N_{new} \cup A_{new} \bowtie \Delta N) \underbrace{-\Pi_{Addr}(A_{old} \bowtie N_{old})}_{\text{effectiveness test}}) \bowtie \\ &(\Pi_{Cust}(C_{new}))]\end{aligned}$$

A close look shows that the incremental ETL job cleanses more tuples than necessary. First, source relations and delta relations are not disjoint. Tuples in ΔA are also contained in A_{new}, for instance, and thus each delta tuple is cleansed twice. Second, the effectiveness tests require the (typically large) source relations to be cleansed in their old state, too. Note that the incremental ETL job performs about twice as many cleansing operations as compared to a full recomputation if no updates occurred at the source and about three times as many cleansing operations for a hundred percent of source updates. This is in line with the result of our experiments shown in Figure 4.4. We will address both issues in the following and show that the cleansing effort can be reduced by reworking the delta rules for algebraic differencing.

We will reformulate the delta rules in two ways. Our first objective is to avoid the repeated cleansing of delta tuples. For this purpose, source relations

can be characterized in terms of preserved and modified tuples instead of their old and new state. The preserved portion of a relation R is denoted by $R_o := R_{new} - \Delta R = R_{old} - \nabla R$. As an example, consider the delta rule for relational joins.

$$\Delta(S \bowtie T) \equiv (S_{new} \bowtie \Delta T) \cup (\Delta S \bowtie T_{new})$$
$$\nabla(S \bowtie T) \equiv (S_{old} \bowtie \nabla T) \cup (\nabla S \bowtie T_{old})$$

It is straightforward to rewrite the original delta rules such that any reference to R_{new} and R_{old} is replaced by references to R_o, ΔR, or ∇R. The join rule above is rewritten as follows.

$$\Delta(S \bowtie T) \equiv (S_o \bowtie \Delta T) \cup (\Delta S \bowtie T_o) \cup (\Delta S \bowtie \Delta T)$$
$$\nabla(S \bowtie T) \equiv (S_o \bowtie \nabla T) \cup (\nabla S \bowtie T_o) \cup (\nabla S \bowtie \nabla T)$$

Note that there is no overlap between the preserved, the inserted, and the deleted portion of a relation. Assume that the computation of the (derived) input relation T in the above equation involves data cleansing. The rewritten delta rule ensures that there is no overlap in the input relations of the cleansing operator. In fact, T_o, ΔT, and ∇T are processed in separation and thus delta tuples are no longer cleansed more than once. Note that common subexpressions, such as an expression to compute ΔT for instance, do not need to be evaluated repeatedly. The ETL data flow may simply be forked to pass derived relations to multiple subsequent stages.

The second way in which the delta rules may be improved concerns the effectiveness tests. Recall that effectiveness tests are used in the original delta rules to ensure that each propagated delta is effective w.r.t. the materialized views (the data warehouse) or, in other words, truly leads to a modification of the view. These kind of deltas are also referred to as *true* deltas.

To test the effectiveness of a delta tuple, it needs to be checked whether the same delta tuple has been propagated to the view earlier; if so, it is not effective and can be discarded. Effectiveness tests make this decision by (partly) recomputing the set of tuples that were derivable from the source relations earlier. Obviously, this approach is expensive if data cleansing is part of computing these derivable tuples. We thus argue that, in the ETL environment, it is usually overly expensive to test the effectiveness of deltas along the way while they are being propagated. It seems favorable to check the effectiveness after the propagation process instead, at the time the deltas are installed into the materialized view (the data warehouse). This can be done by a simple lookup on the materialized view. In this way, the work done during populating the view is exploited rather than (partly) redone during the propagation process.

To avoid costly effectiveness tests, we attempt to omit such test in the delta rules whenever possible. In consequence, the incremental jobs generally do not

yield true deltas but a superset thereof. To be useful, this superset must not contain any deltas that would corrupt the materialized view, i.e. it must not contain insertion of tuples that must not appear in the materialized view and it must not contain deletions of tuples that must remain in the view. However, the superset may safely include insertions of tuples that exist in the view already or deletions of tuples that do not currently exist the view. Supersets of true deltas having this property are referred to as *safe* deltas and have been considered in the context of integrity checking [Beh09a].

Until now we identified two root causes for the poor performance of incremental ETL jobs derived by standard delta rules and suggested ways to address these issues. In the following we will present rewritten variants of the original delta rules. These rules characterize relations in terms of preserved and modified portions instead of their old and new state. Furthermore, these rules are designed for the propagation of safe deltas instead of true ones such that expensive effectiveness tests are avoided when possible. However, simply omitting effectiveness tests altogether would lead to wrong results, thus care must be taken in order not to break the rule set.

Projection We first discuss the delta rule for relational projection. In OHM this rule applies to the TPLTRANS operator and all its subtypes such as PROJECT. Consider the relational expression $R := \Pi_A(S)$. Using the original delta rules, this expression is differentiated as follows.

$$\Delta R = \Delta(\Pi_A(S)) \equiv \Pi_A(\Delta S) - \Pi_A(S_o \cup \nabla S)$$
$$\nabla R = \nabla(\Pi_A(S)) \equiv \Pi_A(\nabla S) - \Pi_A(S_o \cup \Delta S)$$

The projection delta rules contain effectiveness tests to prevent redundant deltas from being propagated. A set difference is used to discard ΔR deltas if an alternative derivation of the same tuple existed in R_{old}. Similarly, ∇R deltas are discarded if an alternative derivation continues to exist in R_{new}. To propagate safe instead of true deltas, the rule can be rewritten as follows.

$$\Delta R = \Delta(\Pi_A(S)) \equiv \Pi_A(\Delta S)$$
$$\nabla R = \nabla(\Pi_A(S)) \equiv \Pi_A(\nabla S) - \Pi_A(S_o \cup \Delta S)$$

In contrast to the original rule, the effectiveness test is excluded from the rule for safe insertions. However, it must be retained within the rule for safe deletions. Note that the same R tuple may be derived from multiple S tuples. Simply propagating ∇R deltas without checking for the non-existence of alternative derivations may corrupt the materialized view.

Key-preserving projection The delta rules for relational projection may be further simplified if the input relation contains a primary key column and this column is neither dropped nor transformed by the projection operator. We call a projection operator with this property *key-preserving*. It can be assumed that key-preserving projections are common in ETL jobs, because keys from operational source systems play an important role in dimensional modeling and may be used as business keys in warehouse tables (see 2.5.1).

Say S contains a primary key column with unique values. Because ΔS and S_o are disjoint, the key values found in these sets are pairwise different. Say Π_A is key-preserving. Hence, the key values are pairwise different in $\Pi_A(\Delta R)$ and $\Pi_A(S_o)$, too. Thus, $\Pi_A(\Delta R)$ and $\Pi_A(S_o)$ are also disjoint, and a set difference will never eliminate any tuples, i.e. $\Pi_A(\Delta R) - \Pi_A(S_o) = \Pi_A(\Delta R)$. For the same reasons $\Pi_A(\nabla R)$ and $\Pi_A(S_o)$ must be disjoint. Therefore the original delta rule can be further simplified for key-preserving projections.

$$\Delta R = \Delta(\Pi_A(S)) \equiv \Pi_A(\Delta S) - \Pi_A(\nabla S)$$
$$\nabla R = \nabla(\Pi_A(S)) \equiv \Pi_A(\nabla S) - \Pi_A(\Delta S)$$

Note that the above rules propagate true deltas. They can be further simplified for the propagation of safe deltas.

$$\Delta R = \Delta(\Pi_A(S)) \equiv \Pi_A(\Delta S)$$
$$\nabla R = \nabla(\Pi_A(S)) \equiv \Pi_A(\nabla S)$$

Again, the effectiveness test is omitted in the delta rules for insertions. For key-preserving projections it may additionally be omitted in the delta rule for deletions. This is because an alternative derivation of a deleted tuples cannot exist if the input relation has a unique key column that is preserved.

Union Similar to projection, the relational union implicitly eliminates duplicates and thus alternative derivations of similar tuples must be taken into account. Consider the expression $R = S \cup T$. The following delta rule allows for the propagation of safe deltas.

$$\Delta R = \Delta(S \cup T) \equiv \Delta S \cup \Delta T$$
$$\nabla R = \nabla(\mathcal{S} \cup \mathcal{T}) \equiv ((\nabla S - T_o) - \Delta T) \cup ((\nabla T - S_o) - \Delta S)$$

The first rule is motivated by the fact that every insertion in S and T yields a safe insertion in R. Therefore, an effectiveness test is not needed in this rule. It is, however, required in the rule for safe deletions, because the relational union implicitly eliminates duplicates and tuples deleted in S (or T) may alternatively be derived from the preserved or inserted tuples in T_o and ΔT (S_o and ΔS).

Intersection The remaining relational algebra operators do not eliminate duplicates. Thus, for propagating insertions an effectiveness test is not necessary and the corresponding delta rules cannot be simplified. However, it is easier to propagate safe deletions instead of true ones, because there is no need to check whether the deleted tuple was derivable earlier. Consider the relational expression $R = S \cap T$ that contains an intersection operator. The original delta rules for intersection can be rewritten as follows.

$$\Delta R = \Delta(S \cap T) \equiv (\Delta S \cap T_o) \cup (S_o \cap \Delta T) \cup (\Delta S \cap \Delta T)$$
$$\nabla R = \nabla(S \cap T) \equiv \nabla S \cup \nabla T$$

Note that the first rule has been changed such that the new and old states of S and T are characterized in terms of preserved and modified tuple sets. The second rule was simplified as suggested before; any deletion in S or T is propagated to R. The rational behind this is that the deleted tuple must no longer be in R, no matter if it currently is or not. The deletion can thus safely be propagated in either case.

Difference The delta rules for relational set difference can be simplified in a similar fashion. Consider the relational expression $R = S - T$. The original delta rules for set difference can be rewritten as follows.

$$\Delta R = \Delta(S - T) \equiv ((\Delta S - T_o) - \Delta T) \cup (\nabla T \cap S_o) \cup (\nabla T \cap \Delta S)$$
$$\nabla R = \nabla(S - T) \equiv \nabla S \cup \Delta T$$

Again, the first rule has been rewritten such that S and T are characterized in terms of preserved and modified tuples instead of the new and old relation state. In addition, the second rule was simplified in a similar manner as the intersection rule above. Again, safe deletions in ∇S may be directly propagated without checking whether a derived tuple is found in R. Note that insert deltas ΔT are used to compute safe deletions ∇R because T is negatively referenced in the above expression.

Join In the beginning of this section, it was shown how to rewrite the join delta rules such that the base relations are characterized in terms of preserved and modified tuples instead of their old and new state. The rewritten delta rules are repeated here for the reader's convenience.

$$\Delta R = \Delta(S \bowtie T) \equiv (S_o \bowtie \Delta T) \cup (\Delta S \bowtie T_o) \cup (\Delta S \bowtie \Delta T)$$
$$\nabla R = \nabla(S \bowtie T) \equiv (S_o \bowtie \nabla T) \cup (\nabla S \bowtie T_o) \cup (\nabla S \bowtie \nabla T)$$

In principle, the simplifications made possible by the propagation of safe deletions can be achieved in join delta rules. However, typically each join operand

contributes some attributes to the result schema. Note that this is not the case for relational union, intersection, and difference discussed before. Consequently, join operands cannot simply be omitted. However, in a cascade of join operators the base relations may be restricted to any subset that contributes all required result attributes.

Selection The remaining relational operator to be considered is selection. As it is a unary operator that does not implicitly eliminate duplicates, the delta rules are straightforward.

$$\Delta R = \Delta(\sigma_p(S)) \equiv \sigma_p(\Delta S)$$
$$\nabla R = \nabla(\sigma_p(S)) \equiv \sigma_p(\nabla S)$$

Note that these rules coincide with the original rules for the propagation of true deltas, since these rules cannot be further simplified.

The delta rules adapted to the ETL environment are summarized in Table 4.1. Using these delta rules for differencing the sample ETL job depicted in Figure 4.1 yields the following incremental expression.

$$\begin{aligned}\Delta\mathcal{D} = {} & [\Pi_{Addr}(\Delta A \bowtie N_o \cup A_o \bowtie \Delta N \cup \Delta A \bowtie \Delta N) \bowtie \Pi_{Tel}(C_o)] \cup \\ & [\Pi_{Addr}(A_o \bowtie S_o) \bowtie \Pi_{Tel}(\Delta C)] \cup \\ & [\Pi_{Addr}(\Delta A \bowtie N_o \cup A_o \bowtie \Delta S \cup \Delta A \bowtie \Delta S) \bowtie \Pi_{Tel}(\Delta C)]\end{aligned}$$

Note that the incremental expression above does not contain effectiveness test, i.e. safe deltas are computed, and base relations are characterized in terms of preserved and modified tuple sets. The incremental expression can be deployed into a native ETL job as described in Chapter 3. The deployed ETL job is shown in Figure 4.5. Note that repeated subexpressions in the above incremental expression are not repeatedly evaluated in the corresponding ETL job. Instead, the ETL data flow is forked to pass the result data to multiple subsequent stages.

We compared the runtime of the incremental ETL job with safe delta propagation to the full recomputation approach discussed earlier. The runtime measures are depicted in Figure 4.6. The adapted delta rules lead to an incremental ETL job that performs slightly better than the full recomputation approach. However, it seems surprising that the runtime of the incremental job is nearly constant. Its cost is already high for low update volumes and increases only slightly if the update volumes grow larger. The cost is comparable to the cost of the initial computation, which has been used to normalize the measures in Figure 4.6.

	$\mathcal{E}$	$\Delta(\mathcal{E})$	$\nabla(\mathcal{E})$
1	R	ΔR	∇R
2	$\sigma_p(\mathcal{S})$	$\sigma_p(\Delta(\mathcal{S}))$	$\sigma_p(\nabla(\mathcal{S}))$
3	$\Pi_A(\mathcal{S})$	$\Pi_A(\Delta\mathcal{S})$	$\Pi_A(\nabla\mathcal{S}) - \Pi_A(\mathcal{S}_o \cup \Delta\mathcal{S})$
3'	$\Pi_A(\mathcal{S})$	$\Pi_A(\Delta\mathcal{S})$	$\Pi_A(\nabla\mathcal{S})$ if primary key $\in A$
4	$\mathcal{S} \cup \mathcal{T}$	$\Delta\mathcal{S} \cup \Delta\mathcal{T}$	$((\nabla\mathcal{S} - \mathcal{T}_o) - \Delta\mathcal{T}) \cup$ $((\nabla\mathcal{T} - \mathcal{S}_o) - \Delta\mathcal{S})$
5	$\mathcal{S} \cap \mathcal{T}$	$(\Delta\mathcal{S} \cap \mathcal{T}_o) \cup (\mathcal{S}_o \cap \Delta\mathcal{T}) \cup$ $(\Delta\mathcal{S} \cap \Delta\mathcal{T})$	$\nabla\mathcal{S} \cup \nabla\mathcal{T}$
6	$\mathcal{S} - \mathcal{T}$	$((\Delta\mathcal{S} - \mathcal{T}_o) - \Delta\mathcal{T}) \cup$ $(\nabla\mathcal{T} \cap \mathcal{S}_o) \cup (\nabla\mathcal{T} \cap \Delta\mathcal{S})$	$\nabla\mathcal{S} \cup \Delta\mathcal{T}$
7	$\mathcal{S} \bowtie \mathcal{T}$	$(\mathcal{S}_o \bowtie \Delta\mathcal{T}) \cup (\Delta\mathcal{S} \bowtie \mathcal{T}_o) \cup$ $(\Delta\mathcal{S} \bowtie \Delta\mathcal{T})$	$(\mathcal{S}_o \bowtie \nabla\mathcal{T}) \cup (\nabla\mathcal{S} \bowtie \mathcal{T}_o) \cup$ $(\nabla\mathcal{S} \bowtie \nabla\mathcal{T})$
8	$\mathcal{S} \times \mathcal{T}$	$(\mathcal{S}_o \times \Delta\mathcal{T}) \cup (\Delta\mathcal{S} \times \mathcal{T}_o) \cup$ $(\Delta\mathcal{S} \times \Delta\mathcal{T})$	$(\mathcal{S}_o \times \nabla\mathcal{T}) \cup (\nabla\mathcal{S} \times \mathcal{T}_o) \cup$ $(\nabla\mathcal{S} \times \nabla\mathcal{T})$

Table 4.1: ETL-adapted delta rules

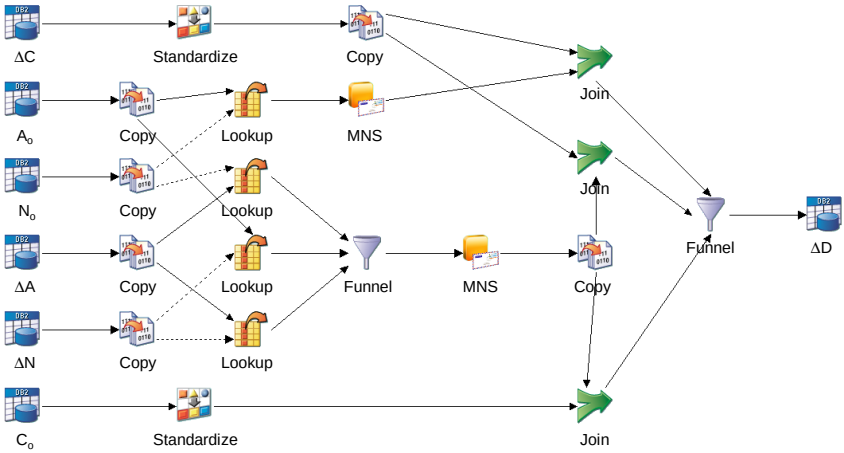


Figure 4.5: ETL job incrementally recomputing the customer dimension table using safe delta propagation

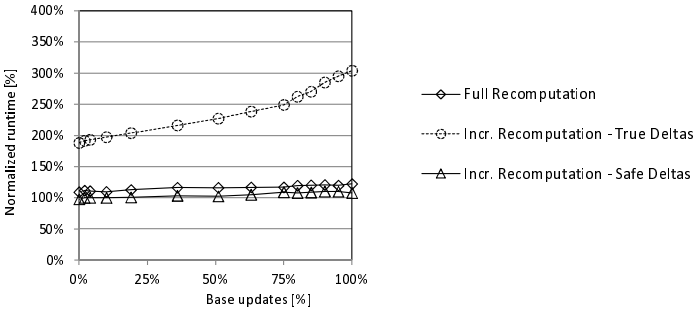


Figure 4.6: Runtime comparison of full recomputation and incremental recomputation ETL jobs using safe delta propagation

This observation can be explained as follows. Data cleansing is typically a costly operation, thus the runtime of an ETL job is heavily influenced by the number of cleansing operations, or in other words, the cardinality of the input relations of the data cleansing operators. The full recomputation job depicted in Figure 4.2 cleanses the customer source relation C_{new} and the joined address relation ($A_{new} \bowtie N_{new}$), each in its current state.

The incremental job to compute safe insertions (shown above) cleanses the preserved and the inserted tuples of both, the customer relation and the joined address relation. The corresponding job to compute safe deletions (not shown above) cleanses the deleted rather than the inserted tuples of the customer relation and the joined address relation. However, for the maintenance of dimension tables it is typically sufficient to compute the business keys of tuples to be deleted. All other attributes are already available at the data warehouse. Thus, a projection dropping all non-key attributes may be appended to the incremental expression. As the projection is pushed down, data cleansing operations that apply to dropped attributes can be eliminated. Note that in the sample ETL scenario, all data cleansing operations are eliminated from the expression to compute safe deletions. That is, the number of cleansing operations is determined by the number of the preserved and inserted tuples of the customer relation and the joined address relation. Thus, the number of cleansing operations required for the initial computation job matches the number of cleansing operations carried out by the incremental recomputation job. For this reason both jobs show comparable performance.

4.3 Projection pushing

The incremental ETL job may be further improved by reducing the number of cleansing operations. One obvious way to achieve this goal would be to store pre-cleansed copies of base data in the staging area of the data warehouse. However, this approach incurs a dramatic storage overhead and may lead to synchronization issues.

An alternative approach is to reduce the size of the input relations of data cleansing operators by algebraic optimization techniques. Projection operators are typically pushed down in a query plan to eliminate irrelevant columns as early as possible thereby reducing the size of the intermediary results. Data cleansing operators, though similar from a modeling perspective, typically do not reduce the size of intermediary result and are computationally expensive. It thus seems reasonable to push cleansing operators upwards to apply them as late as possible, after the in-stream data has been reduced by prior selection and join predicates. Proceeding this way, the incremental expression for safe delta propagation can be rewritten as follows.

$$\begin{aligned}\Delta\mathcal{D} = \Pi_{Addr}(\Pi_{Tel}((&\Delta A \bowtie N_o \cup A_o \bowtie \Delta N \cup \Delta A \bowtie \Delta N) \bowtie C_o \cup \\ &(A_o \bowtie N_o \bowtie \Delta C) \cup \\ &(\Delta A \bowtie N_o \cup A_o \bowtie \Delta N \cup \Delta A \bowtie \Delta N) \bowtie \Delta C))\end{aligned}$$

Note that similar cleansing operators have been merged while being pushed upwards. When such an operator is pushed through a join its input schema may include additional columns hereafter. In our running example the address standardization is now applied after the join of the address and the customer relation, for instance. We assume that the input and output schemas of the cleansing operators are implicitly adjusted and that additional columns are simply passed through.

We did experiments to compare the performance of the incremental recomputation job rewritten using projection pushing with its non-rewritten counterpart. The runtime measures are depicted in Figure 4.7. Note that we performed experiments for different ratios of the size of the customer table to the size of the address table. We chose ratios of 1:1, 1:2, and 1:10; the respective measurements are depicted in the plots a, b, and c in Figure 4.7.

The performance behavior of the rewritten job reflects the volume of source updates more directly and generally outperforms the non-rewritten variant for small volumes of updates. Interestingly, the performance of the rewritten job is strongly influenced by the size ratio of the base relations. Projection pushing provides the biggest advantage if the cardinality of the customer relations matches the cardinality of the address relation (Figure 4.7.a). The performance advantage degrades if multiple customer tuples join to an address tuple

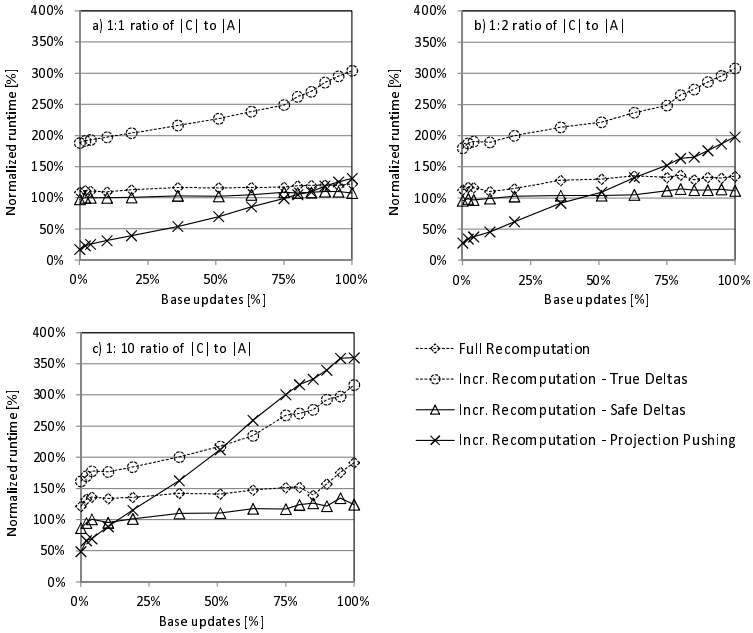


Figure 4.7: Runtime comparison of two incremental recomputation ETL jobs

(Figure 4.7.b and 4.7.c). Note that the non-rewritten incremental job, in contrast, is not influenced by the size ratio of the base relations.

The above observation can be explained as follows. The non-rewritten job cleanses the preserved and modified base tuples before the join is evaluated. The number of preserved tuples is typically far bigger than the number of updated tuples. Hence, many preserved tuples do not join to an updated one. That is, the non-rewritten ETL job puts effort into cleansing tuples that do not survive the join and are thus irrelevant for the overall computation.

The rewritten ETL job avoids this overhead. It cleanses preserved base tuples only if they satisfy the join predicate, i.e. if they appear in the join result. That way, cleansing irrelevant tuples is avoided and the rewritten job shows good performance for small numbers of base updates. However, a tuple may have multiple join partners and may therefore appear multiple times in the join result. In consequence, such tuples are cleansed repeatedly. The resulting overhead explains why the projection pushing rewriting technique is not always advantageous. The rewritten job shows its best performance if customer and

address tuples correspond to each other one-to-one (Figure 4.7.a). In this case individual tuples are obviously not cleansed more than once. However, for size ratios other than that the overhead for repeated cleansing impacts the performance.

In summary, both incremental recomputation approaches have their own specific overhead. The first variant we considered, may cleanse irrelevant tuples, while the second variant may cleanse relevant tuples repeatedly. Which variant is preferable, depends on the distribution and the update volume of the base data. Thus, there is no clear winner. In the next section, we propose a more robust way to reduce the cleansing overhead using an algebraic rewriting technique referred to as Magic Sets.

4.4 Magic ETL optimization

The Magic Sets technique is a query rewriting technique that was originally proposed for the optimization of recursive queries in deductive database systems. We will discuss the principles of the Magic Sets rewriting technique and its application to non-recursive queries in Sections 4.4.1 and 4.4.2, respectively. Apart from its traditional application areas, we found the Magic Sets technique to be useful for the optimization of incremental ETL jobs, as will be discussed in Section 4.4.3.

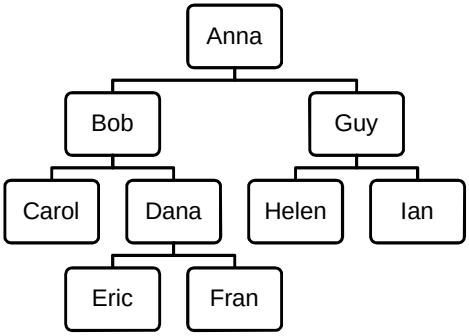
4.4.1 Principles of magic sets

The Magic Sets algorithm has been proposed for the efficient evaluation of recursive queries in the context of deductive databases [BMSU86, Ull89]. Deductive databases typically use a query language called Datalog, which is inspired by the logic programming language Prolog. As compared to SQL, recursive queries[2] can more naturally be expressed in Datalog as illustrated by the following sample query.

$$sl(X, X).$$
$$sl(X, Y) :- mngr(X, X1), sl(X1, Y1), mngr(Y, Y1).$$

The sample Datalog query consists of two rules. Datalog rules can be divided into a head and a body placed on the left-hand side and the right-hand side of the ":–" symbol, respectively. Intuitively, a rule is interpreted as follows: If the tuples mentioned in the body exist in the database, the tuples mentioned in the head must also be in the database, i.e. the body of a rule implies its head. If the body is empty, as is the case in the first sample rule, the head

[2] Support for recursive queries has been added in the SQL:1999 standard.



mngr

X	Y
Bob	Anna
Guy	Anna
Carol	Bob
Dana	Bob
Helen	Guy
Ian	Guy
Eric	Dana
Fran	Dana

Figure 4.8: Sample management hierarchy

always evaluates to true, i.e. the tuples mentioned in the head must be in the database.

The intention of the sample Datalog query is to find all employees in a management hierarchy that are at the same level as a given employee. The query refers to two relations, namely *mngr* and *sl*. It is assumed that the former relation is stored in the database and that $mngr(X, Y)$ means that X is managed by Y as shown in Figure 4.8. The latter relation *sl* is inferred from *mngr* using the above rules.

The first rule says that any employee is on the same level with himself. The second rule says that any two employees X and Y are at the same level, if their respective managers $X1$ and $Y1$ are at the same level. In Datalog, a query to find all employees at the same level as "Carol" is formulated as $sl(Carol, X)$. There are two obvious ways to evaluate such a query.

First, the query may be evaluated top-down. This strategy is referred to as backward-chaining and implemented in Prolog. Starting from "Carol", her manager is visited first. Then, recursively, all managers at the same level are considered to find all employees managed by them. During the recursive processes, the same employee may be visited more than once through different paths in the hierarchy. The flaw with the backward-chaining approach is that it discovers "all derivation paths" rather than "all answers".

Second, the query may be evaluated bottom-up. This strategy is referred to as forward-chaining. Initially, only the tuples in the database (i.e. in the *mngr* relation) are considered and any query parameters (i.e. "Carol") are ignored. The idea of forward-chaining is to fully build the *sl* relation and select the tuples

matching the query hereafter. The *sl* relation is initially empty and build step-by-step in several passes over the query rules. On the first pass, each employee is found to be at the same level with himself. On the second pass, any two employees with a common manager are found to be at the same level. On the third pass, any two employees having managers with a common manager are found to be at the same level, and so on. After the process terminates, the *sl* relation is restricted to those tuples having "Carol" as their first component to answer the query. Note that many tuples generated by forward-chaining may not contribute to the query result.

Whether backward-chaining or forward-chaining is more efficient depends entirely on the given base data. However, neither method is very good in general, because backward-chaining may repeatedly compute the same answer tuple and forward-chaining may compute tuples that do not contribute to the answer in any way. The Magic Sets method has been proposed as a more efficient alternative [BMSU86, Ull89].

The Magic Sets method evaluates queries bottom-up. In contrast to forward-chaining, however, the computation of irrelevant tuples that do not contribute to the answer to the query is generally avoided. The Magic Sets method can be thought of as a generalized selection pushing technique. It allows for pushing the selection, which is the final step of forward-chaining, into recursive queries. For this purpose, the query rules are rewritten and so-called Magic Sets are added that act as filters on the set of tuples generated by each rule.

Reconsider the sample Datalog query. We are interested in finding all tuples in *sl* with the first field equal to "Carol". However, to find all such tuples, other tuples in *sl* have to be computed as well. For example, the tuple $sl(Carol, Ian)$, which is part of the query answer, cannot be computed unless the tuple $sl(Bob, Guy)$ has been computed, which is not part of the query answer. Thus, we cannot simply restrict the query evaluation to those tuples that directly contribute to the answer but need to consider those tuples that indirectly contribute to the answer, too. Intuitively, these are tuples whose first field contains a value on the path from "Carol" to the top of the hierarchy such as $sl(Bob, Guy)$, for instance. These tuples are computed by the following Datalog rules and referred to as a Magic Set.

$$magic(Carol).$$
$$magic(U) :\!- magic(V), mngr(V, U).$$

Any tuple outside the Magic Set, such as $sl(Dana, Ian)$ for instance, cannot be used to compute tuples that directly or indirectly contribute to the query answer. Computing such tuples is thus wasteful and should be avoided. To do so, the Magic Set can be used in the original rules to prevent the computation

of irrelevant tuples. The Magic Set-rewritten rules look like this.

$$sl(X, X) :- magic(X).$$
$$sl(X, Y) :- magic(X), mngr(X, X1), sl(X1, Y1), mngr(Y, Y1).$$

It can be shown that the rewritten query yields the same result as the original one, but may work more efficiently. The Magic Sets rewriting can be done automatically; the rewrite algorithm is presented in [BMSU86, Ull89].

4.4.2 Magic sets for non-recursive queries

The Magic Sets method was originally intended for the optimization of recursive queries. However, it has been reported to be effective in the optimization of correlated nested queries, which are not recursive, too [MFPR90, MP94].

As an example, consider the SQL query shown in Figure 4.9.a that selects programmers who make more than the average salary in their department. In the query, a correlated subquery is used to compute the average salary for each programmer. No irrelevant information is computed, however, if there are multiple programmers in the same department, the average salary of the department is computed repeatedly. Note that a nested loop join is used in the query plan[3] shown in Figure 4.9.a.

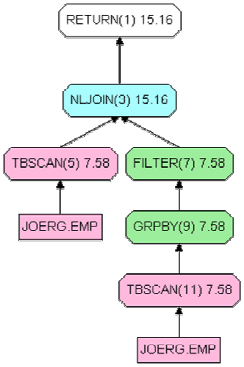
The repeated computation of the average salary in a single department can be avoided by decorrelating the sample query as shown in Figure 4.9.b. The decorrelated query pre-computes the average salary for each department and stores the result in a temporary table which is joined to the employees table. Note that a hash join is used in the query plan. In contrast to the correlated query, its decorrelated counterpart is processed set-oriented. This seems like an advantage, however, the decorrelated query computes the average salary for all departments including those that do not have any programmers and are thus irrelevant for answering the query. In consequence there is no clear winner among correlated and decorrelated queries.

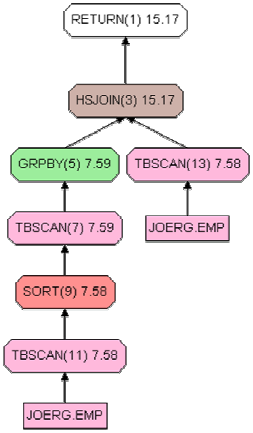
Clearly there is an analogy between backward-chaining and forward-chaining in recursive queries, and correlated and decorrelated evaluation of subqueries. The former methods produce only relevant tuples but may produce the same tuple repeatedly. The latter methods produce each tuple only once but may produce irrelevant tuples. Again, the Magic Sets method can be used to combine the advantages of either approach, at the cost of a somewhat more complex query plan. The Magic Sets-rewritten query and query plan is depicted in Figure 4.9.c. The Magic Set is computed as the set of departments that have at

[3] Even though SQL is a declarative language we found the query plans generated by IBM DB2 to mimic the structure of the SQL query with regard to correlation.

a)

```
SELECT name FROM emp e1
WHERE job = 'Programmer' AND sal >
          (SELECT  AVG(sal) FROM emp e2
           WHERE e2.did = e1.did)
```





b)

```
WITH deptavgsal AS (
          SELECT did, AVG(sal) AS avgsal
          FROM emp GROUP BY did)

SELECT name FROM emp e, deptavgsal d
WHERE job = 'Programmer'
AND sal > avgsal AND e.did = d.did
```

c)

```
WITH smag AS (
          SELECT name, did, sal FROM emp
          WHERE job = 'Programmer'),

mag AS (
          SELECT DISTINCT did FROM smag),

magavgsal  AS (
          SELECT mag.did, AVG(sal) AS avgsal
          FROM mag, emp
          WHERE mag.did = emp.did
          GROUP BY mag.did)

SELECT name FROM smag s, magavgsal m
WHERE sal > avgsal AND s.did = m.did
```

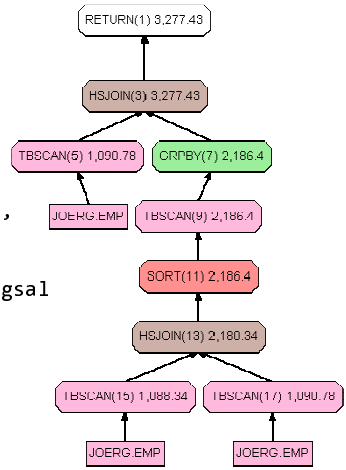


Figure 4.9: Sample correlated, decorrelated, and magic sets-rewriteen queries

least one programmer and used to restrict the average salary calculation to those relevant departments.

The Magic Sets method generalizes selection pushing in the sense that selections can be pushed sideways in the query plan from one branch into another. In [BM04, Beh09b] it was shown that the Magic Sets method is also effective in the optimization of view maintenance expressions. It is proposed to use the delta sets as Magic Sets and push them sideways to prevent the computation of irrelevant tuples in other branches of the query plan, i.e. tuples that do not match any delta tuple in a later operation such as join, difference, or intersections.

4.4.3 Magic Sets for incremental ETL jobs

In the preceding section, we drew an analogy between the bottom-up (forward-chaining) and top-down (backward-chaining) evaluation of recursive queries, and the decorrelated and correlated evaluation of non-recursive queries. In the former case irrelevant intermediate data may be computed; in the latter case relevant intermediate data may be computed repeatedly. The same analogy can be drawn to the incremental ETL job variants discussed in Section 4.2 and 4.3 before. Recall that the first incremental variant we considered may cleanse irrelevant base tuples. The second variant, which was obtained by pushing the cleansing operators upwards, may cleanse base tuples repeatedly. In this section, we will show that the Magic Sets technique is useful in the optimization of incremental ETL jobs that involve data cleansing.

As said, Magic Sets is essentially a generalized selection pushing technique, which allows for selections to be pushed sideways in a query plan. Reconsider the first variant of the sample incremental ETL job proposed in Section 4.2, which is repeated here for the reader's convenience.

$$\begin{aligned}\Delta\mathcal{D} = {} & [\Pi_{Addr}(\Delta A \bowtie N_o \cup A_o \bowtie \Delta N \cup \Delta A \bowtie \Delta N) \bowtie \Pi_{Tel}(C_o)] \cup \\ & [\Pi_{Addr}(A_o \bowtie S_o) \bowtie \Pi_{Tel}(\Delta C)] \cup \\ & [\Pi_{Addr}(\Delta A \bowtie N_o \cup A_o \bowtie \Delta S \cup \Delta A \bowtie \Delta S) \bowtie \Pi_{Tel}(\Delta C)]\end{aligned}$$

It contains two joins between delta relations and cleansed base relations, i.e. $\Pi_{Tel}(C_o)$ and $\Pi_{Tel}(A_o \bowtie S_o)$. The (implicit) join condition can be understood as a selection condition, which discards any base tuple for which no corresponding delta tuple exists. That is, the delta sets act as selection constants themselves. Following the Magic Sets approach, the selections may be pushed sideways in the ETL job, such that irrelevant base tuples are eliminated early. The Magic Sets are defined by the (derived) delta sets restricted to the join columns. In the sample job the join column is the primary key column of

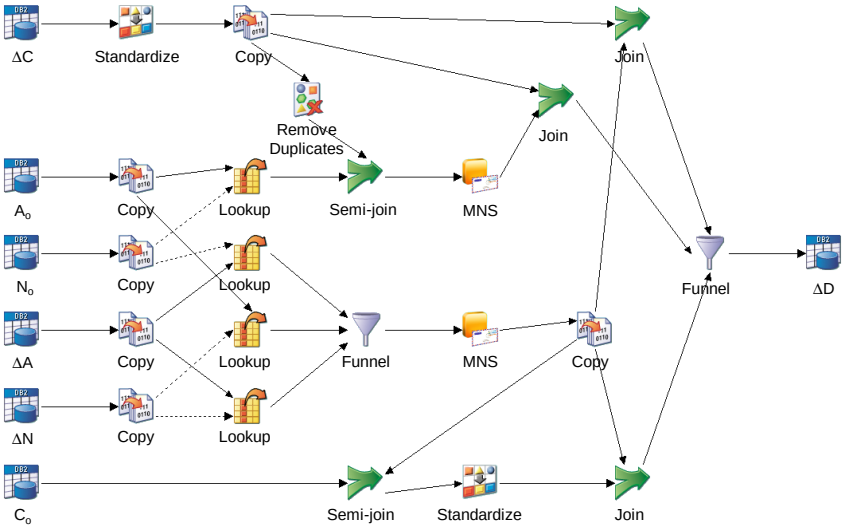


Figure 4.10: Magic Sets-rewritten ETL job incrementally recomputing the customer dimension table

the relation A, which is denoted as a below.

$$magicC = \pi_a(\Pi_{Addr}(\Delta A \bowtie N_o \cup A_o \bowtie \Delta N \cup \Delta A \bowtie \Delta N))$$
$$magicA = \pi_a(\Pi_{Tel}(\Delta C))$$

As said, magic set definitions can be derived automatically. An algorithm to derive magic set definitions from incremental relational algebra expressions is discussed in [Beh09b]. The Magic Sets provide selection constants that are dynamically determined at runtime and applied to an adjacent evaluation branch using semi-joins, denoted as $\ltimes$ in the expression below.

$$\begin{aligned}\Delta\mathcal{D} = &[\Pi_{Addr}(\Delta A \bowtie N_o \cup A_o \bowtie \Delta N \cup \Delta A \bowtie \Delta N) \bowtie \Pi_{Tel}(C_o \ltimes magicC)] \cup \\ &[\Pi_{Addr}(A_o \ltimes magicA \bowtie S_o) \bowtie \Pi_{Tel}(\Delta C)] \cup \\ &[\Pi_{Addr}(\Delta A \bowtie N_o \cup A_o \bowtie \Delta S \cup \Delta A \bowtie \Delta S) \bowtie \Pi_{Tel}(\Delta C)]\end{aligned}$$

The deployed ETL job is shown in Figure 4.10. Again, the ETL flow is forked several times to avoid the repeated computation of repeated subexpressions. Note that the magic set semi-joins eliminate irrelevant preserved base tuples early in the ETL flow. This way, the intermediate data fed into the data

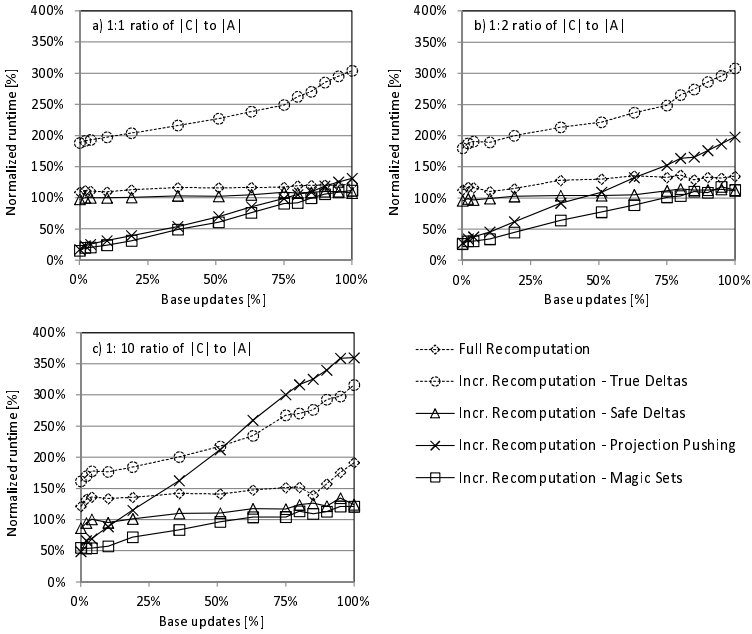


Figure 4.11: Runtime comparison of the Magic Sets-rewritten ETL job

cleansing operators is reduced. In general, Magic Sets-rewritten incremental jobs combine the advantages of both incremental variants discussed before. Similar to the first variant we considered, Magic Sets-rewritten jobs do not cleanse any tuple more than once. Similar to the second variant, Magic Sets-rewritten jobs do not cleanse base tuples that are irrelevant for the overall incremental recomputation.

Figure 4.11 shows runtime measurements for the incremental job variants discussed in this chapter. It is interesting to see that even though the Magic Sets rewriting adds some complexity, the rewritten job becomes more expensive than the initial computation job only if the update volume exceeds about 60%. Thus, the computational overhead introduced remains relatively small. In our experiments, the Magic Sets-rewritten variant generally performed best. It outperforms the incremental variant computing safe deltas, in particular for small volumes of updates, because the overhead of cleansing irrelevant tuples is avoided. It furthermore outperforms the incremental variant obtained by pushing cleansing operators upwards, in particular for uneven size ratios of the

base relations, because the overhead of cleansing tuples repeatedly is avoided. For an even size ratio the Magic Sets-rewritten job performs comparably good.

4.5 Chapter summary

In the previous chapter, we proposed an abstract ETL operator model that borrows from the relational algebra. That way, the algebraic differencing technique, which was originally developed in the context of incremental view maintenance, can be applied to derive incremental ETL jobs. However, as we have shown in this chapter, the original delta rules are inadequate for this purpose and the resulting incremental ETL jobs suffer from performance problems. This is due to the fact that ETL computations usually involve data cleansing operations that are computationally expensive. These operations are quite ETL-specific and do not have a counterpart in the relational algebra. Hence, data cleansing operations have not been considered in the context of incremental view maintenance before.

In the beginning of this chapter, we derived an incremental ETL job using the delta rules originally proposed for algebraic differencing and demonstrated its performance shortcomings. We identified the root causes of the poor performance and, in the following, step-wise adapted the delta rules to resolve the identified issues. In doing so, we were able to use ideas and techniques developed in database research, though in entirely different contexts.

We leveraged the concept of safe delta propagation that was originally proposed for efficient integrity checking. Safe deltas are essentially a superset of true deltas and may be computed more efficiently. We rephrased the delta rules to compute safe deltas instead of true ones. In doing so, we were able to reduce the computational overhead incurred by so-called effectiveness test in combination with data cleansing operators.

To further improve incremental ETL jobs we proposed a technique from yet another context referred to as Magic Sets. The Magic Sets rewriting technique was originally developed for the efficient evaluation of recursive queries in deductive databases and has more recently been considered for optimizing non-recursive queries with correlated subqueries. As we have shown, the Magic Sets technique is furthermore effective in reducing the cleansing overhead in incremental ETL recomputations.

The Magic Sets technique has sometimes been criticized for being unstable. Its inventors note that “sometimes, our algorithm decides that no transformation is desirable, and sometimes it makes a transformation to rules that run more slowly on the given data than the originals” [BMSU86]. The optimization effect of Magic Sets is not easily predictable, because the rewritten queries tend to be more complex than the originals. Whether Magic Sets rewriting pays off

depends on the nature of the input data.

In the context of incremental ETL recomputations, we found the Magic Sets optimization to deliver more robust results. Note that the Magic Sets technique is used in traditional query optimization with the goal of reducing the I/O cost. In the ETL context, however, we used Magic Sets primarily to reduce the CPU cost caused by data cleansing operations. We found this optimization effect to reliably outweigh the increased complexity of the Magic Sets-rewritten jobs. We believe that a similar technique could be used to optimize database queries that contain computationally expensive user-defined functions, but we are not aware of any work in this direction.

5 Incremental view maintenance using partial deltas

A prerequisite for recomputing derived data incrementally is the ability to capture changes to the underlying source data. In the warehousing context, techniques that gather change data (or deltas) at source systems are referred to as Change Data Capture (CDC) techniques. Several different CDC approaches are used in practice; we will present a survey in Section 5.1. As we will see, CDC approaches differ not only in terms of their implementation but, more importantly, capture deltas at different levels of completeness. We refer to deltas that are incomplete in some sense as *partial* deltas. In Section 5.2, we will proceed with a discussion of techniques to apply deltas to data warehouse tables, which we refer to as Change Data Application (CDA) techniques. In particular, we will discuss the application of partial deltas.

Partial input deltas have interesting implications for incremental view maintenance that will be explored in this chapter, which summarizes our previous work in [JD08, JD09a, JD11]. All view maintenance techniques we are aware of require input deltas to be complete. Even approaches to distributed view maintenance discussed in Section 2.3.4 are no exception. However, in an ETL environment, the source systems are autonomous and independently choose the interfaces to be offered for data access by external systems. This includes an autonomous decision for a CDC technique. In consequence, the source systems in a data integration scenario, may not be able or willing to provide non-partial deltas to the ETL system. Tolerating the autonomy of source systems is a major strength of ETL tools and key to data integration, because it is infeasible to adapt existing sources to the needs of the data integration system.

In this chapter, we will explore incremental maintenance of warehouse tables using partial deltas. Warehouse tables are, from a conceptual perspective, much like materialized views hosted at a remote system. The techniques proposed in this chapter are applicable to both environments. We will therefore speak of materialized views and view definitions to mean warehouse tables and ETL job definitions, respectively, throughout this chapter.

In Section 5.3, we will discuss the impact of partial input deltas on algebraic view maintenance. We will show that incremental expressions derived by standard delta rules may produce (partly) incorrect results if the source deltas are partial. In consequence, the materialized view may be corrupted during

incremental maintenance. We will identify different kinds of anomalies that may occur during view maintenance using partial source deltas and look into the reasons for these anomalies to happen. To cope with this problem, we will propose a generalized approach to algebraic view maintenance that deals with partial source deltas and avoids any anomalies in Section 5.4.

5.1 Change data capture

The term Change Data Capture (CDC), which is mainly used in the data warehousing context, is an umbrella term for techniques that gather change information (or deltas) at source systems. Capturing deltas at the sources is a prerequisite for incremental warehouse maintenance. We analyzed existing CDC techniques and identified four main approaches, namely utilization of audit columns, log-based CDC, change tracking, and the computation of snapshot differentials, which will be discussed in the sequel.

Audit columns: Source tables may include dedicated columns to store a timestamp or a version number for each tuple. Such columns are usually referred to as audit columns [KC04, LHM$^+$86]. Consider the sample customer table with three sample modification (an insertion, an update, and a deletion) shown at the top of Figure 5.1. Below this in Figure 5.1.a, the sample customer table is depicted with an additional audit column. Whenever a tuple is changed, its audit column is assigned a fresh timestamp. Audit columns can serve as selection criteria to retrieve tuples that have been updated since a given point in time.

IBM DB2 for z/OS offers native support for audit columns [DB2b]. Columns may be declared as ON UPDATE AS ROW CHANGE TIMESTAMP in CREATE TABLE statements. DB2 generates a value for the change timestamp column whenever a new tuple is inserted or any column of an existing tuple is updated. The value that is generated is a timestamp that corresponds to the insert or update time of the tuple. Tuples can also be filtered based on the time that they were updated or inserted using the change timestamp column. For that purpose, a predicate with the keyword ROW CHANGE TIMESTAMP can be specified in the WHERE clause of SQL queries.

When a single audit column is used, insertions cannot be distinguished from updates when deltas are extracted. Furthermore, deletions remain undetected when tuples are physically deleted at the source. To work around these limitations, additional audit columns can be appended to base tables [Ina10]. To distinguish insertions from updates, separate audit columns can be used for each type of modification. That is, a dedicated column is used to store the time of the insertion while another column is used to store the time of the last

Customer (before the modifications)

ID	Name	Discount
1	Adam	0%
2	Bob	0%
3	Carl	0%

```
INSERT INTO Customer
VALUES (4, 'Dave', 0.00)

UPDATE Customer
SET discount = 0.05
WHERE id = 1

DELETE FROM Customer
WHERE id = 3
```

a) Audit columns

Customer (before the modifications)

ID	Name	Discount	Update Time
1	Adam	0%	10
2	Bob	0%	20
3	Carl	0%	10

Customer (after the modifications)

ID	Name	Discount	Update Time
1	Adam	**5%**	**31**
2	Bob	0%	20
4	Dave	0%	**30**

b) Audit columns (with logical deletions)

Customer (before the modifications)

ID	Name	Discount	Update Time	Delete Time
1	Adam	0%	10	-
2	Bob	0%	20	-
3	Carl	0%	10	**-**

Customer (after the modifications)

ID	Name	Discount	Update Time	Delete Time
1	Adam	**5%**	**31**	-
2	Bob	0%	20	-
3	Carl	0%	10	**32**
4	Dave	0%	**30**	-

c) Log-based Change Data Capture

Customer (after the modifications)

ID	Name	Discount
1	Adam	5%
2	Bob	0%
4	Dave	0%

Customer Log (after the modifications)

Operation	ID	Name	Discount
I	4	Dave	0%
UN	1	Adam	5%
UO	1	Adam	0%
D	3	Carl	0%

Figure 5.1: Change data capture examples

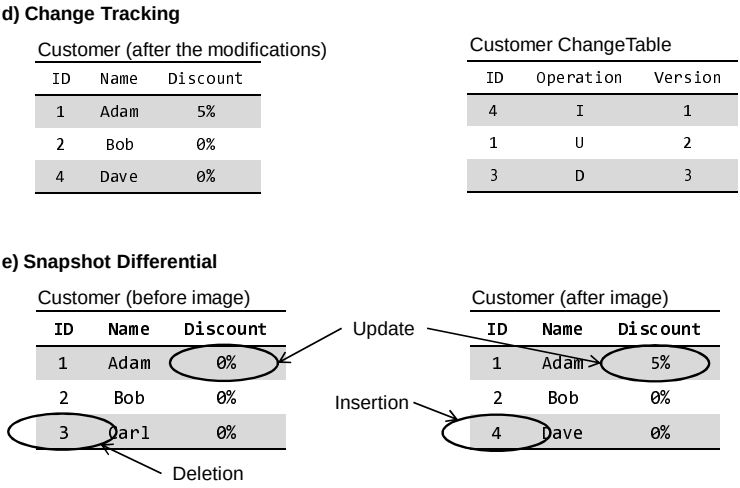


Figure 5.1: Change data capture examples (continued)

update. To capture deletions, tuples must not be physically deleted but rather logically. This can be done by adding yet another audit column to store the time of the (logical) deletion as indicated in Figure 5.1.b.

However, CDC techniques backed by audit columns are generally unable to capture the initial state of updated tuples. This is obvious considering that updates are performed in-place and previous values are overwritten.

Log-based CDC: Source systems may keep a log of changes as shown in Figure 5.1.c. A log entry is appended whenever the base table is modified. Each log entry includes an operation code that indicates the type of the modification. In the example, 'I' is used to denote an insertion and 'D' is used to denote a deletion. For an update both, the original state of the updated tuple (UO) and the new state of the updated tuple (UN) may be written to the log.

Several implementation approaches for log-based CDC exist: If the source system provides active database capabilities such as triggers, deltas can be written to dedicated log tables. Log-based CDC can also be implemented by means of application logic. In this case, the application program that updates the base data is responsible for writing the deltas to the log tables. Database log scraping is another widely used CDC approach. The idea is to exploit the transaction logs kept by the database system for backup and recovery. Deltas can be extracted from transaction logs using database-specific utilities.

Log-based CDC mechanisms are generally capable of providing complete deltas. However, their efficiency can be improved if somewhat incomplete (or partial) deltas are acceptable. For algebraic view maintenance the so-called net effect of changes is required as input, i.e. deltas must be provided as sets rather than bordered multisets as in the log. When a tuple has been changed multiple times, the effects of these changes are combined to produce a single delta tuple. If a tuple has been inserted and subsequently updated, for instance, a delta tuple of type insertion with the updated values is produced. To obtain the net effect, the change log needs to be post-processed.

The net-effect computation is more efficient if partial output deltas are acceptable. The following quote has been taken from the SQL Server 2008 documentation on the change capture feature [SQL].

> Because the logic to determine the precise operation for a given change adds to query complexity, this option is designed to improve query performance when it is sufficient to indicate that [...] the change is either an insert or an update, but it is not necessary to explicitly distinguish between the two.

Note that not being able to distinguish between insertions and updates means that the initial state of updated tuples is not available either.

Change tracking: Change Tracking is an alternative change capture feature of SQL Server 2008 built into the database engine [SQL]. Change tracking is being advertised as light-weight change capture solution that offers better scalability than audit column or trigger-based solutions.

Change tacking is done by making a note of the primary key of the tuple that changed, along with the type of the change (insert, update, or delete) and a version number in an internal table as shown in Figure 5.1.d. To retrieve deltas, the change tracking table needs to be joined to the corresponding base table, because it does not store any non-key attributes. More precisely, an outer join needs to be used, because deleted tuples are no longer found in the base tables. Thus, deltas produced by change tracking do not contain any information about deleted tuples except for the primary keys. Furthermore, the initial state of updated tuples cannot be reconstructed, because it has been overwritten in the base table.

Snapshot differentials: The snapshot differential technique is typically used to capture deltas at unsophisticated sources. Its main advantage is that source systems do not need to be altered in any way. Legacy systems or unsophisticated sources such as file systems are usually not easily extensible and lack a general purpose query interface. Hence, the CDC techniques discussed so

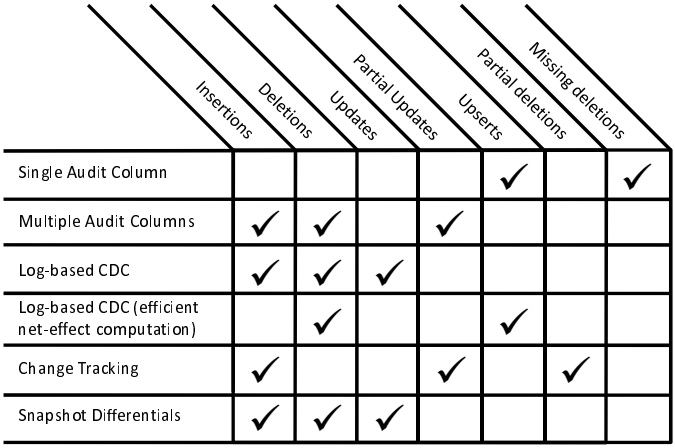


Figure 5.2: Types of deltas captured by different CDC techniques

far cannot easily be implemented. However, virtually all source systems allow for dumping a complete data snapshot into the file system. Computing a snapshot differential means to infer deltas by comparing a current source snapshot (referred to as after image) with an earlier one (referred to as before image) [LGM96]. It is assumed that each tuple can be uniquely identified by its primary key. All tuples that appear in the after image but not in the before image have been inserted. Similarly, tuples that appear in the before image but not in the after image have been deleted. Tuples that appear in both snapshots but differ from each other have been updated. An example is shown in Figure 5.1.e.

Computing snapshot differentials clearly does not scale well [LGM96]. As the volume of source data grows, more and more data has to be extracted and larger and larger comparisons have to be performed. In fact, the technique is usually seen as "a last resort" for capturing deltas at unsophisticated data sources. The snapshot differentials approach is capable of capturing complete deltas. Interestingly, existing implementations such as the snapshot differential operator provided with IBM InfoSphere DataStage [Inf] do not compute the initial state of updated tuples. We can only speculate as to the reasons. Possibly, it was developed with a one-to-one correspondence between the source and target dataset in mind. In such a setup, knowledge of the current state of updated tuples is sufficient for the target dataset to be maintained.

In the remainder of this section, we will give an intuitive notion of partial deltas and establish some basic terminology. A formal model for partial deltas will be presented in Section 5.4.1. Our survey of CDC techniques has shown that deltas are captured at different levels of completeness by different CDC techniques. Intuitively, three types of partial deltas can be distinguished.

First, we will refer to a delta tuple as *partial update* if it captures the current state of a tuple that has been updated but does not capture its initial state. Partial updates are produced by the audit columns CDC technique, change tracking, and log-based approaches with an efficient net-effect computation.

Second, we will refer to a delta tuple as *partial deletion* if it does not completely capture the attribute values of a tuple that has been deleted. Partial deletions are produced by the change tracking technique, because it is incapable of capturing any attribute values except for the primary key. Note that the audit column technique in its most simple form cannot detect deletions at all. We will refer to this situation as *missing deletions*.

Third, we will refer to a delta tuple as *upsert* if the corresponding modification has either been an insertion or an update but these cases cannot be distinguished. Upserts are produced when changes are captured using a single audit column or a log-based approach with an efficient net-effect computation. The types of (partial) deltas captured by different CDC techniques are summarized in Figure 5.2.

5.2 Change data application

Algebraic view maintenance is done by differencing view definition to obtain incremental expressions that propagate deltas from the sources to the materialized view. The deltas computed by the incremental expressions are applied to the materialized view to re-synchronize it with the underlying base data. In analogy to the term Change Data Capture that refers to techniques to capture deltas at the sources, we will use the term Change Data Application (CDA) to mean techniques that apply deltas to the materialized view.

This chapter discusses incremental view maintenance in the light of partial deltas. As we will see, the propagation of partial source deltas will generally result in view deltas that are partial again. However, partial deltas are useful for view maintenance. Several CDA techniques exist that allow for applying partial deltas to materialized views. The choice of a suitable CDA technique depends on the type of the view deltas. In this section, we present a brief survey of CDA techniques and discuss a trade-off between the effort spent on change capture and the effort required for change application.

Insertions To apply insertions, the SQL interface provides the INSERT statement. Furthermore, many databases provide bulk loading facilities to insert larger batches of records in an efficient manner and major ETL tools usually provide adapters to such bulk loading facilities.

(Partial) deletions Assuming that the target table has a primary key column, just key values are required to apply deletions, whereas attribute values are not. Thus partial deletions are sufficient for view maintenance in such cases. Non-partial deletions are, however, relevant during delta propagation. As we will see in the following, the propagated view deltas are generally "less partial" when non-partial deletions are available at the sources. The dependencies between the type of source deltas and the type of the propagated view deltas will be discussed in detail in the following sections.

(Partial) updates Much like partial deletions, partial updates are sufficient for maintaining views with key columns. In contrast to partial updates, complete update pairs include the initial state of updated tuples. To update a tuple in place, the old state is not required and a partial update is sufficient. However, the data warehouse often keeps historical data.

The data history is typically established using the Slowly Changing Dimension techniques that has been discussed in Section 2.5.1 on dimensional modeling. Recall that the choice for the Slowly Changing Dimension technique type 1 or 2 is typically based on which attributes have been updated. Given a complete update delta, this can be found out easily by comparing the initial state of the updated tuple with its current state. Given a partial update delta, however, a warehouse lookup is required to find out about the initial values.

Upserts To apply upserts, one can either attempt an UPDATE first and issue an INSERT if no rows were affected or else run an INSERT first and issue an UPDATE if the inserted key violates the uniqueness constraint. Which of these approaches is more efficient depends on the ratio of the number of insert operations to the number of update operations. Many ETL tools offer database adapters that implement this inserts-else-update or update-else-insert method. However, while this method seems convenient and may help to simplify the flow of data, it has been criticized as being rather inefficient [KC04].

In the latest SQL standard the MERGE statement has been introduced to work around this issue. MERGE can be used to insert or update tuples depending on whether a user-defined condition matches. The MERGE technique is more efficient than the insert-else-update or update-else-insert approach. However, it performs worse than applying insert deltas and update deltas in

separation. The latter approach, however, is only possible when inserts and updates are given in two distinct delta sets, i.e. for non-partial deltas.

It is interesting to understand the dependencies between change capture and change application in the context of partial deltas: While more partial deltas can often be captured more efficiently, the application of more partial deltas is less efficient. Thus, there is a trade-off between the effort for change capture and change application. Note that these steps are performed at distinct systems. It is thus possible to shift workload from the source systems to the warehouse by capturing more partial deltas or, vice versa, by capturing less or non-partial deltas.

5.3 The impact of partial deltas on view maintenance

In this section, we will explore how algebraic view maintenance is impacted when deltas are captured only partially at the sources. We will give an example to show that the standard approach may produce incorrect results for partial source deltas and thus, may corrupt the materialized view during view maintenance. We will analyze the root cause of the problem in this section. Based on the findings of our analysis, we will propose a generalized algebraic view maintenance algorithm that is capable of propagating partial deltas in the following section.

As an example, consider the customer and address datasets shown in Figure 5.1 and denoted by C and A, respectively. The datasets are shown in both, their old state before being modified and their new state. The deltas captured at these base tables are shown below. We assume that the CDC technique applied at the customer table provides non-partial deltas, while the CDC technique applied at the address table provides partial updates and partial deletions only. Hence, the initial state of updated tuples (AID 1) is not available and no corresponding delta tuple is found in ∇A. Furthermore, only the primary key of deleted tuples (AID 3) is captured and available in ∇A, whereas the other attribute values are not. The unavailable attributes are assigned with null values.

Consider the following sample view definition. It is based on the sample ETL job presented in Chapter 3, but has been simplified by removing the data cleansing operators to focus on the most relevant aspects of this chapter.

$$\mathcal{D} = \pi_{CID,CName,AID}(C) \bowtie \sigma_p(A)$$

The sample view definition applies a projection to the customer dataset, a selection to the address dataset (to restrict to certain countries), and joins both datasets together. By algebraic differencing, we obtain two incremental

C_{old}

CID	CName	CDiscount	AID
1	Adam	0	1
2	Bob	0	2
3	Carl	0	3

A_{old}

AID	ACity	ACountry
1	Aachen	DE
2	Berlin	DE
3	Celle	DE

C_{new}

CID	CName	CDiscount	AID
1	Adam	5	1
2	Bob	0	4
4	Dave	0	4

A_{new}

AID	ACity	ACountry
1	Essen	DE
2	Berlin	DE
4	Dresden	DE

ΔC

CID	CName	CDiscount	AID
1	Adam	5	1
2	Bob	0	4
4	Dave	0	4

ΔA

AID	ACity	ACountry
1	Essen	DE
4	Dresden	DE

∇C

CID	CName	CDiscount	AID
1	Adam	0	1
2	Bob	0	2
3	Carl	0	3

∇A (partial)

AID	ACity	ACountry
3	-	-

A_{old} (reconstructed as $A_{new} - \Delta A \cup \nabla A$)

AID	ACity	ACountry
2	Berlin	DE
3	-	-

D_{old}

CID	CName	AID	ACity	ACountry	
1	Adam	1	Aachen	DE	updated
2	Bob	2	Berlin	DE	updated
3	Carl	3	Celle	DE	deleted

D_{new}

CID	CName	AID	ACity	ACountry	
1	Adam	1	Essen	DE	updated
2	Bob	4	Dresden	DE	updated
4	Dave	4	Dresden	DE	inserted

ΔD

CID	CName	AID	ACity	ACountry	
1	Adam	1	Essen	DE	insertion
2	Bob	4	Dresden	DE	update
4	Dave	4	Dresden	DE	insertion

∇D

CID	CName	AID	ACity	ACountry	
2	Bob	2	Berlin	DE	update

Table 5.1: Sample datasets and deltas

expressions $\Delta\mathcal{D}$ and $\nabla\mathcal{D}$ from the view definition that compute the insertions and deletions to the materialized view, respectively.

$$\begin{aligned}\Delta\mathcal{D} = &[(\Pi_{CID,CName,AID}(C_{new})) \bowtie \sigma_p(\Delta A)] \cup \\ &[(\Pi_{CID,CName,AID}(\Delta C) - \Pi_{CID,CName,AID}(C_{old})) \bowtie \sigma_p(A_{new})]\end{aligned}$$

$$\begin{aligned}\nabla\mathcal{D} = &[(\Pi_{CID,CName,AID}(C_{old})) \bowtie \sigma_p(\nabla A)] \cup \\ &[(\Pi_{CID,CName,AID}(\nabla C) - \Pi_{CID,CName,AID}(C_{new})) \bowtie \sigma_p(A_{old})]\end{aligned}$$

It is important to understand, that the old state of the address table A_{old} is not directly available to evaluate the above expressions, because operational sources typically store the most current version of base tables only. However, the old state of a base relation can be reconstructed from the new state by subtracting the insert deltas and adding the delete deltas, i.e. $A_{old} := A_{new} - \Delta A \cup \nabla A$. As shown in the example, it may not be possible to fully reconstruct the old state based on partial deltas however.

The materialized view D is shown in Figure 5.1 in two states, its old state (D_{old}) derived from the base data before it has been modified, and its new state (D_{new}) derived from the base data after it has been modified. Note that two tuples have been updated in D (CID 1 and 2), one tuple has been deleted (CID 3), and one tuple has been inserted (CID 4).

These changes should also be reflected by the view deltas ΔD and ∇D computed by the incremental expressions $\Delta\mathcal{D}$ and $\nabla\mathcal{D}$, respectively, which have been derived from the view definition. However, as shown in Figure 5.1, the view deltas are computed incorrectly, because the source deltas are partial. There are two types of anomalies that may occur during view maintenance using partial source deltas.

First, updates may incorrectly be reported as being insertions. Recall that updates are modeled implicitly by delete-insert pairs. Hence, the set of insertions contains all tuples in ΔD for which no tuple with the same primary key value exists in ∇D. Similarly, the set of updates contains all pairs of tuples in ΔD and ∇D having the same primary key value. The view deltas shown in Figure 5.1 thus contain two insertions (CID 1 and 4) and a single update (CID 2). A comparison between the new state D_{new} and the old state D_{old} of the materialized view reveals that this result is incorrect. Note that the tuple with CID 1 is contained in both D_{new} and D_{old} and hence needs to be updated during incremental view maintenance. However, a delta tuple with CID 1 is found in ΔD but no such tuple exists in ∇D. That is, an insertion delta is propagated, where an update delta would have been correct.

Note that the update of the tuple with CID 2 and the insertion of the tuple with CID 4 are propagated correctly. However, there is no obvious way to

distinguish these correct deltas from the incorrect one. The standard algebraic view maintenance approach is thus generally unable to distinguish between insertions and updates when the source deltas are partial. Recall that we referred to this type of partial deltas as upserts before. As discussed in Section 5.2 upserts can be applied to a materialized view by performing an initial lookup to determine whether a corresponding tuple already exist. However, simply propagating upserts instead of disjoint sets of insertions and updates is undesirable, because of the lookup overhead.

There is a second kind of anomaly that may occur during view maintenance using partial source deltas – deletions may be propagated only incompletely to the materialized view. Note that a tuple (CID 3) exists in the old state of the materialized view D_{old} that is no longer found in its new state D_{new} and thus, needs to be deleted during incremental view maintenance. However, the delta set ∇D computed for view maintenance does not contain a corresponding delta tuple. In consequence, the tuple remains in the materialized view after it is maintained incrementally and the view is corrupted permanently. Unlike the first kind of anomaly we discussed (indistinguishable updates and insertions), the problem of missing deletions cannot be fixed by an alternative change application technique.

In summary, our example has shown that standard algebraic view maintenance struggles when the deltas captured at the sources are partial; updates may incorrectly be propagated as insertions and the propagated deletion set may be incomplete. We will now look into the reasons that cause these anomalies to happen. In the following section, we will propose a generalized approach to algebraic view maintenance that allows for propagating partial deltas such that anomalies are avoided.

Recall that the deltas of a relation R are modeled as two sets, the set of insertions ΔR and the set of deletions ∇R. We distinguished several types of partial deltas, namely partial updates, upserts, partial deletions and missing deletions. Note that for each type of partial deltas, the change information encoded in the insertion delta set ΔR is completely provided, whereas the change information encoded in the deletion delta set ∇R is incomplete.

Reconsider the sample source datasets shown in Figure 5.1. The deletion that occurs at the address dataset A is captured only partially, i.e. no attribute values apart from the primary key value are captured. Hence, the missing attributes are padded with null values in the deletion delta set ∇A. According to the view definition, a selection is applied to A, which drops tuples not satisfying the given predicate. Note that the same predicate is evaluated on the deletion delta set ∇A in the incremental expressions $\nabla \mathcal{D}$. However, the predicate involves non-key attributes – which is the common case – and it is thus unclear whether it is satisfied by the partial deletion. Since the missing attributes were assigned with null values in the example, the predicate evaluates

to false and the delta tuple is dropped. For this reason the partial deletion is not further propagated and, as a result, a deletion is missing in the view delta set ∇D.

Apart from the partial deletion, a partial update is captured at A. Recall that only the current state of the updated tuple is known whereas its initial state is not. Hence, a delta tuple encoding the current state is added to the insertion delta set ΔA. It is however unclear how a delta tuple to be added to the deletion delta set ∇A should look like, because the partial update does not include the initial state of the updated tuple. In the example, such a delta tuple is simply omitted. The incremental expressions $\Delta\mathcal{D}$ and $\nabla\mathcal{D}$ include joins between C_{new} and ΔA and C_{old} and ∇A, respectively. Since the partial update is encoded in ΔA and a matching tuple is found in C_{new}, the first join produces a corresponding result delta tuple. However, since the partial update is not encoded in ∇A the latter join cannot produce a result tuple. Hence only the "insert part" of the partial update is further propagated while the "delete part" is not. In consequence the computed view deltas include an insert delta where an update delta would have been appropriate.

In summary, the anomalies that occur during view maintenance using partial source deltas are caused by two problems. There is no obvious way to evaluate predicates that rely on unknown attribute values and, similarly, there is no obvious way to evaluate joins that involve (partly) unknown delta sets. As we have shown, the consequence for standard algebraic view maintenance is that inserts and updates cannot reliably be distinguished in the computed view deltas and, even worse, the view deltas may not contain all deletions needed to maintain the materialized view in a correct manner. This situation is clearly unsatisfying and we will thus propose an improved view maintenance approach in the following section.

5.4 A generalized approach to algebraic view maintenance

In this section, we will generalize the algebraic view maintenance approach to work for partial source deltas such that the anomalies discussed in the previous section are avoided. In Section 5.4.1 we will introduce a formal model for partial deltas that will serve as both, the model for representing source deltas and the underlying model of the change propagation process. We will show that in the general case, views cannot be maintained using partial source deltas such that any anomaly is avoided. However, we will show that this is possible for a certain class of views that we will call *dimension views*. The class of dimension views will be characterized in Section 5.4.2.

In Section 5.4.3 we will discuss the maintenance of dimension views given partial source deltas. Basically, we will extend the notion of partial deltas

from change capture to the change propagation process. For this purpose, we will generalize the delta rules for algebraic differencing to work for partial deltas. Finally, we will show that partial deltas are closed under the relational operations admissible in dimension view definitions.

5.4.1 A formal model for partial deltas

The survey on CDC techniques presented in Section 5.1 has shown that deltas may be captured at different levels of completeness at the source systems. We developed an intuitive notion of partial deltas and distinguished between three types that we called partial updates, upserts, and partial deletions. In this section, the intuitive notion of partial deltas will be refined and captured in a formal model.

From an abstract point of view, deltas may be partial in two respects. Deltas may lack information on certain attribute values as it is the case for partial updates and partial deletions. We will refer to this kind of deltas as *value-partial* deltas. Additionally, deltas may lack information about the exact type of the modification and we will refer to this kind of deltas as *type-partial* deltas.

If a delta is type-partial, we cannot know whether the modified tuple used to affect the materialized view before it has been modified. Consider an upsert delta captured at the sources. It may have been propagated to the materialized view before, if it is in fact an update, or it may not have been propagated, if it is in fact an insertion. It is thus unclear whether a derived tuple is found in the materialized view or not.

Similar to partial updates and upserts, two kinds of partial deletions can be distinguished that are value-partial and type-partial and referred to as true partial deletions and safe partial deletions, respectively. True partial deletions will, when propagated to the materialized view, cause a view tuple to be deleted. That is, each true partial deletion is effective. Safe partial deletions can be thought of as an overestimate of view tuples to be deleted. Hence, safe partial deletions either cause a view tuple to be deleted or do not have an effect on the view, if no such tuple can be found. That is, safe partial deletions are either effective or ineffective. The CDC techniques we discussed in Section 5.1 provide (at least) true partial deletions. Safe partial deletions may, however, arise during change propagation as will be discussed in Section 5.4.3. A formal model for partial deltas is defined as follows.

Definition 5.1 (Partial deltas) *Let $R(pk, a)$ be a relation with primary key pk and a set of attributes a. Let R_{old} be the state of R before it is changed and R_{new} the state of R hereafter. Partial deltas are a seven-tuple of sets $(R_{ins}, R_{un/uo}, R_{del}, R_{pup}, R_{ups}, R_{tpdel}, R_{spdel})$ where*

- *$R_{ins}(pk, a) \subseteq R_{new}$ denotes a set of tuples inserted into R (insertions),*

- $R_{del}(pk, a) \subseteq R_{old}$ *denotes a set of tuples deleted from* R *(deletions),*
- $R_{un/uo}(pk, a_{un}, a_{uo})$ *with* $\pi_{pk,a_{un}}(R_{un/uo}) \subseteq R_{new}$ *and* $\pi_{pk,a_{uo}}(R_{un/uo}) \subseteq R_{old}$ *denotes a set of tuples updated in* R *(updates). The initial state and the current state of updated tuples is given by* (pk, a_{uo}) *and* (pk, a_{un})*, respectively,*
- $R_{pup}(pk, a) \subseteq R_{new}$ *denotes a set of tuples updated in* R *in their current state only (partial updates),*
- $R_{ups}(pk, a) \subseteq R_{new}$ *denotes a set of tuples either inserted or updated in* R *(upserts),*
- $R_{tpdel}(pk, b) \subseteq \pi_{pk,b}(R_{old})$ *with* $b \subseteq a$ *denotes a set of tuples deleted from* R *of which only the primary key* pk *and a subset* b *of the attributes is known (true partial deletions),*
- $R_{spdel}(pk, a)$ *with* $\forall d \in R_{spdel} \forall o \in \pi_{pk,b}(R_{old}) : \pi_{pk}(d) = \pi_{pk}(o) \rightarrow d = o$ *denotes a set of tuples deleted or not existent in* R *of which only the primary key* pk *and a subset* b *of the attributes is known (safe partial deletions),*

such that each change at the tuple level from R_{old} *to* R_{new} *is reflected by exactly one tuple in one of the delta sets* R_{ins}*,* R_{del}*,* $R_{un/uo}$*,* R_{ups}*,* R_{tpdel}*, or* R_{spdel}*. That is, the primary key values are pairwise disjoint across the delta sets and*

$$\pi_{pk}(R_{new} - R_{old}) = \pi_{pk}(R_{ins} \cup R_{un/uo} \cup R_{pup} \cup R_{ups}) \text{ and}$$
$$\pi_{pk}(R_{old} - R_{new}) \subseteq \pi_{pk}(R_{del} \cup R_{un/uo} \cup R_{tpdel} \cup R_{spdel}).$$

Figure 5.3 depicts a hierarchy of partial deltas. Each data modification at the tuple level is represented by a single delta tuple in one of the delta sets. However, a delta tuple may appear in different delta sets. The alternative placements are indicated by the arrows. The completeness decreases while moving from the upper to the lower delta sets. As the figure suggests, type-partial deltas are always value-partial, but not vice versa.

5.4.2 Dimension views

In general, materialized views cannot be maintained using partial deltas. Consider the following example. Say there is a base relation $R(pk, a)$ with pk being the primary key column and a simple derived view $V(a) := \pi_a(R)$. Say we use a CDC mechanism that captures partial updates (such as audit columns or change tracking). Obviously, V cannot be maintained in case of an update

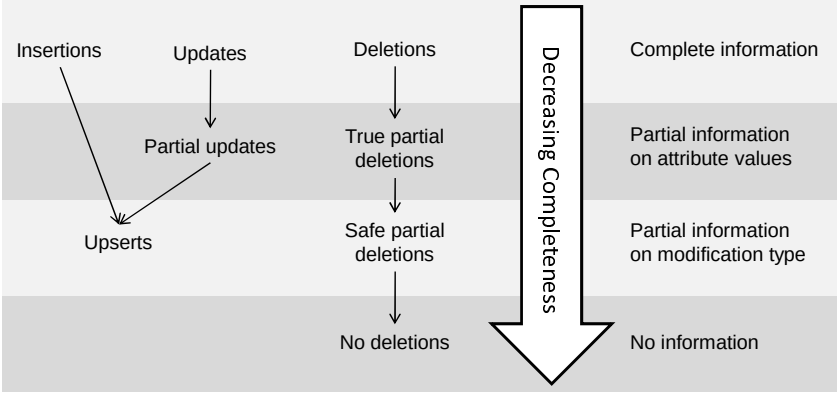


Figure 5.3: Partial deltas with decreasing completeness

to R. While it is straightforward to add the updated tuple to V, it is unclear which tuple in V needs to be discarded (or overwritten) in return. Similar considerations hold for the other kinds of partial deltas, i.e. partial deletions and upserts.

Note, that V was maintainable if it included the primary key column pk. Including primary keys is thus a necessary condition for views to be maintainable using partial deltas. However, not all primary keys from the source relations need to be retained in the view definition. In particular, the view does not need to be self-maintainable w.r.t. deletions. We will discuss the selection of keys in the following. At first, we define a class of views, which we call dimension views, that has interesting properties w.r.t. maintenance using partial deltas.

Definition 5.2 (Dimension view) *Let R_i $(1 \leq i \leq n)$ be a set of relations (not necessarily distinct) and pk_i denote the (composite) primary key of R_i. Let V be a view defined by a relational expression of the following form or any possible rewriting.*

$$\pi_A(\sigma_s(R_1 \bowtie_{p_1} R_2 \bowtie_{p_2} \ldots \bowtie_{p_{n-1}} R_n))$$

V is called "dimension view" if $pk_1 \in A$, all joins are equi-joins, and each join predicate p_i involves pk_{i+1}.

In other words, dimension views join base relations along foreign key relationships and include those primary key attributes that are not functionally dependent on any other key attributes. In the remainder of this chapter, we will show that dimension views are maintainable using partial source deltas.

Note that the view definition $\mathcal{D}$ presented as an example in Section 5.3 is a dimension view w.r.t. Definition 5.2.

Dimension views are commonly found in data warehouses. While they are usually called dimension tables here, they store derived data and can thus be seen as views. Data warehouses typically use a star schema to store multi-dimensional data that consists of fact and dimension tables [KC04, KR02]. Dimension tables are used to join together data on business entities that originates from multiple source systems. For improved query performance, dimension tables are typically denormalized. Dimension tables include a unique identifier for business entities referred to as business key. Typically, no other keys from the sources are stored here. Note that these keys would be functionally dependent on the business key in the denormalized dimension table. Our work is thus directly applicable to incremental maintenance of dimension tables.

5.4.3 Generalized delta rules for algebraic differencing

We propose a generalization of the algebraic view maintenance approach that allows for maintaining dimension views using partial deltas. We proceed as follows: First, we explain how partial deltas can be represented by means of delete-insert sets used by the original algorithm. Second, we propose generalized delta rules for projection, selection, and join. We show that partial deltas are closed under each of these operators. Third, we conclude that dimension views can be maintained by our algorithm.

It is common for view maintenance algorithms to model deltas by two sets, i.e. the set of deleted tuples and the set of inserted tuples. Updates are implicitly given by delete-insert pairs. We will therefore refer to this model as implicit delta model or implicit deltas for short. The implicit delta model does not directly match the model for partial deltas defined in Section 5.4.1. The latter uses seven distinct sets to explicitly distinguish between insertions, updates, partial updates, upsert, deletions, true partial deletions, and safe partial deletions. We will therefore refer to it as explicit delta model or explicit deltas for short.

While the explicit delta model allows for a natural representation of partial deltas, it is more complex to handle seven distinct sets during change propagation. We experienced that delta rules become rather complex. In particular, the join delta rules require a large number of joins to capture the interactions between the different delta sets.

Overly complex incremental expressions can be avoided by using the implicit delta model as the underlying model for change propagation. To do so, partial deltas need to be transferred to the implicit model. Furthermore, the result of the change propagation needs to be converted back into the explicit model for change data application. To do so, we extend the schema of the delta sets by

$$\begin{aligned}\Delta R(pk, a) :=& R_{ins} \cup \pi_{pk, a_{un}}(R_{un/uo}) \cup R_{pup} \cup R_{ups} \\ \nabla R(pk, a, flag) :=& \pi_{pk, a, \mathsf{comp}}(R_{del}) \cup \pi_{pk, a_{uo}, \mathsf{comp}}(R_{un/uo}) \cup \pi_{pk, \mathsf{NULL}, \mathsf{pval}}(R_{pup}) \cup \\ & \pi_{pk, \mathsf{NULL}, \mathsf{ptype}}(R_{ups}) \cup \pi_{pk, \mathsf{NULL}, \mathsf{pval}}(R_{tpdel}) \cup \pi_{pk, \mathsf{NULL}, \mathsf{ptype}}(R_{spdel}) \\ & \text{with } flag \in \{\mathsf{comp}, \mathsf{pval}, \mathsf{ptype}\}\end{aligned}$$

Figure 5.4: Conversion from explicit deltas to implicit deltas

adding a type flag column. This flag is used to indicate the type of individual delta tuples. Unknown attribute values (of partial deltas) are padded with null values. One could say that partial deltas are "encoded" as special kinds of complete deltas. We proceed by describing the conversion from the explicit delta model to the implicit delta model, and continue with the conversion in opposite direction.

Explicit delta model to implicit delta model The equations for converting from the explicit delta model to the implicit delta model are given in Figure 5.4. For a relation R it is straightforward to express the explicit delta sets R_{ins}, $R_{un/uo}$, and R_{del} in the implicit model, because these sets are non-partial. To distinguish complete delta tuples from partial ones, they are assigned a type flag with the value comp.

The remaining delta sets are treated as follows: Partial update tuples R_{pup} are added to the insert set ΔR. To distinguish them from insertions and complete updates, an additional tuple is added to the delete set ∇R for each partial update. These additional tuples are assigned with the same primary key value, while all remaining attributes are padded with null values, because the initial attribute values before the update are unknown. Furthermore, a type flag with the value pval is assigned to indicate that the delta tuple is value-partial. Note that the schema of ∇R is extended to accommodate the type flag.

Upserts are handled somewhat similarly. Like partial updates, upserts are added to the insert set ΔR and an additional tuple with the same primary key value and null-padded attributes is added to the delete set ∇R. In contrast to partial updates, upserts are type-partial. This is indicated by assigning the value ptype to the type flag in the delete delta set.

Partial deletions are added to ∇R. Because only the primary key value is known for partial deletions, all remaining attributes are again padded with null values. True partial deletions R_{tpdel} are assigned with a type flag with the value pval, while safe partial deletions R_{spdel} are assigned with a type flag with the value ptype. Note that partial deletions can be distinguished from partial updates and upserts, because the former do not have corresponding tuples in

ΔC

CID	CName	CDiscnt	AID
1	Adam	5	1
2	Bob	0	4
4	Dave	0	4

ΔA

AID	ACity	ACountry
1	Essen	DE
4	Dresden	DE

∇C

CID	CName	CDiscnt	AID	flag
1	Adam	0	1	comp
2	Bob	0	2	comp
3	Carl	0	3	comp

∇A

AID	ACity	ACountry	flag
1	-	-	pval
3	-	-	pval

Figure 5.5: Sample deltas converted to the implicit model

$$
\begin{aligned}
R_{ins} &:= \Delta R \ \overline{\ltimes}_{pk} \ \nabla R \\
R_{del} &:= \nabla R \ \overline{\ltimes}_{pk} \ \sigma_{(flag=\mathsf{comp})} \Delta R \\
R_{un/uo} &:= \Delta R \bowtie_{pk} \sigma_{(flag=\mathsf{comp})} \nabla R \\
R_{pup} &:= \Delta R \ltimes_{pk} \sigma_{(flag=\mathsf{pval})} \nabla R \\
R_{ups} &:= \Delta R \ltimes_{pk} \sigma_{(flag=\mathsf{ptype})} \nabla R \\
R_{tpdel} &:= \pi_{pk}(\nabla R \ \overline{\ltimes}_{pk} \ \sigma_{(flag=\mathsf{pval})} \Delta R) \\
R_{spdel} &:= \pi_{pk}(\nabla R \ \overline{\ltimes}_{pk} \ \sigma_{(flag=\mathsf{ptype})} \Delta R)
\end{aligned}
$$

Figure 5.6: Conversion from implicit deltas to explicit deltas

the insert set ΔR while the latter do.

Example 5.1 *Recall the running example introduced in Section 5.3 before. Figure 5.1 depicts both, the old and the new state of the base relations C and A. Recall that non-partial deltas are captured at C while partial updates and partial deletions are captured at A. The deltas converted to the implicit model are depicted in Figure 5.5.*

Implicit delta model to explicit delta model The equations for converting from the implicit delta model back to the explicit delta model are given in Figure 5.6. Note that the symbols $\ltimes$ and $\overline{\ltimes}$ are used to denote semi join and anti join, respectively. The set of insertions R_{ins} consists of those tuples in ΔR that have a primary key value that does not exist in ∇R. Similarly, the set of deletions R_{del} consists of those tuples in ∇R that have a primary key value that does not exist in ΔR and are complete, i.e. have a type flag with a value of comp.

The set of complete updates $R_{un/uo}$ consists of pairs of tuples in ΔR and ∇R having equal primary key values and being complete, i.e. the tuples in ∇R

have a type flag with a value of comp. The set of partial updates R_{pup} and the set of upserts R_{ups} consist of tuples in ΔR that join to tuples in ∇R having a type flag with a value of pval or ptype, respectively.

The set of partial deletions consist of tuples in ∇R having a primary key that does not exist in ΔR. The set of true partial deletions R_{tpdel} can be distinguished from the set of safe partial deletions R_{spdel} based on the type flag. Tuples in the former set have a type flag with a value of pval while tuples in the latter set have a type flag with a value of ptype.

In Chapter 4, we proposed adaptations to the original delta rules for algebraic differencing. In the following, we will generalize the adapted delta rules for projection, selection, and join for the propagation of partial deltas.

5.4.4 Projection

Recall that dimension views contain primary key attributes that must not be dropped by a projection. That is each projection operator in a dimension view definition is key-preserving in the sense of Section 4.2. The delta rules for key-preserving projection depicted in Table 4.1 are repeated for the reader's convenience.

$$\Delta(\pi_A(S)) \equiv \pi_A(\Delta S) - \pi_A(\nabla S)$$
$$\nabla(\pi_A(S)) \equiv \pi_A(\nabla S) - \pi_A(\Delta S)$$

Recall that the aim of the effectiveness test is to discard ineffective updates, i.e. updates that do not change the view. In the presence of keys, an ineffective update occurs when all updated attributes are dropped by the projection. In this case, the initial state of the propagated attributes is equal to their current state. Thus, the update is ineffective w.r.t. the view.

We now discuss the implications of partial deltas w.r.t. the effectiveness test. In fact, the test may not work as expected here. The reason is that the initial state of an updated tuple may not be available. Hence, its effectiveness cannot be tested. Without having the initial state available, we do not know which attributes have been updated. Hence, we cannot know whether the update will affect the view. However, propagating ineffective updates is not problematic, because a view is not changed when an ineffective update is applied. While ineffective updates cause some overhead, the view does not become inconsistent.

Delta rules for projection can be generalized to handle partial deltas. To this end, the effectiveness test is only done for complete delta tuples and omitted for partial ones.

$$\Delta(\pi_A(S)) \equiv \pi_A(\Delta S) - \pi_A(\sigma_{flag=\mathsf{comp}}\nabla S)$$
$$\nabla(\pi_A(S)) \equiv \pi_{A,flag}(\nabla S) - \pi_{A,\mathsf{comp}}(\Delta S)$$

$\Delta C'$

CID	CName	AID
2	Bob	4
4	Dave	4

$\nabla C'$

CID	CName	AID	flag
2	Bob	2	comp
3	Carl	3	comp

Table 5.2: Result of the sample incremental expressions $\Delta C'$ and $\nabla C'$

In the second delta rule, the schema of ΔS is extended by adding a type flag column which is assigned the value comp. Note that the effectiveness test may safely be omitted. Doing so may result in a larger number of ineffective updates being propagated. However, the rules become even simpler and may be evaluated more efficiently. Furthermore, note that partial deltas are closed under projection using the above delta rule. That is, the result of the operation is again partial deltas.

Example 5.2 *Reconsider the running example. The first term of the dimension view definition $\mathcal{D}$ is $\mathcal{C}' := \pi_{CID,CName,CAddr}(C)$. We can apply the delta rules given above to derive incremental expressions $\Delta\mathcal{C}'$ and $\nabla\mathcal{C}'$.*

$$\Delta\mathcal{C}' \equiv \pi_{CID,CName,AID}(\Delta C) - \pi_{CID,CName,AID}(\sigma_{flag=\mathsf{comp}}(\nabla C))$$
$$\nabla\mathcal{C}' \equiv \pi_{CID,CName,AID,flag}(\nabla C) - \pi_{CID,CName,AID,\mathsf{comp}}(\Delta C)$$

The result deltas are depicted in Figure 5.2. Note that the update to the customer tuple with ID 1 is ineffective and thus discarded.

5.4.5 Selection

The delta rules for selection found in Table 4.1 are repeated here for the reader's convenience.

$$\Delta(\sigma_p(S)) \equiv \sigma_p(\Delta S) \qquad \nabla(\sigma_p(S)) \equiv \sigma_p(\nabla S)$$

These equations need to be adapted to handle partial deltas. We will discuss each type of delta separately in the following. The inserted tuples in S_{ins} are propagated if they satisfy the selection predicate and discarded otherwise. The deleted tuples in S_{del} are treated similarly. For the update pairs in $S_{un/uo}$ both, the initial and the current state have to be considered. If the initial and the current state satisfy the selection predicate the delta tuple is passed on as an update, i.e. it remains in $S_{un/uo}$. If neither the initial nor the current state satisfy the selection predicate the update pair is discarded. If the initial state did satisfy the predicate but the current state no longer does, the initial state is propagated as a deletion, i.e. it becomes part of S_{del}. Similarly, if the initial state did not satisfy the predicate but the current state does, the current state is propagated as an insertion, i.e. it becomes part of S_{ins}.

For partial updates S_{pup} the initial state is not known. It could have either satisfied the selection predicate or not. Thus, given that the current state does satisfy the predicate, the resulting delta is either an insert or an update. Since we cannot distinguish these cases, the delta tuple becomes part of S_{ups}, i.e. the delta tuple changes its type and becomes an upsert. Given that the current state of an partial update does not satisfy the selection predicate, a deletion needs to be propagated. Since the initial state of the updated tuple is unavailable, a partial deletion is propagated. Note that this deletion may be ineffective, i.e. the tuple to be deleted may not be found in the view, because it did not satisfy the predicate in its initial state either. That is, the delta tuple changes its type and becomes a safe partial deletion (S_{spdel}).

The upsert delta set S_{ups} is handled in a very similar way as partial updates. Again, the initial state of delta tuples is unavailable. Given that the current delta tuple satisfies the selection predicate, it remains in S_{ups}. If it does not, it becomes a safe partial deletion (S_{spdel}), because it may, again, be ineffective.

Partial deletions may not contain all attributes for the selection predicate to be evaluated. However, partial deletions may safely be propagated anyway. If a tuple with the same primary key value exists in the view, it needs to be deleted. If no such tuple exists, the view remains unchanged, i.e. the deletion turns out to be ineffective. Thus, true partial deletions are turned into safe partial deletions if propagated through a selection operator. Ineffective deletions occur, when the original tuple did not satisfy the selection predicate and therefore never appeared in the view.

$$\Delta(\sigma_p(S)) \equiv \sigma_p(\Delta S) \qquad \nabla(\sigma_p(S)) \equiv \sigma_{p \vee flag \neq \mathsf{comp}}(\pi_{a, s(flag)} \nabla S)$$

$$\text{with } s(flag) := \begin{cases} \mathsf{comp} & \text{if } flag = \mathsf{comp} \\ \mathsf{ptype} & \text{if } flag = \mathsf{pval} \\ \mathsf{ptype} & \text{if } flag = \mathsf{ptype} \end{cases}$$

Given these considerations, the original delta rules for selection can be adapted to partial change data. The rule to compute the insert set does not need to be changed. The rule to compute the delete set needs some adaptations though, because the selection predicate can only be checked for non-partial delta tuples (with $flag = \mathsf{comp}$). All partial delta tuples are simply passed on. As mentioned before, value-partial deltas turn into type-partial deltas and the type flag needs to be changed accordingly. Note that partial deltas are closed under selection using the above delta rules.

Example 5.3 *Reconsider the running example. The second term of the dimension view definition $\mathcal{D}$ is $\mathcal{A}' := \sigma_p(A)$. By applying the above delta rules*

$\Delta A'$

AID	ACity	ACountry
1	Essen	DE
4	Dresden	DE

$\nabla A'$

AID	ACity	ACountry	flag
1	-	-	ptype
3	-	-	ptype

Table 5.3: Result of the sample incremental expressions $\Delta Addr'$ and $\nabla Addr'$

the incremental expressions $\Delta\mathcal{A}'$ *and* $\nabla\mathcal{A}'$ *can be derived.*

$$\Delta\mathcal{A}' \equiv \sigma_p(\Delta A)$$
$$\nabla\mathcal{A}' \equiv \sigma_{p \vee flag \neq \mathsf{comp}}(\nabla A)$$

The result deltas are depicted in Figure 5.3. Note that partial updates are turned into upserts and true partial deletions into possibly ineffective safe partial deletions.

5.4.6 Join

The join delta rules found in Table 4.1 are repeated here for the readers convenience. Note that besides the delta relations the base relations are required to incrementally maintain join views.

$$\Delta(S \bowtie T) \equiv (S_o \bowtie \Delta T) \cup (\Delta S \bowtie T_o) \cup (\Delta S \bowtie \Delta T)$$
$$\nabla(S \bowtie T) \equiv (S_o \bowtie \nabla T) \cup (\nabla S \bowtie T_o) \cup (\nabla S \bowtie \nabla T)$$

In the data warehouse environment, source systems are decoupled and base relations are usually available in their new state only. However, the preserved state referenced in the above equations can be reconstructed from the new state and the insert deltas. Given a relation R, the preserved state R_o can be computed by subtracting the insert delta set from the new state ($R_o := R_{new} - \Delta R$). Note that both, the R_{new} and ΔR are non-partial and thus R_o is non-partial too. When used in the deletion delta rule, the preserved state relation R_o is hence assigned with a comp type flag.

In this chapter, we focus on the maintenance of so-called dimension views defined in Section 5.4.2. All join predicates used in dimension views follow a common pattern. They are equality predicates and involve the primary key attribute of at least one relation. In the following, we consider the join of two relations S and T with the join predicate ($S.a = T.pk$) where $S.a$ is an arbitrary attribute of S and $T.pk$ the primary key attribute of T. It is important to understand that the type of the resulting (joined) delta tuples depends on the type of both input delta tuples. To adapt the join delta rules, all possible combinations of delta types need to be considered. The different combinations are represented by the matrices in Figure 5.7 and Figure 5.8. Consider the

		S						
		pre	ins	del	pup	ups	tpdel	spdel
T	pre	-	ins	del	ups	ups	spdel	spdel
	ins	ins	ins	-				
	del	del	-	del				
	un	un	ins	-				
	uo	uo	-	del				
	pup	pup	ins	-				
	ups	ups	ins	-				
	tpdel	tpdel	-	tpdel				
	spdel	spdel	-	spdel				

Figure 5.7: Partial delta type join matrix

matrix in Figure 5.7. The column headings represent the different delta sets of S participating in the join. From left to right, there are preserved tuples, insertions, deletions, partial updates, upserts, true partial deletions, and safe partial deletions. For the sake of clarity, update pairs are shown in a separate matrix (Figure 5.8). The row headings in the matrix represent the different delta sets of T participating in the join. The cells of the matrix indicate the delta type resulting from a join between the corresponding delta sets of S and T.

Consider the matrix cell at the intersection of the S_{ins} column and the T_{pup} row, for instance. The cell indicates that the join result of these delta sets is to be propagated as insertion. This is obvious considering that any tuple added to S has a key that is unique in S. Thus, the key cannot be in the view yet. Hence, the result of the join is an insertions w.r.t. the view.

Consider the four right-most columns in the matrix referring to partial updates, upserts, and true and safe partial deletions in S. These deltas lack certain attribute values. They hence cannot be joined to T_{old}, because the join predicate cannot be evaluated. Consider a partial update in S. Recall, that the initial state of the updated tuple is not available. Hence, it is unclear whether the updated tuple used to find a join partner in T_{old} before the update. Assuming that the updated tuple joins to a tuple in T_{new}, the result tuple is either an update to the view (if the updated tuple used to find a join partner before) or an insertion (if it did not). Since these cases cannot be distinguished for partial updates, an upsert has to be propagated.

Similarly, it is unclear whether partial deletions in S used to find a join partner in T_{old} before the deletion. The propagated partial deletion is thus either effective (if it used to find a join partner) or ineffective (if it did not). Both true and safe partial deletion, are hence propagated as safe partial deletions.

Consider an upsert in S. For upserts, it is unknown whether the tuple was in fact updated or inserted. If an upsert in S joins to a tuple in T_{new}, it is hence unknown whether an update or an insert has to be propagated to the

S	T_{new}	S	T_{old}	result
un	-	uo	-	-
un	-	uo	pre	del
un	-	uo	uo	del
un	-	uo	del	del
un	-	uo	pup	tpdel
un	-	uo	ups	spdel
un	-	uo	tpdel	tpdel
un	-	uo	spdel	spdel
un	pre	uo	-	ins
un	pre	uo	pre	un/uo
un	pre	uo	uo	un/uo
un	pre	uo	del	un/uo
un	pre	uo	pup	pup
un	pre	uo	ups	ups
un	pre	uo	tpdel	pup
un	pre	uo	spdel	ups
un	ins	uo	-	ins
un	ins	uo	pre	un/uo
un	ins	uo	uo	un/uo
un	ins	uo	del	un/uo
un	ins	uo	pup	pup
un	ins	uo	ups	ups
un	ins	uo	tpdel	pup
un	ins	uo	spdel	ups

S	T_{new}	S	T_{old}	result
un	un	uo	-	ins
un	un	uo	pre	un/uo
un	un	uo	uo	un/uo
un	un	uo	del	un/uo
un	un	uo	pup	pup
un	un	uo	ups	ups
un	un	uo	tpdel	pup
un	un	uo	spdel	ups
un	pup	uo	-	ins
un	pup	uo	pre	un/uo
un	pup	uo	uo	un/uo
un	pup	uo	del	un/uo
un	pup	uo	pup	pup
un	pup	uo	ups	ups
un	pup	uo	tpdel	pup
un	pup	uo	spdel	ups
un	ups	uo	-	ins
un	ups	uo	pre	un/uo
un	ups	uo	uo	un/uo
un	ups	uo	del	un/uo
un	ups	uo	pup	pup
un	ups	uo	ups	ups
un	ups	uo	tpdel	pup
un	ups	uo	spdel	ups

Figure 5.8: Partial delta type join matrix (continued)

view. Hence, upserts in S are again propagated as upserts.

The matrix in Figure 5.8 represents joins involving update pairs in S. Recall that the new state of an updated tuple is joined to the new state of T (T_{new}) while the old state of an updated tuple is joined to the old state of T (T_{old}) in the delta rules for change propagation. The matrix shows all possible join combinations. Let s be an update pair in $S_{un/uo}$, s_{un} the new state of s, and s_{uo} the old state of s. The first two columns of the matrix indicate where s_{un} finds a join partner in T_{new}. There are the following possibilities: A join partner may not exists, it may be a preserved tuple, an inserted tuple, an updated tuple in its new state, a partial update, or an upsert.

The third and fourth column indicate where S_{uo} finds a join partner in T_{old}. A join partner may not exists, it may be a preserved tuple, an updated tuple in its old state, a deleted tuple, a partial update, an upsert, a true partial deletion, or a safe partial deletion. The fifth column indicates the type of the delta resulting from the joins. As an example, consider the second row of the matrix. It treats the case where s_{un} does not find a join partner in T_{new}, while s_{uo} used to join to a preserved tuple (i.e. the join attribute of s was updated). Hence, a tuple $S_{uo} \bowtie T_o$ used to be in the view and needs to be discarded now. Thus, the resulting delta is of type deletion.

The matrix in Figure 5.8 reveals a pattern. Whenever s_{uo} joins to a complete

tuple in T_{old}, namely a preserved tuple, an updated tuple, or a deleted tuple, the resulting delta tuple is again complete, i.e. an update pair or a deletion. When s_{uo} joins to a value-partial delta in T_{old}, i.e. a partial update or a partial deletion, the resulting delta tuple is again value-partial. Whether the resulting delta tuple is a partial update or a partial deletion is decided based on the existence of a corresponding delta tuple in the insert set $\Delta(S \bowtie T)$, when the deltas are converted to the explicit model (see Section 5.4.3). When s_{uo} joins to a type-partial delta in T_{old}, i.e. an upsert or a safe partial deletion, the resulting delta tuple is type-partial again. Based on these considerations and the considerations that lead to the first join matrix, the join delta rules are adapted as follows to handle partial deltas.

$$\begin{aligned}\Delta(S \bowtie T) &\equiv (S_o \bowtie \Delta T) \cup (\Delta S \bowtie T_o) \cup (\Delta S \bowtie \Delta T)\\ \nabla(S \bowtie T) &\equiv \pi_{\ldots,T.flag}((S_o \bowtie \nabla T) \cup (\nabla S \bowtie T_o) \cup (\nabla S \bowtie \nabla T)) \cup\\ &\qquad \pi_{\ldots,s(flag)}(\sigma_{flag \neq \mathsf{comp}}(\nabla S))\end{aligned}$$

Once again, the delta rule for computing the insert delta set remains unchanged. The delta rule for the delete delta set is changed in the following way. Note that the initial three joins are evaluated for complete S tuples only, because partial S tuples have null-padded attribute values. As shown above, in this case the type of the resulting tuples are determined by the type of the joining T tuples. Hence the type flag is reused in the result. An additional term is added to the rule to handle partial tuples in ∇S that lack the attribute values required to compute the join with T_{old}. In this additional term the function s (defined in Section 5.4.5) is reused to modify the type flag appropriately.

The delta rules implement the delta type transitions suggested by the matrices in Figure 5.7 and 5.8. Note that any combination of delta types yields a delta type that can be expressed in the partial delta model proposed in Section 5.4.1. That is, partial deltas are closed under join operations permissible in dimension view definitions.

Example 5.4 *Reconsider the running example. The sample dimension view was defined as $\mathcal{D} = \mathcal{C}' \bowtie \mathcal{A}'$. The incremental expressions $\Delta\mathcal{D}$ and $\nabla\mathcal{D}$ can be derived using the delta rules given above.*

$$\begin{aligned}\Delta\mathcal{D} &\equiv (C'_o \bowtie \Delta A') \cup (\Delta C' \bowtie A'_o) \cup (\Delta C' \bowtie \Delta A')\\ \nabla\mathcal{D} &\equiv \pi_{CID,CName,AID,ACity,ACountry,A'.flag}(C'_o \bowtie \nabla A') \cup\\ &\qquad \pi_{CID,CName,AID,ACity,ACountry,A'.flag}(\nabla C' \bowtie A'_o) \cup\\ &\qquad \pi_{CID,CName,AID,ACity,ACountry,A'.flag}(\nabla C' \bowtie \nabla A') \cup\\ &\qquad \pi_{CID,CName,AID,NULL,NULL,s(flag)}(\sigma_{flag \neq \mathsf{comp}}(\nabla C'))\end{aligned}$$

ΔD

CID	CName	AID	ACity	ACountry
1	Adam	1	Essen	DE
2	Bob	4	Dresden	DE
4	Dave	4	Dresden	DE

∇D

CID	CName	AID	ACity	ACountry	flag
1	Adam	1	-	-	ptype
2	Bob	2	Berlin	DE	comp
3	Carl	3	-	-	ptype

Table 5.4: Result of the sample incremental expressions ΔD and ∇D

Note that the preserved relation states referenced above are computed as follows.

$$\begin{aligned} C'_o &= C'_{new} - \Delta C' = \pi_{attr_C}(C_{new}) - (\pi_{attr_C}(\Delta C) - \pi_{attr_C}(\nabla C)) \\ &\quad with\ attr_C = \{CID, CName, AID\} \\ A'_o &= A'_{new} - \Delta A' = \sigma_p(A_{new}) - \sigma_p(\Delta A) \end{aligned}$$

Alternatively, these relations may be derived from the preserved states of the base relations C_o and A_o instead of the current states C_{new} and A_{new}, whatever is more appropriate.

$$\begin{aligned} C'_o &= \pi_{attr_C}(C_o \cup \Delta C) - (\pi_{attr_C}(\Delta C) - \pi_{attr_C}(\nabla C)) \\ &= \pi_{attr_C}(C_o) \cup (\pi_{attr_C}(\Delta C) \cap \pi_{attr_C}(\nabla C)) \\ A'_o &= \sigma_p(A_{new} - \Delta A) = \sigma_p(A_o) \end{aligned}$$

The result deltas are depicted in Figure 5.4. When ΔD and ∇D are converted back to the explicit model, we obtain an insertion (ID 4), an upsert (ID 1), an update pair (ID 2), and a partial deletion (ID 3).

5.4.7 Set operators and aggregation

We deliberately restricted dimension views to relational selection, projection, and join to achieve maintainability w.r.t. partial deltas. In this section we will discuss the remaining relational set operators and aggregation. To compute difference and intersection views, all attribute values of the input tuples need to be considered. Note that this is in contrast to join views. For this reason, we do not see a way to incrementally maintain difference or intersection views based on partial deltas, because attribute values may be unknown.

The relational union operator performs an implicit duplicate elimination. Again, all attribute values of the input tuples need to be considered to detect

duplicates. Thus, in the general case union views cannot be maintained based on partial deltas. However, if the existence of duplicates can be ruled out, e.g. by adding unique source identifiers to tuples, union views can be maintained based on partial deltas. The delta rules for this case are straightforward.

$$\Delta(S \cup T) \equiv \Delta S \cup \Delta T \qquad \nabla(S \cup T) \equiv \nabla S \cup \nabla T$$

Incremental maintenance of aggregate views is usually done by aggregating insert and delete deltas to understand the effect on the materialized view and updating it accordingly. However, value-partial deltas lack attribute values and thus partial deltas cannot be aggregated. For this reason, we do not see a way to maintain aggregate views using partial deltas. To work around this limitation, the input data to be aggregated may be materialized at the warehouse as a dimension view and incrementally maintained.

5.4.8 Putting it all together

In the previous sections, it has been shown that partial deltas are closed under projection, selection, and join (with equality predicates involving at least one relation's primary key). Recall that dimension views are assembled from exactly these operations. To conclude that dimension views are maintainable using partial deltas, we need to show that the propagated deltas can be applied to the view.

In Section 5.4.2 we have shown that the presence of a primary key attribute is a necessary condition for partial deltas to be applied to a view. However, dimension views may not contain all primary key attributes of the base relations. In fact, dimension views may not include any primary key attributes of the source relations that appear in a join predicate. These primary key attributes, however, are functionally dependent on the primary key attribute of the second relation participating in the join. View tuples can hence be uniquely identified by means of the non-functionally dependent primary key attributes, which must be included in dimension views according to Definition 5.2. Thus partial deltas can always be applied to dimension views. Given that and the closure property of partial deltas discussed earlier, we conclude that dimension views are maintainable using partial source deltas.

5.5 Chapter summary

Previous work on view maintenance, including approaches to view maintenance in a distributed environment, tacitly assumed that deltas are captured completely at the source systems. We analyzed existing CDC techniques and discovered that this assumption does often not hold in practice. In fact, we found

several CDC techniques to be incapable of providing complete deltas. Since there are some CDC techniques that do not have this restriction, one could argue that only these techniques should be used for warehouse maintenance. However, partial deltas may be captured more efficiently. Furthermore, source systems are typically autonomous and the system owners are often reluctant to changes, thereby limiting the choice of practicable CDC techniques. We discussed properties of integration systems before (Section 2.2.3) and found it desirable to allow for a large degree of source autonomy. That is, an ideal integration system should make the most of what the sources are able and willing to offer, including partial deltas.

In this chapter we studied view maintenance using partial deltas. At first, we introduced a formal model for partial deltas. As we have shown, views cannot be maintained using partial deltas in general, but there is a class of view that is maintainable. We referred to this class as dimension views, because of their close relation to dimension tables typically used in warehouse star schemas. Based on our formal model for partial deltas, we then proposed a generalization of the algebraic view maintenance approach. To our knowledge, our algorithm is the first that allows for maintaining (a class of) views using partial deltas.

6 Preventing incremental maintenance anomalies

Data warehouses are traditionally maintained in a periodic manner, most often on a daily basis. Thus, there is some delay between a business transaction and its appearance in the data warehouse. The most recent data is trapped in the operational sources where it is unavailable for analysis. For timely decision making, today's business users ask for ever fresher data.

In this thesis we propose to address this challenge by incremental ETL processing. That way, the data warehouse maintenance intervals may be shortened and source deltas may hence be propagated to the warehouse with lower latency. This approach is sometimes referred to as near-real time data warehousing or micro-batch ETL [KC04]. One consequence of this approach is that warehouse maintenance can no longer be performed in off-peak hours such as nightly batch windows. For this reason, the source data may change while maintenance is in progress. We will show that anomalies may arise under these circumstances leading to an inconsistent warehouse state and we propose several approaches to avoid such maintenance anomalies.

This chapter is based on our work published in [JD09b]. In Section 6.1 we will discuss the origin of maintenance anomalies and distinguish between different anomaly types. In Section 6.2 we will propose two principal approaches to prevent maintenance anomalies that will be further discussed in Sections 6.2.2 and 6.2.3. We will conclude the chapter in Section 6.3.

6.1 Maintenance anomalies

As discussed in Section 2.3.4, so-called distributed incremental view maintenance anomalies (or maintenance anomalies for short) were first recognized in [ZGMHW95] and stimulated further research in this area. Maintenance anomalies may occur when view maintenance is performed outside a transaction scope. Several compensating algorithms have been proposed to avoid maintenance anomalies. Refer to Section 2.3.4 for a discussion of the ECA, Strobe, and SWEEP compensating algorithms.

Recall that these algorithms make strong assumptions about the data sources. The sources are required to actively notify the warehouse about changes and must be able to evaluate compensation queries. In this chapter, we consider a

range of sources with different capabilities.

The compensating algorithms themselves are rather complex. It is necessary to track unanswered queries sent to the sources, detect source changes that occurred concurrently to query evaluation, construct compensating queries, or perform local compensation of previous query results. In particular, the algorithms are designed for a message-oriented data exchange.

State-of-the-art ETL tools, however, allow for the implementation and execution of rather simple data flows only. The underlying model is typically a directed, acyclic graph whose edges indicate the flow of data and whose nodes represent transformation operators provided by the ETL tool. Furthermore, ETL tools are not built for message-oriented data exchange but rather for processing data in larger batches. Therefore, we do not see a way to implement ECA, Strobe, or SWEEP using a state-of-the-art ETL tool. Future real-time ETL tools may well offer such advanced features, if there will be a convergence between ETL and EAI technologies. However, for the time being other approaches need to be considered to prevent maintenance anomalies in incremental ETL processing. In this chapter we propose several approaches that can be realized with state-of-the-art ETL tools.

Let us illustrate potential incremental maintenance anomalies through an example. Consider a simplistic warehouse product dimension table D with only three columns storing the business key, the name, and the category of products. Furthermore consider two source tables P and C storing product data and product category data, respectively. The product dimension D is computed as $\mathcal{D} := \pi_{pid,pname,cname}(P \bowtie C)$ as depicted in Figure 6.1.a. Through algebraic differencing using the delta rules proposed in Section 4.2, the following incremental maintenance expressions are derived from $\mathcal{D}$.

$$\Delta\mathcal{D} = \pi_{pid,pname,cname}(P_o \bowtie \Delta C \cup \Delta P \bowtie C_o \cup \Delta P \bowtie \Delta C)$$
$$\nabla\mathcal{D} = \pi_{pid,pname,cname}(P_o \bowtie \nabla C \cup \nabla P \bowtie C_o \cup \nabla P \bowtie \nabla C)$$

In the following, we will provide examples to illustrate different types of maintenance anomalies.

Maintenance anomaly related to deletions As shown in Figure 6.1.a, the initial source relation states are $P = \{[1, \mathsf{apple}, 1]\}$ and $C = \{[1, \mathsf{groceries}]\}$. Hence the product dimension is initially computed as $D = \{[1, \mathsf{apple}, \mathsf{groceries}]\}$. Suppose that the tuples $[1, \mathsf{apple}, 1]$ and $[1, \mathsf{groceries}]$ are deleted from P and C, respectively. That is, P and C are empty in their current states $P_{new} = \varnothing$ and $C_{new} = \varnothing$.

For reasons we will discuss in detail in the subsequent sections, there may be some delay between the point in time at which base relations are updated and the point in time at which changes are captured and visible in the corresponding

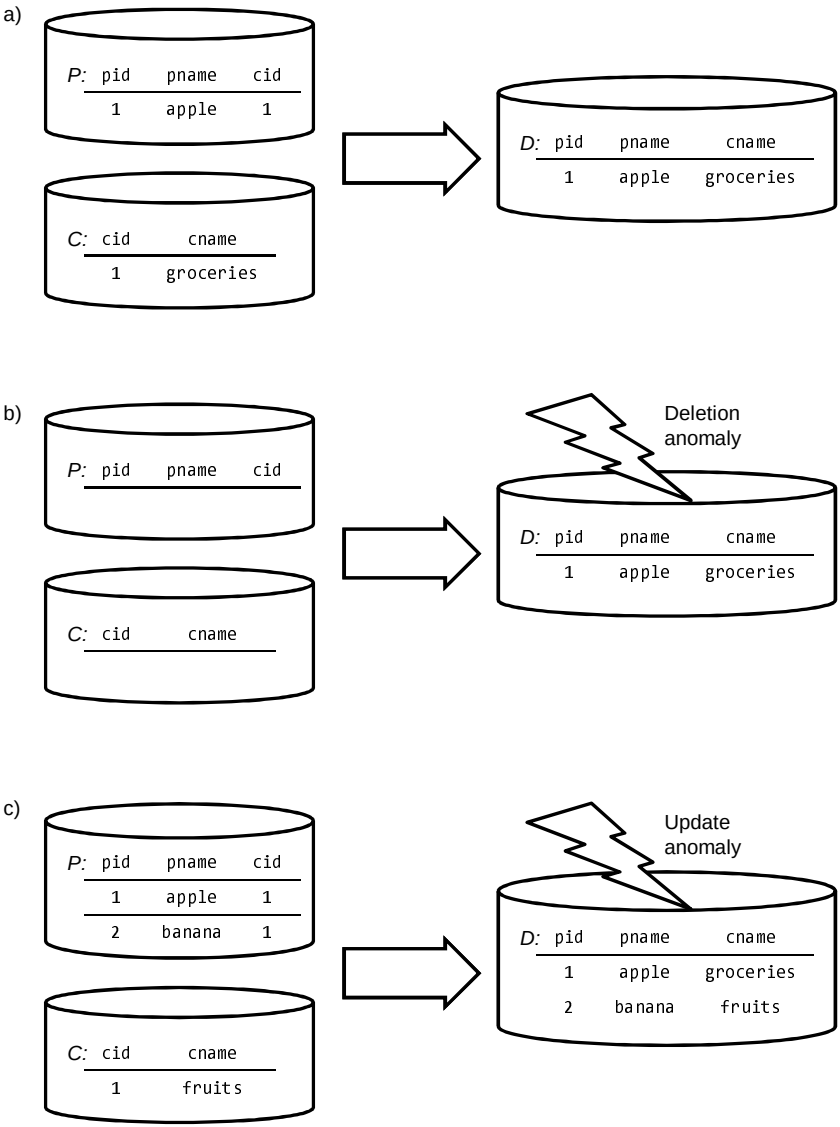


Figure 6.1: Sample maintenance anomalies

delta relations. Therefore, the ETL system may already see the first deletion $\nabla P = \{[1, \mathsf{apple}, 1]\}$ but it may not see the second deletion yet, i.e. $\nabla C = \varnothing$. When the ETL job $\nabla\mathcal{D}$ is executed it returns an empty set $\nabla D = \varnothing$. Note that $P_o = P_{new} - \Delta P = \varnothing$ and $C_o = C_{new} - \Delta C = \varnothing$. Hence, each of the three joins in $\nabla\mathcal{D}$ returns an empty result. Thus $\nabla\mathcal{D}$ evaluates to the empty set and the product dimension D remains unchanged.

At some later point in time, the second deletion will be captured and ∇C will turn to $\{[1, \mathsf{groceries}]\}$. However, because P_o and ∇P are empty at this point in time, the maintenance expression $\nabla\mathcal{D}$ will evaluate to the empty set again. The warehouse dimension thus remains $D = \{[1, \mathsf{apple}, \mathsf{groceries}]\}$ as shown in Figure 6.1.b. This is obviously incorrect and we speak of a *deletion anomaly*. Deletion anomalies arise when base tables are affected by deletions that have not been captured by the time incremental maintenance is performed.

Maintenance anomaly related to updates Again, suppose the initial base relation states are $P = \{[1, \mathsf{apple}, 1]\}$ and $C = \{[1, \mathsf{groceries}]\}$. Now suppose that product categories are reorganized and the tuple $[1, \mathsf{groceries}]$ is updated to $[1, \mathsf{fruits}]$. The current state of C is hence $C_{new} = \{[1, \mathsf{fruits}]\}$. Say, a new product tuple $[2, \mathsf{banana}, 1]$ is inserted and $P_{new} = \{[1, \mathsf{apple}, 1], [2, \mathsf{banana}, 1]\}$. At some point in time the change in P is captured and available in $\Delta P = \{[2, \mathsf{banana}, 1]\}$.

However, the change capture at C may be delayed and both, ΔC and ∇C are empty at this point in time. In this situation, the maintenance jobs $\Delta\mathcal{D}$ and $\nabla\mathcal{D}$ will evaluate to $\Delta D = \{[1, \mathsf{banana}, \mathsf{fruits}]\}$ and $\nabla D = \varnothing$, respectively. Hence, the state of the dimension table after maintenance is $D = \{[1, \mathsf{apple}, \mathsf{groceries}], [2, \mathsf{banana}, \mathsf{fruits}]\}$ as shown in Figure 6.1.c. However, both products are in the same category at the source system. Thus, the product dimension D is inconsistent after being maintained and we speak of an *update anomaly*. If a warehouse user issues a business intelligence query to aggregate figures by category, she will receive a confusing result containing categories that did not exist at the same time at the source systems.

At some later point in time, the update at C will be captured and the delta relations will turn to $\Delta C = [1, \mathsf{fruits}]$ and $\nabla C = [1, \mathsf{groceries}]$. The maintenance jobs $\Delta\mathcal{D}$ and $\nabla\mathcal{D}$ will evaluate to $\Delta D = \{[1, \mathsf{apple}, \mathsf{fruits}], [2, \mathsf{banana}, \mathsf{fruits}]\}$ and $\nabla D = \{[1, \mathsf{apple}, \mathsf{groceries}], [2, \mathsf{banana}, \mathsf{groceries}]\}$, respectively. The product dimension will hence be $D = \{[1, \mathsf{apple}, \mathsf{fruits}], [2, \mathsf{banana}, \mathsf{fruits}]\}$ after incremental maintenance. Note that this state is consistent with the source states again.

Update anomalies arise when base tables are affected by updates that have not been captured at the time of an incremental recomputation. The resulting inconsistencies are a temporary issue. Given that no other updates occur, the inconsistencies are resolved in the subsequent maintenance cycle. Note that

this is not the case for inconsistencies arising from deletion anomalies.

After having seen examples of deletion and update anomalies, one may ask if there are insertion anomalies as well. In the strict sense, insertion anomalies do exist. They arise from insertions that affected the base table but have not been captured at the time an incremental recomputation is performed. Insertion anomalies cause the same tuple to be sent to the data warehouse multiple times. If keys exist in the target relation, however, the data warehouse does not become inconsistent. Therefore anomalies caused by insertions may not be regarded as actual anomalies.

We conclude this section with two remarks on maintenance anomalies. First, we emphasize that maintenance anomalies are not a problem particular to the join operator, which was presented in the examples above. In fact, maintenance anomalies are caused by the need to fetch base data during an incremental recomputation. Besides join, Cartesian product, (duplicate eliminating) union, difference, intersection, and aggregation may require access to the base relations for their results to be incrementally recomputed. The approaches to avoid maintenance anomalies discussed in the remainder of this paper address these relational operators, too.

As a second remark, we note that aspects of dimensional modeling (see Section 2.5.1) were ignored in the above examples to keep them as simple as possible. Typically, dimension data is not updated or physically deleted but a history of data is kept. However, maintenance anomalies are a fundamental problem that may arise whenever derived relations are incrementally recomputed. Though data is usually not physically deleted from warehouse dimension tables, it is often logically deleted by setting a flag or an expiration timestamp. Deletion anomalies cause dimension tuples to live on when they should have been expired.

The dimensional modeling methodology includes the Slowly Changing Dimension (SCD) technique to keep a history of data updates. Using SCD type 1, dimension tuples are updated in place; using SCD type 2, expired dimension tuples are kept and superseded with a newer version instead. Hence, the examples above apply to SCD type 1. In this setup, update anomalies are a temporary issue and resolve once all changes are captured. However, in a SCD type 2 scenario, updates are timestamped to capture the data history. This makes update anomalies not only a temporary issue, but causes inconsistencies that affect a part of the warehouse history permanently.

6.2 Preventing maintenance anomalies

In the previous section we have shown that maintenance anomalies cause the data warehouse to become inconsistent. Any analysis based on inconsistent

data will likely lead to wrong decisions being made, thus an inconsistent data warehouse is of no use. In this section we discuss approaches to prevent maintenance anomalies and keep the data warehouse consistent. Note that maintenance anomalies occur for two reasons.

- The ETL system sees base tables in a changed state but it does not see the corresponding deltas. Thus, there is an inconsistency between base tables and their delta sets. Such a *delta inconsistency* may occur for two reasons. First, several CDC techniques incur some latency between the original change in the base relation and the change being captured. Second, even in case changes are captured as part of the original transaction, the ETL system may still see an inconsistency. ETL jobs for incremental recomputing often evaluate joins between base relations and change data in a nested loop fashion. That is, the change data is first extracted and then used in the outer loop. Subsequently, the operational source is queried for matching tuples. When the base relation is not locked, it may be changed in the meantime and the ETL system effectively sees an inconsistency between the extracted deltas and the current base relation.

- The ETL jobs for incremental recomputing presented in Section 6.1 are based on traditional change propagation principles. In particular, an inconsistency between the base relations and its deltas is not anticipated.

Considering these two reasons for maintenance anomalies, there are two basic approaches to prevent them: Either ETL jobs can be prevented from seeing an inconsistency between a base relation and its deltas or ETL jobs can be redesigned to work correctly in spite of inconsistent deltas. We will discuss both options in the remainder of this section. Prior to this, we will discuss properties of operational source systems w.r.t. data access and change data capture, which are relevant for preventing maintenance anomalies.

6.2.1 Properties of operational data sources

Operational data sources differ in the way changes are captured and data is accessed. We gave a survey of change data capture (CDC) techniques in Section 5.1. Recall that we classified CDC techniques into four categories, namely audit columns, log-based approaches, change tracking, and snapshot differentials. Our survey was focused on how complete deltas are captured by different techniques. In the following, we will focus on the latency with which changes are captured, which is an important property in the context of maintenance anomalies.

Using audit columns or change tracking, changes are captured as part of the original transaction. Hence, these techniques capture changes without latency

and deltas are never inconsistent with base data. For log-based approaches this depends on the implementation approach. We will refer to CDC techniques as *synchronous* if changes are captured as part of the original transaction and as *asynchronous* if not.

Log-based CDC may be supported by active database capabilities such as triggers. In this way, logging may be done synchronously as part of the original transaction. Alternatively, triggers can be specified to be deferred causing deltas to be logged asynchronously in a separate transaction.

Log-based CDC may also be implemented by means of application logic. In this case, the application program that updates the back-end database is responsible for writing the respective deltas to a log table. Again, logging can be performed either synchronously as part of the original transaction or asynchronously in a separate transaction.

Database log scraping or log sniffing are two more popular implementation approaches for log-based CDC [KC04]. The idea is to exploit the transaction logs kept by the database system for backup and recovery. Log scraping means to parse archive log files. Log sniffing, in contrast, polls the active log file and captures changes on the fly. While these techniques have little impact on the source database, they involve some latency between the original transaction and the changes being captured, i.e. deltas are captures asynchronously. Obviously, the latency is higher for the log scraping approach.

The snapshot differential approach is typically chosen for unsophisticated source systems without a built-in CDC mechanism. It works by extracting complete data dumps referred to as snapshots at a regular basis and computing deltas through a comparison of successive snapshots. Since snapshots are stable once extracted, the computed deltas are guaranteed to be consistent with the snapshots.

Besides CDC techniques there are other properties of operational sources that are relevant for our discussion of maintenance anomalies. Sources may offer locking mechanisms to prevent data from being modified such as database table locks or file locks, for instance. We will refer to such sources as *lockable*.

Sources may also be able of providing stable read-only snapshots without locking. Such systems include so-called temporal databases that allow to explicitly query the database in any past state [Myr05]. Another type of systems providing point-in-time consistent snapshots are databases with a multiversion concurrency control scheme [RG03]. As the name suggests, such systems manage several data versions. This avoids managing locks for read transactions, because concurrent write transactions may be isolated by creating new data versions. Unlike temporal features, multiversion concurrency control schemes are widely used in open-source and commercial database system such as PostgreSQL, Microsoft SQL Server, or Oracle. We will refer to systems providing point-in-time consistent snapshots as *snapshot* sources.

6.2.2 Preventing delta inconsistencies

As suggested before, one way of avoiding maintenance anomalies is to ensure that the ETL system does not see inconsistent deltas. This can be achieved in several ways depending on the properties of the operational sources. We will describe several approaches and discuss pros and cons in the following.

Staging base data dumps When the snapshot differentials techniques are used for change capture, the source content is completely dumped into the staging area on a regular basis. Deltas are computed through a comparison of successive data dumps. Instead of reading base data from the sources, incremental ETL jobs may read from the staged data dumps. Because these dumps are stable, deltas are always consistent and thus maintenance anomalies are avoided. However, staging base data dumps has severe drawbacks. Frequently extracting complete data dumps poses a significant overhead. Computing snapshot differentials is computationally expensive and does not scale well with increasing problem amounts of data. Furthermore the storage requirements at the staging area are at least two times the size of the base data, because at least one prior data dumps needs to be kept for future use.

Locking source systems Another way to avoid inconsistencies between base data and deltas and thus avoid maintenance anomalies is to ensure that base data is not changed during incremental maintenance. This is feasible for lockable source systems using audit columns, change tracking, or log-based change capture techniques. Special care must be taken if deltas are logged asynchronously. Then there is some latency between the original change and the corresponding log entry. Thus, simply locking the base table cannot avoid delta inconsistencies, because changes that occurred before the lock was placed may not have been written to the log yet. If there is no mechanism to "flush" the change log after the base relations have been locked, this approach cannot avoid maintenance anomalies in the general case. The drawback of locking operational sources is obvious. For the duration of incremental maintenance, all writing transactions at the sources are blocked. This may not be acceptable apart from off-peak hours.

Maintaining staged base data copies Another way to avoid deltas inconsistencies is maintaining copies of base relations at the staging area. At each incremental maintenance cycle the ETL system retrieves the sources deltas that are used in two ways. First, the deltas serve as the input for the incremental ETL jobs. Second, the deltas are used to maintain local copies of base relations. Local copies may either be maintained before or after incremental

ETL processing. In the former case the staged copies match the initial state of the base relations, in the later case the current state.

Staging copies of base relations avoids maintenance anomalies for both synchronous and asynchronous CDC techniques. Because asynchronous techniques capture deltas with some latency, the staged copies may lag behind the base relations. However, the staged copies are always consistent with the captured deltas.

Note that it may not be required to stage copies of entire base relations. The base relations may contain attributes that are not included in the data warehouse schema. Furthermore, only source tuples satisfying given predicates may be relevant to the data warehouse. To save storage space, copies of base relations can be restricted to relevant attributes and tuples. The staged copies are essentially Select-Project (SP) views in the sense of [GJM96]. It has been shown that SP views are always self-maintainable with respect to insertions. A sufficient condition for self-maintainability of SP views with regard to deletions is to retain the key attributes in the view. Therefore, staged copies should include the key attributes of their base relation.

Maintaining staged copies of base data has advantages compared to the approaches discussed so far. The impact on the operational sources is small; no data apart from deltas is extracted and concurrent source transactions are not blocked. An obvious disadvantage is the considerable storage overhead at the staging area.

Virtual base data snapshots As suggested before, inconsistencies between base data and deltas can be avoided by keeping base data stable while incremental maintenance is in progress. One way of doing this is to lock base datasets. Another, probably more attractive approach is offered by snapshot sources. As discussed in Section 6.2.1, snapshot sources are able to efficiently provide point-in-time consistent data snapshots. These snapshots are virtual in the sense that they need not be physically materialized. This is usually achieved through a multiversion concurrency control scheme. Instead of accessing live base data, incremental ETL jobs may access virtual snapshots and thus prevent maintenance anomalies. Note that similar to the locking approach, delta logs need to be flushed if an asynchronous CDC technique is used.

The virtual base data snapshot approach is attractive because it incurs no storage overhead and has little impact on the operational source system. In particular, concurrent source transactions are not blocked. The disadvantage is merely that this approach is only applicable to snapshot sources. Surprisingly, exploiting virtual base data snapshots has not been considered in work on distributed view maintenance (see Section 2.3.4) though it is a simple and efficient approach to avoid maintenance anomalies. However, exploiting multi-

version concurrency control schemes at source systems has also been proposed to solve synchronization issues in constraint-based database caching [Kle11].

Distributed transactions Incremental ETL processing may be performed in a distributed transaction in which the source systems act as resource managers. In this way, deltas and base data are read in the same transaction context and thus guaranteed to be consistent. Note that source data is never updated by the ETL system. The two-phase commit protocol performed for transaction completion is thus aborted early, because all sources vote "read-only" in the prepare phase. However, distributed transactions may still incur a considerable overhead. This is largely dependent on the concurrency control mechanism used at the source systems. As discussed earlier, multiversion concurrency control schemes are suitable, because reads are never blocked and concurrent write transactions are not interfered with. However, concurrency control schemes based on locking are problematic. Read locks are held for the duration of incremental recomputing and during this period, any write transactions at the sources are blocked, which is likely not acceptable.

6.2.3 Anomaly-Proof Incremental Recomputations

We identified two principal reasons for maintenance anomalies. First, anomalies may arise through deltas being inconsistent with base data. Second, the incremental ETL jobs are based on traditional incremental maintenance approaches. In this section we propose approaches that are "anomaly-proof", i.e. work correctly in spite of delta inconsistencies. In particular, we are interested in solutions that neither lock operational sources nor maintain data copies in the staging area.

All solutions discussed in the previous section guarantee that the data warehouse remains consistent with the sources. Intuitively, this means that incrementally recomputing always leads to the same data warehouse state as fully recomputing would do. Some approaches proposed in this section do not achieve this level of consistency. Depending on the data warehousing application, lower levels of consistency may be acceptable. Therefore we define a hierarchy of consistency levels based on [ZGMHW95] that allows us to classify the approaches proposed in the remainder of this section.

- *Convergence:* For each sequence of source changes and each sequence of incremental recomputations, after all changes have been captured and no other changes occurred in the meantime, a final incremental recomputation leads to the same data warehouse state as a full recomputation would do. However, the data warehouse may pass through intermediary states

that would not appear, if it was fully recomputed in each maintenance cycle.

- *Weak Consistency:* Convergence holds and for each data warehouse state reached after incrementally recomputing, there are valid source states such that fully recomputing led to this state of the data warehouse.

- *Strong Consistency:* For each sequence of source changes and each sequence of maintenance cycles, an incremental recomputation leads to the same data warehouse state as a full recomputation would do.

To satisfy the convergence property a data warehouse maintenance approach must avoid deletion anomalies. However, it may permit for update anomalies because they appear only temporarily and are resolved in subsequent maintenance cycles.

To satisfy the weak consistency property a maintenance approach must not allow for update anomalies. As shown in Section 6.1 an update anomaly may lead to a data warehouse state that does not correspond to any valid state of the sources. This is contradictory to the definition above. However, the weak consistency property holds as long as some valid source state can be found at all. In particular, we are free to postpone the propagation of updates to subsequent maintenance cycles.

The strong consistency property is most stringent and requires the warehouse to always enter the same state after maintenance that a full recomputation would produce. In particular, the propagation of updates must not be postponed to subsequent maintenance cycles. Note that all data warehouse maintenance approaches discussed in the previous section achieve strong consistency.

In the remainder of this section, we will propose three approaches to make incremental ETL jobs anomaly-proof. For each approach, we will discuss the advantages and drawbacks, the requirements w.r.t. the source systems, and the achieved level of consistency.

Compensating for delta inconsistencies Sources with synchronous, log-based CDC techniques may compensate for delta inconsistencies. Such sources capture changes as part of the original transaction. Delta inconsistencies may still occur, when the ETL system extracts deltas and queries base data in separate transactions. Reconsider the sample incremental ETL job presented in Section 6.1, $\Delta\mathcal{D} = \pi_{pid,pname,cname}(P_o \bowtie \Delta C \cup \Delta P \bowtie C_o \cup \Delta P \bowtie \Delta C)$. Since ΔC and ΔP are typically much smaller than P_o and C_o, the joins are usually evaluated in a nested loop fashion. That way, only matching tuples need to be extracted from the base relations.

When the incremental ETL job is started, it extracts the deltas that are used in the outer loop of a join. For each delta tuple the base relations are queried in separate transactions. Hence, base data changes that occur after the deltas are extracted and before the last query is answered, cause delta inconsistencies and may thus lead to maintenance anomalies. To avoid such inconsistencies, the sources may use information from the delta log to compensate for delta inconsistencies locally.

Say, the previous incremental recomputation was performed at time t_1 and the current incremental recomputation is started at time t_2. When the incremental ETL job is started, it extracts changes to C and P for the time interval from t_1 to t_2, denoted as $\Delta C[t_1, t_2]$ and $\Delta P[t_1, t_2]$, respectively. Hereafter, queries against the base relations are issued to evaluate the joins. The state of C and P may change at any time, thus query answers may contain unexpected tuples (inserted after t_2) or lack expected tuples (deleted after t_2). To avoid this, the change log can be used to compensate for changes that occurred after t_2 and reconstruct the state of the C and P at time t_2 as $C_{\mathsf{now}} - \Delta C[t_2, \mathsf{now}] \cup \nabla C[t_2, \mathsf{now}]$ and $P_{\mathsf{now}} - \Delta P[t_2, \mathsf{now}] \cup \nabla P[t_2, \mathsf{now}]$.

Compensating for delta inconsistencies is feasible if the source system meets several prerequisites. It must be capable of evaluating the compensation expression locally and in a single transaction. Furthermore, the source must be logged synchronously and it must be possible to browse the log instead of reading it in a destructive manner. If these prerequisites are met, the outlined approach avoids maintenance anomalies and achieves strong consistency.

Exploiting view self-maintainability For synchronously logged sources that do not meet these prerequisites, we do not see any possibility to achieve strong consistency unless delta inconsistencies can be prevented following one of the approaches outlined in Section 6.2.2. However, there is a way to achieve convergence. Recall that the convergence property precludes deletion anomalies while it allows for update anomalies. Thus, making deletion propagation anomaly-proof is sufficient to achieve convergence. Consider the sample ETL jobs for incremental recomputing presented in Section 6.1 again. To achieve convergence, we need to modify $\nabla\mathcal{D}$ in a way such that deletions are correctly propagated in spite of delta inconsistencies.

In [GJM96] it has been shown that a sufficient condition for SPJ views to be self-maintainable with respect to deletions is to retain all key attributes in the view. Thus, deletions can be propagated to the warehouse dimension D, using only the deltas and D itself, if D contains all key attributes of the base relations and the ETL transformation logic consists of selection, projection, and join operators only. In particular, querying base relations is not required to propagate deletions and hence, delta inconsistencies are not an issue.

Reconsider the deletion anomaly example presented in Section 6.1. The initial situation is $P_{old} = \{[1, \mathsf{apple}, 1]\}$, $C_{old} = \{[1, \mathsf{groceries}]\}$, $P_{new} = \varnothing$, $C_{new} = \varnothing$, $\nabla P = \{[1, \mathsf{apple}, 1]\}$, and $\nabla C = \varnothing$. Note that delta set ∇C is inconsistent with C_{new}, because the deletion has not been captured yet. Assume that the warehouse dimension D includes all base keys and is given by $D_{old} = \{[1, \mathsf{apple}, 1, \mathsf{groceries}]\}$. Note that D is self-maintainable w.r.t. deletions. In response to the deletion ∇P, D can be maintained by removing each tuple with a key value of $pid = 1$. Doing so, the tuple $[1, \mathsf{apple}, 1, \mathsf{groceries}]$ is removed from D. When the deletion $\nabla C = \{[1, \mathsf{groceries}]\}$ is captured, each tuple with a key value of $cid = 1$ is deleted form D, however, no such tuple is found. Finally D is empty, which is the correct result.

In summary, a warehouse relation may be incrementally maintained and convergence may be achieved if it is self-maintainable w.r.t. deletions. For this purpose, all base relation key attributes need to be included. However, this may be problematic if multiple sources with heterogeneous schemas are used and is generally discouraged in dimensional modeling.

Detecting dirty base data For source systems using audit columns to timestamp updates, there is a way to achieve weak consistency. The basic idea is to exploit such timestamps to detect "dirty" base data during incremental recomputations. To achieve weak consistency update anomalies must be ruled out. Recall that update anomalies occur when base data is updated after deltas have been extracted but before the incremental recomputation is completed.

Update anomalies can be avoided by exploiting timestamps to detect "dirty" base data. Say, the previous incremental recomputation was performed at time t_1 and the next incremental recomputation is about to start. If audit columns are used for capturing changes, deltas are extracted by querying for tuples with a modification timestamp greater than t_1. The biggest timestamp seen during the delta extraction determines the current time t_2. When the ETL system queries the base relations, the answers may include tuples that have been modified after t_2. These tuples are considered dirty. Whether a tuple is dirty or not can easily be detected by its timestamp. It is generally not possible to find out about the state of dirty tuples before t_2, because the audit column CDC technique captures partial deltas (see Section 5.1). However, ignoring dirty tuples already avoids update anomalies.

Reconsider the update anomaly example presented in Section 6.1. Assume that the base relations include audit columns, which are appended to their schemas. Initially, $P_{old} = \{[1, \mathsf{apple}, 1, 10]\}$, $C_{old} = \{[1, \mathsf{groceries}, 10]\}$, $P_{new} = \{[1, \mathsf{apple}, 1, 10], [2, \mathsf{banana}, 1, 11]\}$, and $t_1 = 10$. Change capture using audit columns retrieves $\Delta P = \{[2, \mathsf{banana}, 1, 11]\}$ and $t_2 = 11$. Suppose that product categories are reorganized and C is updated to $C_{new} = \{[1, \mathsf{fruits}, 12]\}$.

Note that there is an inconsistency between ΔC, ∇C and C_{new} because the update has not been captured. However, the maintenance jobs $\Delta\mathcal{D}$ and $\nabla\mathcal{D}$ ignore any tuple with a timestamp greater than t_2 and thus evaluate to $\Delta D = \nabla D = \varnothing$. In the next maintenance cycle, the update at C will be captured as $\Delta C = \{[1, \mathsf{fruits}, 12]\}$. The maintenance job $\Delta\mathcal{D}$ will evaluate to $\Delta D = \{[1, \mathsf{apple}, \mathsf{fruits}], [2, \mathsf{banana}, \mathsf{fruits}]\}$ and D will be updated accordingly. Note that D is consistent with the sources and did not pass through any inconsistent intermediate states.

Ignoring dirty tuples does not prevent any changes from being propagated. In fact, the propagation is just postponed. All dirty tuples carry a timestamp greater than t_2 and will thus be part of the delta sets in the subsequent maintenance cycle. However, deltas may be propagated with a delay and the outlined approach thus achieves just weak consistency.

6.3 Chapter summary

As previous work on distributed materialized view management has shown, anomalies may occur when materialized views are incrementally maintained over distributed sources. Maintenance anomalies may corrupt the view and must hence be avoided. Incremental ETL processing is, at a conceptual level, similar to incremental view maintenance. As we have shown in this chapter, maintenance anomalies may thus occur in incremental ETL recomputations as well. We argued that techniques proposed in the context of distributed view maintenance cannot be applied in the ETL environment, because these techniques do not fit into the data flow-oriented ETL processing model and make rather strong assumptions w.r.t. source systems.

In this chapter, we proposed ways to avoid maintenance anomalies in incremental ETL processing. We identified two fundamental approaches. First, the ETL system can be prevented from seeing inconsistencies between deltas and base relations that cause maintenance anomalies in the first place. Second, the way incremental recomputing is done may be changed such that it works correctly in spite of inconsistent deltas. We considered both options and proposed several approaches to avoid maintenance anomalies that can be implemented using state-of-the-art ETL tools. In doing so, we considered a broad range of source systems with different properties in terms of change capture and data access. For each proposed approach we discussed its impact on the operational sources, storage overhead, level of consistency, and prerequisites w.r.t. the operational source system.

7 Incremental recomputations in MapReduce

In this chapter, we will shift the focus from traditional data integration solutions supported by ETL middleware to a more recent data transformation paradigm called MapReduce that gained popularity with the advent of cloud computing. In short, MapReduce is a programming model and an associated implementation for data-intensive computations on clusters of commodity hardware. MapReduce is designed to scale up to massive datasets and thousands of cluster nodes.

The advent of MapReduce sparked a controversial debate in the database community. Among others, MapReduce has been sharply criticized by DeWitt and Stonebraker who perceived it as "a major step backwards" [DS08a, DS08b]. The MapReduce opponents argue that key lessons of modern database design are neglected, because MapReduce does not support schemas and does not offer a declarative query and transformation language. Worse than that, the MapReduce opponents feel that MapReduce lacks crucial data management features such as indexing, integrity constraints, views, transaction support, and query optimization.

The MapReduce advocates argue that MapReduce should not be compared to database systems since both serve different purposes [DS08a, DS08b]. In their point of view, MapReduce is not designed for query evaluation but rather for batch transformations of massive datasets and thus, features such as indexing are useless. Since MapReduce is frequently used for processing heterogeneous and semi-structured data, database-like schemas are perceived as cumbersome. Furthermore, the MapReduce advocates claim that MapReduce scales to larger data volumes and larger number of cluster nodes than any existing (parallel) database system.

The debate around MapReduce reminds us of an earlier one between advocates of the ETL and the ELT approach [Rau05]. Recall that ETL and ELT are acronyms for Extract-Transform-Load and Extract-Load-Transform, respectively. The difference between ETL and ELT tools is that the former rely on special purpose runtime systems while the latter load data into the target database and use database functionality to perform the transformations. In this regard, ETL and MapReduce take a similar approach. And there are further similarities. ETL systems are primarily built for batch processing. Systems such as DataStage, for instance, process data in parallel in cluster environments [Inf]. Just like MapReduce, these systems do not support transactions,

neither do they offer a declarative query language nor a query optimizer. ETL middleware is purpose-build for materialized data integration and these features may be considered as overhead in this regard.

We feel that the debate between MapReduce advocates and opponents suggests that MapReduce should be perceived as "cloud ETL tool" rather than as general purpose "cloud database system". This being said, we decided to explore incremental recomputation approaches in the MapReduce environment. Just like ETL, MapReduce has similarities with materialized views from a high level of abstraction. A materialized view is derived from one or more base tables in a way specified by a user-supplied view definition and persistently stored in the database. Similarly, a MapReduce job reads data from some distributed storage system, transforms it in a way specified by user-supplied Map and Reduce functions and writes the result back to persistent storage.

However, while there are techniques to incrementally maintain materialized views, i.e. to refresh the view in response to base data changes, no such technique exists in the world of MapReduce. As yet, MapReduce programs are typically re-executed to obtain up-to-date results after the base data has changed, i.e. the "materialized view" is recomputed from scratch. We found this situation to be very comparable to the state-of-the-art in ETL and realized that recurrent MapReduce computations may just as well benefit from view maintenance concepts. We proposed our initial ideas to Google and were granted a Google Research Award in December 2010. This chapter summarizes our work published in [JPYD11a, JPYD11b] and the results of two bachelor theses [Par11, Sch12].

In this chapter, we will show that MapReduce results can often be maintained more efficiently in an incremental manner. In Section 7.1, we will provide some preliminaries and briefly introduce the MapReduce paradigm and some related technologies. Related work will be discussed in Section 7.2. We propose a novel view maintenance-inspired approach to incremental recomputations in MapReduce and discuss the challenges posed by the specifics of the MapReduce environment in Section 7.3. In the form of a case study, we will first show how to incrementalize several typical MapReduce programs and finally work out the general case in Section 7.4. We will evaluate our approach in Section 7.5 and conclude in Section 7.6.

7.1 MapReduce and related technologies

The MapReduce system has been designed for data processing on large clusters of commodity servers. Apart from data processing, data storage systems have been build for this environment that act as data source and data sink for MapReduce jobs. We will briefly introduce MapReduce in Section 7.1.1 and two

Algorithm 1 Word count

```
1: function Map(url, document)
2:     for all word in document do
3:         emit(word, 1)
4:     end for
5: end function
6: function Combine(word, list of values)
7:     for all v in values do
8:         sum = sum + v
9:     end for
10:    emit(word, sum)
11: end function
12: function Reduce(word, list of values)
13:    for all v in values do
14:        sum = sum + v
15:    end for
16:    emit(word, sum)
17: end function
```

storage systems referred to as Google File System and Bigtable in Section 7.1.2 and 7.1.3, respectively.

7.1.1 MapReduce

MapReduce is a programming model and an associated implementation for processing large datasets in parallel on clusters of commodity hardware that was introduced by Google in 2004 [DG04, DG08]. The MapReduce programming model is based on two functional programming primitives known as Map and Reduce.

A MapReduce programs consists of a user-supplied Map and Reduce functions that are executed in sequence. We will use the terms Mapper and Reducer to mean instances of Map and Reduce functions. Mappers take key-value pairs as input and produce a set of intermediate key-value pairs. It is typical for intermediate keys to be derived from the original values and hence, differ from the original keys. The MapReduce framework groups together all intermediate values having the same intermediate key and passes them to the Reduce function. Reducers accept an intermediate key along with a group of values, aggregate these values, and typically produce zero or one output key-value pair per input group. The authors of [DG04] state that most computations performed at Google can naturally be expressed by (sequences of) MapReduce programs.

As an example, consider the problem of counting the number of occurrences of words in a large collection of documents. This computation is expressed by

Algorithm 2 Reverse web-link graph

```
1: function Map(url, document)
2:     for all link in document do
3:         emit(link.target, url)
4:     end for
5: end function
6: function Reduce(target, list of sources)
7:     inlinks = deduplicate(sources)
8:     emit(target, list of inlinks)
9: end function
```

Algorithm 1. The Map function accepts text documents as input values. For each word occurrence in a document, it emits an intermediate key-value pair with the word itself being used as intermediate key and the numeric value one being used as intermediate value.

Before the intermediate key-value pairs are fed into the Reduce function, they may optionally be passed to a so-called *Combine* function. Combiners can be thought of as local Reducers. They pre-aggregate the output of all Mappers executed on the local node and thus help to reduce the network communication. In Algorithm 1, the Combine function sums up the counts emitted for a particular word per node. The total sum is subsequently computed by the Reduce function.

As another slightly more complex example, consider the computation of reverse web-link graphs, which is considered another typical MapReduce use case in [DG04]. Reverse web-link graphs are computed from large collections of web documents. The hyperlinks in these documents constitute a web-link graph. As the name suggests, the reverse web-link graph is derived from the web-link graph by reversing the direction of the edges of the graph. The reverse web-link graph thus makes the incoming links of each web document explicit instead of its outgoing links and is useful for computing page ranks, for instance.

The pseudo-code for the computation of reverse web-link graphs is shown as Algorithm 2. For each link found in a web page, the Map function emits the link's target URL along with the URL of the source page as intermediate key and intermediate value, respectively. The Reduce function receives target URLs together with a list of associated source URLs. It concatenates all distinct source URLs and emits the target URL together with the concatenated source URLs as result key and result value, respectively. Note that computing reverse web-link graphs is straightforward in MapReduce, while it would be rather cumbersome using SQL. This is because the relational model does not support set-valued attributes.

The MapReduce run-time system executes Map and Reduce functions in parallel and takes care of the data partitioning, scheduling, handling machine

failures, and managing the required inter-machine communication. Google's MapReduce implementation is reported to scale to petabytes of data[1] and thousands of machines [DG08]. A popular open-source implementation of the MapReduce framework has been developed under the Apache Hadoop project [Apab].

7.1.2 Google file system

The Google File System (GFS) is a distributed file system designed for large clusters of commodity machines [GGL03]. A GFS cluster consists of a single master and multiple chunk servers. Files are divided into fixed-size chunks that are stored on the chunk servers' local disks. Chunks are replicated within the cluster for fault tolerance and availability. GFS provides a uniform access to clients, i.e. the physical distribution is transparent. The GFS master maintains file metadata such as the mapping from files to chunks and the locations of chunks. GFS is mainly designed for append-only files. While files can be appended in an atomic manner, random writes are supported only with weaker consistency guarantees.

A popular open-source implementation of GFS called Hadoop File System (HDFS) has been developed under the Apache Hadoop project [Apab]. While GFS files may be modified, HDFS files are immutable once written. MapReduce programs may read from and write to GFS (or HDFS) files. Typically the nodes in a cluster act as chunk servers and MapReduce workers (task tracker) at the same time. To exploit data locality and thus avoid network congestion, Map tasks are primarily spawned at nodes that store a copy of the input chunk locally.

7.1.3 Bigtable / HBase

Bigtable is a distributed storage system for managing structured data built on top of GFS [CDG+06]. An open-source implementation of Bigtable called HBase has been developed under the Apache Hadoop project [Apab]. A similar system called Accumulo has been developed by the U.S. Department of Defense. Accumulo has recently been donated to the open-source community and became an Apache Incubator project in 2011 [Apaa]. HBase and Accumulo are both built on top of HDFS. We chose to use HBase in our project, because Accumulo was not yet available and Bigtable is not open-source. We will hence speak of HBase in the following, but the things said most often apply to any Bigtable-like storage system.

[1] According to [DG08] more than twenty petabytes of data were being processed on Google's cluster per day in 2008.

The HBase data model is structured, but differs from the widely used relational model. An HBase table is essentially a multidimensional, sorted map. The map is indexed by a row key, a column key, and a timestamp. Row keys are arbitrary strings. HBase stores data in lexicographic order by row key. Tables are dynamically partitioned into so-called regions that cover the rows within a certain range of row keys. Regions are the unit of distribution and load balancing. Regions are dynamically assigned to so-called region servers that handle read and write requests and split regions that have grown too large.

The columns of HBase tables are organized into so-called column families. A column key that identifies an individual column consists of a reference to a column family and a so-called column qualifier. Column families must be declared upfront whereas qualifiers can be dynamically created within column families at runtime. HBase does not impose any structure on the data stored within columns, but treats it as uninterpreted string. However, clients usually serialize structured or semi-structured data into these strings. The column family layout directly influences the physical data layout on disk and thus allows clients to reason about locality properties.

HBase has built-in support for data versioning, i.e. each table column may contain multiple versions of the same data. Different versions are indexed by timestamps. HBase can be configured to keep a fixed number of versions or, alternatively, keep all versions that have been created within a sliding time window. To read data from HBase, specifying a version number is optional. If the version number is omitted, the latest version is retrieved by default.

The client API provided by HBase allows users to lookup data items by row key or iterate over a subset of rows in a table. Note that there is no notion of secondary indexes in HBase, i.e. the rows can only be efficiently retrieved by their row key. The API furthermore allows clients to create or drop tables and column families, and write or delete column values. In contrast to HDFS, HBase thus allows for fine-grained, incremental updates.

HBase supports single-row transactions, which can be used to atomically read, modify, and write back data stored within a single row. However, general transactions involving multiple rows are not supported. Furthermore, HBase does not support a query language as relational database systems do. However, HBase data can be used both as data source for MapReduce programs and thus, MapReduce can be seen as a query evaluator sitting on top of HBase.

7.2 Related work

While the MapReduce framework was proposed about seven years ago, the interest in incremental recomputation techniques in this context has emerged only recently. We are aware of several other efforts towards this common goal

that will be discussed in the following.

DryadInc A system called DryadInc has been developed at UC Berkeley and Microsoft Research [PBYI09]. DryadInc is an extension of the Dryad system [IBY+07], which can be thought of as Microsoft's version of MapReduce but offers a somewhat richer programming model. DryadInc executes Dryad programs more efficiently by reusing (intermediary) results computed in prior executions. This can be done in two ways. First, DryadInc caches intermediary results and reuses them if they occur unchanged in future computations. Second, using a user-supplied merging function, newly appended data is merged into a previously computed result.

The caching mechanism is transparent to Dryad users, i.e. Dryad programs do not need to be modified to take advantage of incremental recomputations. However, caching may incur a significant space overhead and the developers of DryadInc note that it is impractical to cache the intermediate results in their entirety. The system rather needs to choose a subset of intermediate results to be cached. Since the shape of future computations is not known a priori, the developers of DryadInc propose a heuristic selection approach.

DryadInc has been developed with programs for log analyzes in mind. For this reason, the underlying delta model of DryadInc is rather restricted. It is assumed that all base datasets are append-only structures, i.e. neither updates nor deletions are supported.

Incoop A system called Incoop has been developed at the Max Planck Institute for Software Systems in Saarbrücken, Germany [BWA+11, BWR+11]. Incoop extends the Apache Hadoop framework [Apab] in two ways. First, the HDFS is extended to allow for incremental file updates and equipped with a change capture mechanism. Second, a so-called memoization server is added to the Hadoop architecture that allows for caching intermediate results of MapReduce computations.

For Mappers, caching is performed at the task-level. The result of Map tasks is stored persistently beyond the life time of a MapReduce program and a reference is inserted to the memoization server. Future Map tasks may take advantage of the cached results if their input data happens to be unmodified. For Reducers caching is performed at both the task- and the subtask-level. For this purpose, the developers of Incoop propose a so-called Contraction phase that essentially divides larger Reduce tasks into smaller subtasks and thus allows for caching to be performed at a more fine-grained level. Incoop uses a so-called memoization-aware scheduler that considers the locality of cached results during task assignment to avoid unnecessary data movements.

The Incoop approach is transparent in the sense that it exposes the standard

Hadoop MapReduce API to application developers. Hence, existing MapReduce programs may be executed in an incremental manner without the need for any changes. However, the downside of caching is an increased disk space consumption. The developers of Incoop report that they observed a space overhead of up to a factor of nine during their experiments. This is quite significant considering that the datasets processed on MapReduce clusters are already very large.

Percolator Google build a system called Percolator for processing updates to large data sets incrementally [PD10]. It replaced the former MapReduce-based web-indexing system in 2010. The major design goal of Percolator was reducing the latency of indexing updated documents.

Percolator does not rely on the MapReduce infrastructure but is built on top of Google Bigtable and extends it with multi-row, distributed transactions. It furthermore provides an event-driven programming model based on so-called observers, which are somewhat similar to database triggers. An observer is a piece of application code that is invoked by the system whenever a user-specified column changes. Observers can be chained together, i.e. an observer that completed its task may triggers further observers by writing to the table column they are monitoring. Compared to MapReduce, which processes data in batch, Percolator processes individual updates with lower latency, but at the cost of additional computing resources.

Continuous bulk processing A system for continuous bulk processing (CBP) has been developed at UC San Diego and Yahoo Research [LOR+10]. The CBP approach is related to the data stream paradigm in the sense that input data is arriving continuously and output is continuously produced. CBP introduces a dedicated programming model for incremental processing, which offers primitives that store and reuse prior state. CBP programs are expressed as a graph of possibly stateful data transformation steps.

The CBP programming model is more complex than the rather simplistic MapReduce programming model. Each data transformation step is described by five user-defined functions that specify a data transformation to be performed, a grouping of input records into disjoint partitions, a framing of input records within a partition to be processed together, a criteria to determine the runnability of a transform step depending on its available input data, and optionally, a sorting of the output records. The concepts of framing and runnability allow the CBP system to determine what data to consider in an individual transformation step and to synchronize the consumption of data across multiple input flows.

The CBP system extends the Apache Hadoop framework [Apab]. Internally,

the CBP programming primitives are mapped onto the MapReduce programming model. The Hadoop framework has been extended to support all CBP primitives and efficiently handle stateful operators.

In summary, the existing approaches fall into two categories. DryadInc and Incoop are based on caching whereas Percolator and CBP make use of dedicated programming models for incremental processing. The main advantage of caching is its transparency in the sense that the MapReduce programming model and API remain unaffected. This allows for existing MapReduce programs to be processed in an incremental manner without changes. Moreover, MapReduce programmers are not required to develop incremental algorithms, which are typically far more complex than their non-incremental counterparts. The drawback of caching, however, is a considerable storage space overhead.

Dedicated programming models for incremental processing break transparency and force programmers to work at a lower level of abstraction. Programmers may, however, take advantage of problem-specific properties. A custom-built incremental program can be much more efficient than any transparent solution that is necessarily generic. In particular, excessive caching and the related storage space overhead can usually be avoided.

In this chapter, we will explore an alternative approach that is neither based on caching nor on dedicated incremental programming models, but inspired by techniques for incremental view maintenance. We propose to rewrite given MapReduce programs to derive incremental program variants that compute the changes in the prior result without doing redundant computations. This is quite analogous to view maintenance techniques that rewrite view definitions to derive incremental expressions that compute the changes in the materialized view.

We argue that a view maintenance-inspired approach may strike a balance between both alternative approaches, i.e. caching and dedicated programming models. Like caching, it is transparent to the user in the sense that incremental MapReduce programs are derived in an automated fashion. At the same time, excessive caching of intermediate results may be avoided. Depending on the MapReduce program it may be necessary to add some additional information to the result (the materialized view) to support incremental maintenance, but the storage overhead is generally very small.

7.3 Challenges of MapReduce view maintenance

At the beginning of this chapter we argued that, at a conceptual level, MapReduce programs can be seen as view definitions and their output data as materialized views. We will therefore use the terms MapReduce program and

MapReduce view interchangeably in the sequel. It thus seems desirable to maintain MapReduce views in an incremental way similar to materialized views in database systems to improve the efficiency of recurrent recomputations. However, from a more detailed point of view there are fundamental differences between traditional relational database systems and the MapReduce environment. Applying traditional database view maintenance techniques to MapReduce programs thus seems challenging, mainly for the following reasons.

- First, the programming models (or query languages) and data models differ heavily. View definitions are specified in SQL, a language closely tied to the relational algebra and the relational data model. The MapReduce programming model is more generic; the framework provides hooks to plug-in custom Map and Reduce functions written in standard programming languages.

  In the relational data model, attribute domains are restricted to atomic values. MapReduce computations, in contrast, often deal with set-valued attributes. Google's original paper on MapReduce mentions several typical use cases using set-valued attributes [DG04], which will be discussed in detail in Section 7.4.1. Expressing such computations in SQL would be rather cumbersome.

- Second, HBase is essentially a distributed hash table. Hence, data can only be efficiently located and retrieved by its primary key. Unlike in database systems, there is no notion of secondary indexes. The choice of efficient access paths is thus very limited. While it is common for MapReduce computations to scan through large amounts of data, incremental recomputations are much more dependent on efficient access paths. If scanning cannot be avoided, the efficiency gain will likely be very limited.

- Third, HBase supports single-row transactions only, i.e. single rows can be read and written atomically, but general transactions involving multiple rows are not supported. However, database view maintenance depends on transactional guarantees. First, transactions allow for synchronizing view maintenance and concurrent base data updates. Second, transactions allow for refreshing the materialized view such that clients cannot read any inconsistent intermediary state. Thus, standard view maintenance techniques cannot be applied in absence of transactions.

7.4 A case study

We did an initial case study to understand whether a view maintenance-like approach to incremental recomputations is feasible in MapReduce, despite the

concerns expressed in the previous section. For the case study, we selected a number of typical MapReduce programs, which will be introduced in Section 7.4.1.

We assume that these MapReduce jobs read from and write to HBase. In contrast to HDFS files that are immutable, HBase tables support row-level updates, which is essential for incremental view maintenance. Note that GFS does support file-appends and random writes and the techniques proposed in these sections are not tied to a specific storage system such as HBase. While HBase supports fine-grained updates, it does not provide any Change Data Capture (CDC) mechanism out of the box. We hence extended HBase with CDC capabilities that will be discussed in Section 7.4.2.

In Section 7.4.3, we will introduce a maintenance algorithm for aggregate views from the relational context referred to as Summary Delta Algorithm. We will show that this algorithm can be adapted and generalized using sample MapReduce views in Section 7.4.4 and extend our discussion to the general case in Section 7.4.5.

7.4.1 MapReduce use cases

In our case study, we initially considered five different MapReduce programs that are presented as typical use cases in Google's original paper on MapReduce [DG04]. Two of these use cases, namely computing word histograms and revers web-link graphs, have already been introduced in Section 7.1.1. However, we will revisit these use cases to explain how the output data is represented in the HBase data model, and introduce the remaining use cases in greater detail in the following.

Word histogram The MapReduce program for computing word histograms has been presented in Section 7.1.1. A word histogram is essentially a list of words, each with an occurrence count. There are multiple ways to represent word histograms in the HBase data model. However, it seems essential that the frequency of any given word can be retrieved efficiently. Hence, the word itself should act as row key and the occurrence count is stored in some fixed column family under some fixed qualifier.

Count of URL access frequency The MapReduce program processes logs of web page requests and computes the access frequency per URL. The computation is very similar to the computation of word histograms. The MapReduce program differs only in the Map function that needs to parse URLs from web logs instead of words from web pages. Similarly, the representation of the result data in the HBase data model resembles that of word histograms.

Reverse web-link graph The MapReduce program for computing reverse web-link graphs has been presented in Section 7.1.1. The resulting reverse web-link graph is represented as a list of target URLs, each with a set of associated source URLs. It seems natural to use the target URL as row key in the HBase data model. In this way, the referencing web pages can be efficiently retrieved for any given URL. The set of referencing web pages could be encoded as a single string using some separator character. However, is seems favorable to exploit HBase column qualifiers to impose more structure. Recall that column qualifiers can be dynamically created and dropped. Each referencing URL can thus be stored as a column qualifier within some fixed column family. Note that the actual column value can be left empty. Alternatively, it may be used to store an occurrence count for each target URL. We will elaborate more on this aspect in Section 7.4.4.

Term-vector per host A term vector summarizes the (most important) words that occur in a set of documents stored at the same host. Note that "term vector" is just another term for word histogram. That is, the MapReduce program essentially computes word histograms for multiple sets of documents at the same time. To do so, the Map function computes a term vector for each input document and emits ⟨hostname, term vector⟩ intermediate key-value pair. Note that the intermediate value is complex structured. The Reduce function consumes all per-document term vectors for a given host and adds them together.

For the representation of per-host term vectors in the HBase data model, it seems natural to use the host name as row key. Words within a given term vector may be stored as column qualifiers (within a fixed column family) and the per-word frequency count as column value.

Inverted index Inverted indexes are popular in information retrieval and full text searching and essentially provide a mapping from words to documents that contain those words. To compute inverted indexes using MapReduce, a Map function that parses each document and emits ⟨word, URL ⟩ intermediate key-value pairs is used. The Reduce function collects all URLs associated with a given word. Note that computing inverted indexes is quite similar to computing reverse web-link graphs and that the output data is equally structured. For this reason, it can be mapped into the HBase data model in much the same way, i.e. words act as row keys and URLs as column qualifiers.

Two more MapReduce use cases are mentioned in [DG04], namely distributed grep and distributed sort. We will not consider these use cases here, because distributed grep is an embarrassingly parallel problem and incremental main-

tenance is trivial. Distributed sort is not interesting when HBase is used at the storage layer, because HBase data is always stored in sort order.

All MapReduce use cases we consider process large collections of web-related documents. These documents will change over time (as the web is being crawled, for instance). Hence, the result of any computation will become more and more stale. To obtain recent results, the MapReduce program can simply be re-executed. However, typically only a fraction of base documents did actually change, making this approach rather inefficient. Incremental recomputation approaches, in contrast, avoid re-processing unchanged base data and are thus often preferable.

To perform incremental recomputations, dedicated MapReduce programs may be derived from the original programs in a similar way that incremental SQL/RA expressions are derived from view definitions for view maintenance. We will refer to such programs as *incremental MapReduce programs*. The latter problem has been extensively studied in the database research community as discussed in Section 2.3.3. Since all MapReduce use cases included in our case study perform aggregation-like tasks, work on the maintenance of aggregate views proved most relevant. We found that strictly algebraic approaches (such as [GL95, Qua96]) could not easily be transferred into the MapReduce environment. However, we found the so-called summary delta algorithm proposed and refined in [MQM97, LYC+00, GM06] to be a better fit for the MapReduce programming model. The summary delta algorithm will be discussed in Section 7.4.3.

7.4.2 HBase change capture

Identifying changes at the base data is a prerequisite for incremental recomputations. Gathering change information (or deltas) at source systems is referred to as Change Data Capture (CDC) in the relational context. We presented a survey of existing CDC techniques in Section 5.1 and identified four main approaches, namely, computing snapshot differentials, change tracking, utilizing audit columns, and log-based CDC. In this section we will briefly discuss ways to capture changes in HBase.

Computing snapshot differentials means to infer deltas by comparing periodically taken database snapshots. This technique is expensive, especially for large datasets and usually used only when dealing with unsophisticated sources. Change tracking strongly depends on the source system's ability to perform joins efficiently and is thus ill-suited for HBase.

Changes may also be captured using audit column that store timestamps or version numbers. HBase has native support for timestamping and versioning. Update operations are not done in-place, but create a new version with a fresh timestamp. Similarly, deletions work by creating a so-called tombstone marker

```
CREATE VIEW Parts AS
SELECT partID, SUM(qty*price) AS revenue, COUNT(*) AS tplcnt
FROM Orders
GROUP BY partID
```

Figure 7.1: Sample view definitions

in the version history that masks deleted values, i.e. deletions are logical rather than physical. Hence, the timestamps in the version history can be used as selection criteria to retrieve rows that have been modified since some previous point in time. We have implemented a timestamp-based CDC approach on top of HBase.

Log-based CDC works by keeping a log of changes that is appended when updates occur. In database systems log-based CDC techniques either use dedicated log tables, which are usually populated by triggers, or read changes from the transaction log (log scraping). Both approaches are feasible in HBase. Just like traditional databases, HBase maintains transaction logs for recovery purposes that could be scraped. Triggers are not supported in HBase. However, log tables can be maintained by an extended HBase region server implementation. We implemented a CDC mechanism following the latter approach on top of HBase that we refer to as log table-based CDC. Timestamp-based CDC and log table-based CDC will be further discussed and compared to each other in Section 7.5.3, which reports on the experimental evaluation.

7.4.3 The summary delta algorithm

The Summary Delta Algorithm was originally proposed for the maintenance of data cubes and summary tables in data warehouses in [MQM97]. It is applicable to aggregate views defined using the SQL aggregate functions SUM, COUNT, and AVG. The basic idea of the summary delta algorithm is to aggregate deltas captured at the sources to compute the effect on the materialized view. In [LYC+00] several variations of the original algorithm have been proposed that use alternative delta installation strategies.

Consider the sample view definition shown in Figure 7.1, which has been taken from [LYC+00]. The materialized view is defined over a table storing customer orders and computes the revenue achieved per part. For this purpose, order tuples are grouped by part id and the revenue is computed by summing up the products of quantity and price.

The base table may change over time as orders are inserted, updated, and deleted. Suppose that deltas are captured in two separate tables such that insertions are stored in the first table, deletions are stored in the second table, and updates are stored in both tables as delete-insert pairs. To maintain the

```
SELECT partID, SUM(revenue) AS revenue, SUM(tplcnt) AS tplcnt
FROM (
   (SELECT partID, SUM(qty*price) AS revenue,
           COUNT(*) as tplcnt
   FROM Orders_Insertions
   GROUP BY partID)
   UNION ALL
   (SELECT partID, -SUM(qty*price) AS revenue,
           -COUNT(*) as tplcnt
   FROM Orders_Deletions
   GROUP BY partID)
)
GROUP BY partID
```

Figure 7.2: Maintenance expression for the view definition in Figure 7.1

stale materialized view, a so-called summary delta table is computed. For this purpose, the summary delta algorithm automatically derives a so-called maintenance expressions from the original view definition.

The maintenance expression derived from the sample view definition is shown in Figure 7.2. In the two nested sub-queries, the same grouping and aggregation functions that are applied to the base table in the original view definition are applied to the delta tables. Note that the aggregated values computed from deleted tuples are negated. The result of both sub-queries is unioned and combined in a final aggregation step, which again uses the same grouping and aggregation functions.

The maintenance expression computes the so-called summary delta table, which allows for maintaining the materialized view incrementally. The summary delta table essentially encodes the changes in the aggregated values in the materialized view. For each of the affected aggregated values, the summary delta table contains a value to be added (or subtracted if negative). Reconsider the sample view definition. The maintenance expression shown in Figure 7.2 computes the increase or decrease in revenue and in quantity for each part.

Applying the summary delta table to the materialized view is referred to as delta installation. In [MQM97], an update-in-place installation strategy is proposed. An alternative strategy referred to as summary delta with *overwrite installation* has been proposed in [LYC+00]. To distinguish the two, we will refer to the former approach as *increment installation* in the following.

As suggested in Figure 7.3, increment installation works by applying the summarized deltas to the materialized view in-place. The summary delta algorithm with overwrite installation, in contrast, uses the content of the materialized view as additional input and merges it with the summarized deltas as suggested in Figure 7.3. Unlike to the increment installation variant, the

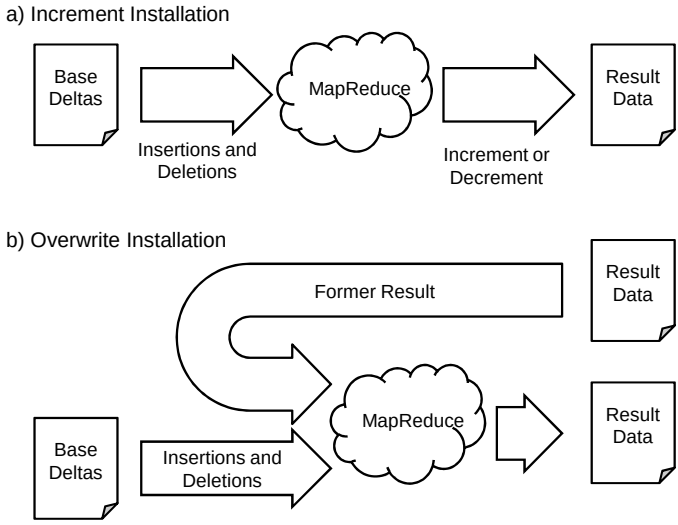


Figure 7.3: Increment installation and overwrite onstallation

overwrite installation variant does not compute view deltas, but rather the refreshed view content itself, which is used to replace (overwrite) the out-dated view.

7.4.4 Incremental recomputation of word histograms and reverse web-link graphs

As a first step in the case study, we manually derived incremental variants from several typical MapReduce programs discussed in Section 7.4.1 to see whether view maintenance techniques work in a MapReduce environment. In this section, we will describe our experiences for two out of these programs – the programs for computing word histograms and reverse web-link graphs. A more general solution to the problem of deriving incremental MapReduce programs will be presented in the following section.

Reconsider the MapReduce program for computing word histograms introduced in Section 7.1.1. As we will see the summary delta algorithm naturally fits the MapReduce programming model. An incremental variant of the original program that follows the overwrite installation approach is shown in Algorithm 3. The Reducer and Combiner functions have been omitted, because the respective functions from the original program (Algorithm 1) can be reused

Algorithm 3 Incremental word count - overwrite installation

```
1: function Map(key, value)
2:     if key is inserted then
3:         for all word in value do
4:             emit(word, 1)
5:         end for
6:     else if key is deleted then
7:         for all word in value do
8:             emit(word, -1)
9:         end for
10:    else
11:        emit(key, value)
12:    end if
13: end function
```

without changes. The MapReduce program reads in two data sets as shown in Figure 7.3. First, it retrieves deltas from the base document collection. We will discuss how deltas can be captured and retrieved in HBase in Section 7.4.2. Second, it reads the result computed before, either by the original program or the previous run of the incremental program. Deltas are provided as two sets, the set of inserted documents and the set of deleted documents. Updates are modeled implicitly as a deletion of the original document and a re-insertion of the modified document.

The Map function of Algorithm 3 processes input key-value pairs differently depending on their origin. Inserted or deleted documents are tokenized into words. Like in the original program, these words are subsequently used as intermediate key. An intermediate value of one or minus one is assigned to words from inserted or deleted documents, respectively. Key-value pairs from the result table (i.e. words with occurrence frequencies) are simply passed on unmodified. In the Combine and Reduce phases, the counts emitted for each word are added together to yield the final result. At last, the result table is overwritten. Note that the incremental MapReduce program reads from and writes to the same table. This can safely be done, because the MapReduce framework does not spawn any Reducers until all map tasks are completed.

The summary delta with increment installation provides an alternative approach to incremental recomputations. In contrast to the overwrite installation approach, the result table is not overwritten but updated in-place as indicated in Figure 7.3. Algorithm 4 shows a MapReduce program to recompute word counts incrementally using the increment installation approach. Again, the Combine function of Algorithm 1 can be reused and has therefore been omitted. The Map function reads in deltas from the base document collection and processes them just like the Map function of Algorithm 3 (i.e. documents are tokenized into words that are assigned a count of one if the respective docu-

Algorithm 4 Incremental word count - increment installation

```
 1: function Map(key, value)
 2:     for all word in value do
 3:         if key is inserted then
 4:             emit(word, 1)
 5:         else if key is deleted then
 6:             emit(word, -1)
 7:         end if
 8:     end for
 9: end function
10: function Reduce(word, list of values)
11:     for all v in values do
12:         sum = sum + v
13:     end for
14:     inc(word, 'someColFamily', 'someColQualifier', sum)
15: end function
```

ment has been inserted or minus one if it has been deleted). The Map function does, however, not read in the result computed previously. In the Combine and Reduce phases, the counts for each word are summed up. Note that the result obtained does not express absolute numbers ("the word x occurs 100 times now"), as is the case for the overwrite installation approach, but an increase or decrease ("the word x has 10 more occurrences now"). The result is used to update the affected rows in the result table in-place. HBase supports single-row transactions, hence values can be increased or decreased atomically. In the pseudo-code of Algorithm 4, this operation is indicated by the "inc" command.

Note that the Reduce phase may be omitted when increment installation is used. In fact, the pre-aggregated output of the Combine phase can immediately be applied to the result table. In this case, the final aggregation is performed by sequences of increment operations targeted to the same row in HBase. Leaving out the Reduce phase may be advantageous, if the final aggregation cannot reduce the data volume considerably.

Reconsider the somewhat more complex MapReduce use case of computing reverse web-link graphs. The summary delta algorithm was originally intended for the standard SQL aggregate functions COUNT, SUM, and AVG, which are numerical and self-maintainable aggregate functions. Recall, that a materialized view is called self-maintainable, if it can be maintained using only the deltas and the content of the view itself, without accessing the base data [GJM96]. However, computing reverse web-link graphs is essentially a set aggregation.

A set aggregation is neither numerical nor self-maintainable and the result computed by Algorithm 2 cannot easily be maintained incrementally. While it is straightforward to deal with links inserted into base documents, deleted

Algorithm 5 Self-maintainable reverse web-link graph

```
 1: function Map(url, document)
 2:     for all link in document do
 3:         source.url = url
 4:         source.count = 1
 5:         emit(link.target, source)
 6:     end for
 7: end function
 8: function Reduce(target, list of sources)
 9:     for all s in sources do
10:         sum[s.url] = sum[s.url] + s.count
11:     end for
12:     for all distinct s in sources do
13:         emit(target, 'someColFamily', s.url, sum[s.url])
14:     end for
15: end function
```

links pose a problem. The reason is that several links having the same target may exist in a single document. When a deletion is propagated, one cannot know if the link needs to be removed from the resulting reverse web-link graph (because no other link with the same source and target remains) or must not be removed (because similar links continue to exist in the base document). The decision can be made by accessing the base data, which would incur additional cost. Algorithm 2 is thus said to be not self-maintainable. Note that the word count program, in contrast, is self-maintainable.

Self-maintainability can sometimes be achieved by altering the view definition to include some additional information. The reason that set aggregation is not self-maintainable is the elimination of duplicates. To achieve self-maintainability, Algorithm 2 may be changed to essentially compute a multiset aggregation instead. To turn a set aggregation into a multiset aggregation it is sufficient to provide an additional counter per set element and include it into the materialized view. A variant of Algorithm 2, rewritten in such a way is shown as Algorithm 5. Note that the reverse web-link graph with added element counters naturally fits the HBase data model. As before, target URLs are used as row keys and source URLs as column qualifiers in some fixed column family. The link quantity is stored as column value under the corresponding column qualifier.

From Algorithm 5 incremental MapReduce programs may be derived. A program following the increment installation approach is shown as Algorithm 6. Similar to the original program, the Map function emits counts of one and minus one for inserted and deleted links, respectively. The Reduce function adds up the counts per link and stores the sums in the result table together with the respective link sources. When a deletion is propagated, the respective

Algorithm 6 Reverse web-link graph - increment installation

```
 1: function Map(url, document)
 2:     for all link in document do
 3:         source.url = url
 4:         if url is inserted then
 5:             source.count = 1
 6:         else if url is deleted then
 7:             source.count = -1
 8:         end if
 9:         emit(link.target, source)
10:     end for
11: end function
12: function Reduce(target, list of sources)
13:     for all s in sources do
14:         sum[s.url] = sum[s.url] + s.count
15:     end for
16:     for all distinct s in sources do
17:         if sum[s.url] != 0 then
18:             inc(target, 'linkFamily', s.url, sum[s.url])
19:         end if
20:     end for
21: end function
```

Algorithm 7 Reverse web-link graph - overwrite installation

```
 1: function Map(key, value)
 2:     if key is inserted then
 3:         for all link in value do
 4:             source.url = key
 5:             source.count = 1
 6:             emit(link.target, source)
 7:         end for
 8:     else if key is deleted then
 9:         for all link in value do
10:             source.url = key
11:             source.count = -1
12:             emit(link.target, source)
13:         end for
14:     else
15:         emit(key, value)
16:     end if
17: end function
```

counter is simply decremented. When zero is reached, the respective link can be removed from the reverse web-link graph or simply be ignored by readers. Obviously, the MapReduce view is self-maintainable.

An incremental program following the overwrite installation approach is shown as Algorithm 7. The Reduce function has been omitted, because again, the Reduce function of the original program (Algorithm 5) can be reused without changes. The Map function resembles the one of Algorithm 6 for increment installation. The difference is that the materialized view is used as additional input source and any tuples read from it are passed on unmodified.

All MapReduce use cases considered in our case study essentially perform either numerical, set, or multiset aggregations. Hence, we were able to derive incremental programs as described in this section in each case. In the following section, we will propose a general solution for this kind of MapReduce programs.

7.4.5 Incremental recomputations in the general case

In the case study, all incremental MapReduce programs were initially derived manually. This was an insightful exercise that led to two important observations. First, we found ourselves following a common pattern for each sample MapReduce program (view). Second, we realized that each of these views belongs to a common class – the class of self-maintainable aggregate views. In this section, we will formally define this class of MapReduce views and provide a general solution to the problem of deriving incremental MapReduce programs from members of this class of views.

Aggregate MapReduce views are evaluated in three steps. First, the source key-value pairs are translated into intermediate key-value pairs. Second, the intermediate key-value pairs are grouped together based on the intermediate key. Third, an "aggregate function" is applied to each group of intermediate values resulting in a single (possibly set-valued) result value. The result value along with its grouping key is stored at the materialized view.

Abstractly, a MapReduce aggregate view can thus be defined by a translation function and an aggregate function. The translation function translates source key-value pairs into intermediate key-value pairs. Recall, that a single source key-value pair may be translated into multiple intermediate key-value pairs. We denote the domain of source keys, source values, intermediate keys, and intermediate values as D_{sk}, D_{sv}, D_k, and D_v, respectively. We denote the power set of a set S as $\mathcal{P}(S)$. A translation function is defined as *translate* : $D_{sk} \times D_{sv} \to \mathcal{P}(D_k \times D_v)$.

An aggregate function is defined by a binary function $\circ : D_v \times D_v \to D_v$. Let $v_1, \ldots, v_n \in D_v$ be a group of intermediate values, then the aggregated value $a \in D_v$ in the materialized view is computed as $a := v_1 \circ v_2 \circ \ldots \circ v_n$. Because an aggregate function should be insensitive to the order of the input values, we

require the function $\circ$ to be commutative and associative.

A view is called self-maintainable if it can be maintained using the base deltas and the content of the view itself, without accessing the base data [GJM96]. Let $\Delta V = \{a_{n+1}, \ldots, a_m\}$ be inserted values (deltas). The latest aggregate value a' in the materialized view is computed as $a' := v_1 \circ \ldots \circ v_n \circ v_{n+1} \circ \ldots \circ v_m$. Alternatively, a' can be computed using the content of the materialized view itself as $a' := a \circ v_{n+1} \circ \ldots \circ v_m$. Thus, an aggregate view based on a commutative and associative function $\circ$ is always self-maintainable w.r.t. insertions.

Let $\nabla V = \{v_i, \ldots, v_k\}, 1 \leq i < k \leq n$ be deleted values (deltas). The latest aggregate value a' in the materialized view is computed as $a' := v_1 \circ \ldots \circ v_{i-1} \circ v_{k+1} \circ \ldots \circ v_n$. To achieve self-maintainability a' needs to be computable using only the content of the materialized view a and the deltas ∇V. This can be done if each element $v \in D_v$ has an inverse element $v^* \in D_v$ such that $v \circ v^* = e$, with e being the neutral element w.r.t. function $\circ$. If this condition holds, the latest aggregate value can be computed as $a' := a \circ v_i^* \circ \ldots \circ v_k^*$. Note that a self-maintainable aggregate function is thus defined by an Abelian group $(D_v, \circ)$.

In summary, a MapReduce aggregate view definition can be specified by four user-defined functions.

- A function *translate* : $D_{sk} \times D_{sv} \rightarrow \mathcal{P}(D_k \times D_v)$
- A function $\circ : D_v \times D_v \rightarrow D_v$ that is commutative and associative.
- A function $* : D_v \rightarrow D_v$ that maps each element of the Abelian group $(D_v, \circ)$ to its inverse element.
- A nullary function $e : \emptyset \rightarrow D_v$ that returns the neutral element of the Abelian group $(D_v, \circ)$.

Given these functions, MapReduce programs that fully compute or incrementally recompute aggregate views can be assembled. Consider Algorithm 8 that performs a full (re)computation. In the Map phase the translation function is applied to obtain intermediate key-value pairs. Intermediate values having the same intermediate key are grouped together. In the Combine and Reduce phase, the elements of each group are composed using the function $\circ$ to obtain an aggregated result value.

Algorithm 9 and Algorithm 10 maintain aggregate views using overwrite or increment installation, respectively. The Map function of Algorithm 9 reads in the deltas and the content of the materialized view. The deltas are fed into the translate function, while the content of the materialized view is simply passed through. The Map function of Algorithm 10 reads and translates the deltas only. The Combine and Reduce functions of Algorithm 9 (overwrite installation) are similar to Algorithm 8. Increment installation differs in the

Algorithm 8 Full (re)computation

```
1: function Map(key, value)
2:     for all v in translate(value) do
3:         emit(v.key, v.value)
4:     end for
5: end function
6: function Combine(key, values)                          ▷ optional
7:     temp := e                                          ▷ neutral element
8:     for all v in values do
9:         temp = temp ∘ v
10:    end for
11:    emit(key, temp)
12: end function
13: function Reduce(key, values)
14:    temp := e
15:    for all v in values do
16:        temp = temp ∘ v
17:    end for
18:    write(key, temp)
19: end function
```

Algorithm 9 Incremental recomputation - overwrite installation

```
1: function Map(key, value)                               ▷ Reads deltas and the view
2:     if key is inserted then
3:         for all v in translate(value) do
4:             emit(v.key, v.value)
5:         end for
6:     else if key is deleted then
7:         for all word in translate(value) do
8:             emit(v.key, v.value*)                      ▷ inverse element
9:         end for
10:    else
11:        emit(key, value)
12:    end if
13: end function
14: function Combine(key, values)                         ▷ see Algorithm 8
15:    ...
16: end function
17: function Reduce(key, values)                          ▷ see Algorithm 8
18:    ...
19: end function
```

Algorithm 10 Incremental recomputation - increment installation

```
1: function Map(key, value)                                   ▷ Reads deltas only
2:     if key is inserted then
3:         for all v in translate(value) do
4:             emit(v.key, v.value)
5:         end for
6:     else if key is deleted then
7:         for all word in translate(value) do
8:             emit(v.key, v.value*)                          ▷ inverse element
9:         end for
10:    end if
11: end function
12: function Combine(key, values)                             ▷ optional
13:    temp := e                                              ▷ neutral element
14:    for all v in values do
15:        temp = temp ∘ v
16:    end for
17:    emit(key, temp) ▷ If Reducer is used. Alternatively, write to the materialized
    view.
18: end function
19: function Reduce(key, values)
20:    temp := e
21:    for all v in values do
22:        temp = temp ∘ v
23:    end for
24:    begin                                                  ▷ single-row transaction
25:        old := read(key)                                   ▷ read from the materialized view
26:        new := old ∘ temp
27:        write(key, new)                                    ▷ write to the materialized view
28:    commit
29: end function
```

way deltas are applied. Algorithm 10 uses the notion of a single-row transaction to atomically read a value from the materialized view, update it, and write it back (lines 25 - 29). Note that HBase provides an atomic increment operation that can be used if $\circ$ is the addition operation.

We review the MapReduce programs to compute word histograms and reverse web-link graphs discussed in the previous section to show how these fit into our theoretical framework. In both cases, D_{sk} is the set of all URLs and D_{sv} is the set of all html documents. The *translate* function of the word histogram view produces intermediate key-value pairs where D_k is the set of all words and D_v is the natural numbers $\mathbb{N}$. The aggregate function is based on the Abelian group $(+, \mathbb{N})$, i.e. the operation $\circ$ is addition, the inverse element of $n \in \mathbb{N}$ is $n^* := -n$, and the neutral element e is zero.

The *translate* function of the reverse web-link graph view produces inter-

mediate key-value pairs where D_k is the set of all URLs and D_v is the power multiset of all URLs. The aggregate function is based on the Abelian group $(\uplus, D_v)$, where $\uplus$ denotes multiset union. The inverse element is computed by negating the multiplicity of the elements in a multiset and the neutral element is the empty set.

The lessons learned from the case study allow for designing a programming model on top of MapReduce for defining self-maintainable aggregate computations. Such a programming model will provide hooks to plug-in user-defined functions including a translation function, a commutative and associative aggregate function, an inverse element function, and a neutral element function, as discussed earlier. For any computational problem that can be described in this form, MapReduce jobs for the incremental recomputation of previous results can be assembled automatically. A framework supporting the proposed programming model for self-maintainable aggregates is currently being developed in the course of a Master thesis[2].

7.5 Evaluation

In this section we will evaluate our approach to incremental recomputations in MapReduce proposed in the previous section with regard to view consistency (Section 7.5.1), the MapReduce fault tolerance (Section 7.5.2), and performance gains as compared to full recomputations (Section 7.5.3).

7.5.1 View consistency

We expressed several concerns about materialized views on top of HBase in Section 7.3, one of it being the restriction to single-row transactions. As a consequence result tables (views) cannot be updated atomically and readers may see intermediate states of the view maintenance process. Three levels of consistency can be distinguished.

First, all visible rows result from either the last recomputation or the next to last recomputation. In other words, readers may see rows that have not yet been updated while view maintenance is in progress. This level of consistency is achieved by both the overwrite installation approach and the increment installation approach when a Reduce function is used.

Second, a weaker level of consistency is achieved by the increment installation approach when the Reduce phase is omitted. Here, the view may take any state while view maintenance is in progress. This is because intermediate Map results are used to update result rows in place.

[2] Johannes Schildgen, "Ein MapReduce-basiertes Programmiermodell für selbstwartbare Aggregatsichten" (working title)

Third, consistency, in its classical sense, can be achieved by exploiting HBase versioning. If readers are aware of the point in time the last view maintenance process was completed, queries can be limited to the respective time range to exclude any (potentially dirty) data written ever since. A "valid timestamp" could be provided to readers as additional view metadata. It can be updated atomically whenever a view maintenance cycle is completed. The choice of a suitable consistency level, however, depends on application requirements.

7.5.2 Fault tolerance

Tolerating machine failures in a graceful manner is a major design goal of MapReduce. The framework follows a forward recovery strategy and re-executes Map and Reduce tasks of failed machines. When a Mapper is re-executed, during an incremental recomputation, its input data may have been updated meanwhile. However, in HBase multi-row transactions are not supported and, hence, no notion of cross-row consistency exists. Because MapReduce programs cannot make any assumptions about cross-row consistency, re-executing a failed Map task will not change the semantics of a computation (whether incremental or not).

Re-executing a Reduce task is more delicate, as the failed Reducer may have written partial results to HBase already. If an idempotent write operation is used, as in the overwrite installation approach, the view will not become inconsistent. However, the increment operation is not idempotent and must thus not be re-applied to the same row. Idempotence of increment installation could be achieved by exploiting HBase's versioning capabilities. By means of timestamps (or version numbers) assigned with each update operation, values that have already been incremented by a failed Reducer could be identified. Such values would need to be skipped, when the Reduce task is re-executed.

7.5.3 Performance

We performed experiments on a six-node cluster (Xeon Quadcore CPU at 2.53GHz, 4GB RAM, 1TB SATA-II disk, Gigabit Ethernet) running Apache Hadoop and Apache HBase. One cluster node was designated the master and ran the MapReduce, HDFS, and HBase master processes (job tracker, name node, and HBase master). The remaining five nodes served as worker nodes (task tracker, data node, and region server).

We used a test dataset consisting of 5M html documents of about 60GB in total, with exception of the inverted index computation (e) for which we used a smaller dataset of about 12GB in size. The dataset was created with a data generation tool originally developed for a benchmark between MapReduce and parallel database systems in [PPR+09] and made publicly available for others

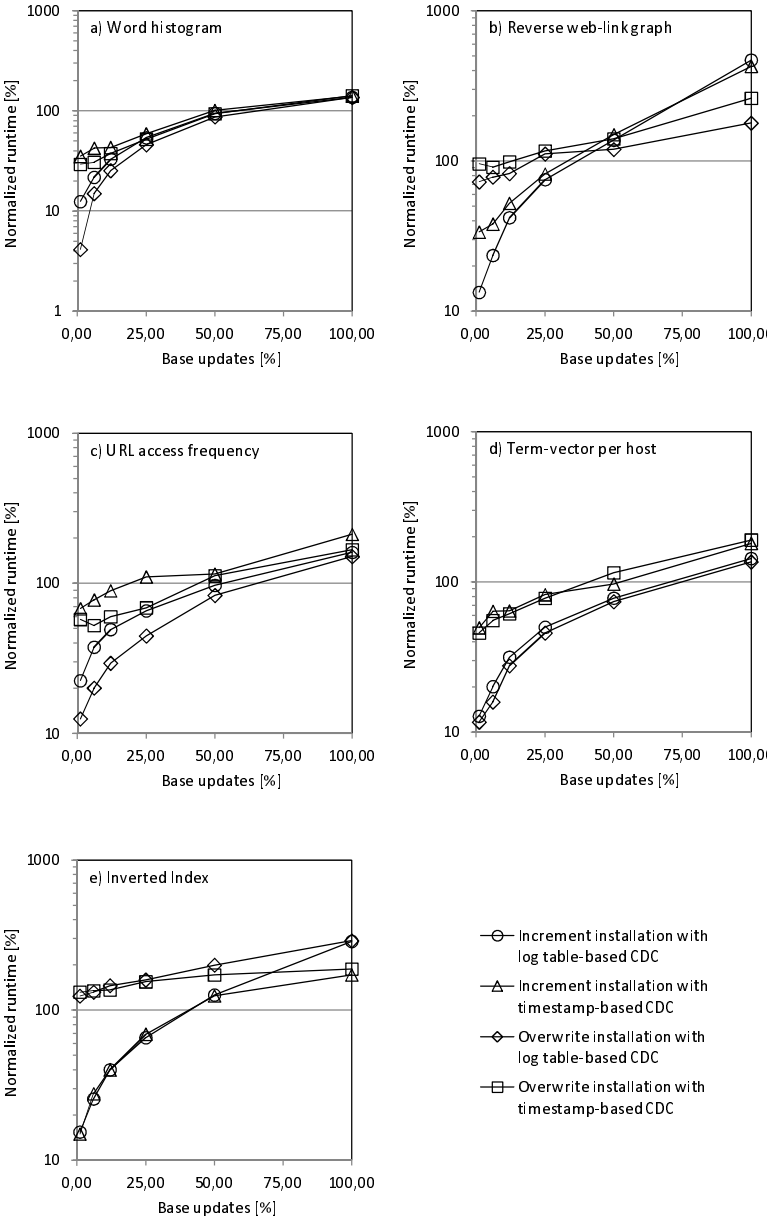


Figure 7.4: Processing times for incremental recomputation approaches

to recreate the results [Bro]. Each document in the collection contained about 10^3 words, randomly chosen from a set of 10^5 distinct words with a uniform distribution, and about 40 hyperlinks with a Zipfian distribution of link targets.

Figure 7.4 shows the processing times for the MapReduce use cases discussed in Section 7.4.1, namely recomputing word count histograms (a), reverse web-link graphs (b), URL frequencies (c), term vectors per host (d), and inverted indexes (e). Each plot in Figure 7.4 depicts the processing times of four different incremental recomputation approaches using either overwrite or increment installation along with log table-based or timestamp-based change capture, which have been discussed in Section 7.4.2. Note that the runtime measurements are normalized as percent of the naive full recomputation time, which is unaffected by the volume of base document updates.

The incremental variants performed considerably better than the full recomputation for small numbers of base document updates throughout our experiments. This may seem unexpected, considering the concerns about BHase's lack of secondary indexes expressed in Section 7.3. However, this restriction is unproblematic for self-maintainable views, because no access to base data is required. Hence, none of the incremental MapReduce programs was dependent on efficient access paths.

Log table-based CDC outperformed timestamp-based CDC throughout our experiments. However, we were surprised to see how small the difference was, because the latter technique requires a full table scan to extract deltas. Apparently HBase is able to skip over irrelevant rows quickly. The drawback of log table-based CDC is that it requires additional write operations to maintain the log, i.e. write operations roughly become twice as expensive (not subject of the experiment).

The experimental results do not show a clear winner among the alternative delta installation approaches. Overwrite installation outperformed increment installation in the recomputation of word count histograms (a), for instance. Recall that the input documents were generated from 10^5 distinct words with a uniform distribution. Hence, the resulting word count histogram was rather small (about 4MB) and had to be updated almost completely, even for small numbers of base document updates.

Increment installation was superior for the incremental recomputation of reverse web-link graphs (b), for instance. The reason is that the link targets were Zipf-distributed in the base document collection, i.e. the majority of hyperlinks in the document collection are targeted to a small set of documents. A limited number of base documents updates thus required a small number of updates in the reverse web-link graphs, mainly to the frequently referenced nodes.

Moreover, increment installation greatly outperformed overwrite installation in recomputing inverted indexes (e). This can be explained by the size of the materialized view. It is more than six times as big as the base data, i.e. about

80GB in our benchmark setup. For this reason, overwrite installation suffers from a significant I/O overhead for reading the inverted index view, whereas increment installation reads the much more compact delta sets only.

Whether overwrite or increment installation is preferable depends on the ratio of view updates to the overall view size. While overwrite installation completely rebuilds the materialized view, increment installation performs fewer updates in place. An increment operation, however, requires a random read on the materialized view and is thus more expensive than an overwrite operation.

In summary, our experiments show that incremental maintenance of MapReduce views achieves a considerable efficiency gain compared to recomputing from scratch unless the base data changed dramatically. This gain in efficiency allows for keeping MapReduce views more up-to-date by increasing the recomputation frequency or for performing view maintenance with less hardware resources.

7.6 Chapter Summary

In this chapter, we pursued the idea of transferring incremental view maintenance techniques into the MapReduce environment to increase the efficiency of recurrent computations. We started out with a case study on several MapReduce programs considered as typical examples in Google's original MapReduce paper. Even though the programming and data model of MapReduce differs considerable from that of relational databases, we were able to reuse and generalize work on the maintenance of aggregate views from the database context. The insights gained from the case study led to a general solution to the incremental recomputation problem for a certain class of MapReduce programs (or views) that we referred to as self-maintainable aggregate views.

We believe that the class of self-maintainable aggregates is of great practical relevance. The fact that all use cases taken from the original MapReduce paper belong to this class is a fairly strong indication and there are further noteworthy examples. A recently published paper coauthored by the University of Michigan, Google Research, and Yahoo! Research, proposes the distributed computation of data cubes using MapReduce [NYBR11]. A recent bachelor thesis in our group showed that data cubes belong to the class of self-maintainable aggregates and can efficiently be maintained using our approach [Sch12].

Another interesting real-world use case was shared with us by a Japanese Telco company. To improve the quality of speech recognition, the probability with which certain words appear after a given word or phrase is computed, which allows the speech recognition system to make more informed guesses. The word occurrence probabilities are computed from twitter feeds within a sliding time window using MapReduce. As we have been told, the probabilities

are fully recomputed from time to time and an incremental solution would be much needed.

To compute the word occurrence probabilities, two subsequent MapReduce jobs may be used. The first job may extract pairs of words from twitter feeds and count the number of occurrences of each word pair. The second job may count the number of occurrences of word pairs starting with the same word. To do so, it may input the result of the first job and use the first word in each pair as intermediate key and sum up the per-pair frequency counts. The probability that a given word a follows another given word b is the quotient of the frequency of the $\langle b, a \rangle$ word pairs and the overall frequency of pairs starting with b. This quotient may either be computed by the second MapReduce job or on-the-fly when accessed by the client. In either case, the frequency counts must be stored separately in the materialized view to facilitate incremental maintenance. The approach sketched so far may be extended to compute the probabilities with which certain words appear after given sequences of words with a fixed length.

As we have seen, the problem of computing word occurrence probabilities may be broken down into two sub-problems, which are both expressible though the API proposed in Section 7.4.5 for self-maintainable aggregate views. Hence, the overall problem can be recomputed incrementally following the approach described in the same section. We believe that there are many other useful real-world MapReduce computations that may be implemented through the API we proposed in this chapter, i.e. expressed in terms of (sequences of) self-maintainable aggregate computations. In some sense, our API follows the same spirit as the original MapReduce API. The latter forces programmers to structure a computational problem in a way such that it may easily be computed in parallel. In a similar way, our API forces programmers to structure a computational problem in a way such that its result may easily be recomputed in an incremental fashion.

8 Conclusion

In this thesis we proposed approaches to perform recurrent computations in an incremental manner in areas of data management where naively recomputing from scratch is more common today. Incremental recomputation techniques have been studied by the database research community mainly in the context of the maintenance of materialized views. As we pointed out, from an abstract point of view, there are interesting similarities between the concept of materialized views and two other areas of data management, namely materialized data integration using ETL middleware and data-intensive computations using MapReduce. In either environment derived datasets are computed from given base datasets and need to be recomputed (or maintained) when the underlying base data is changed. There has been a lot of research on the maintenance of materialized views and it has been shown that incremental view maintenance is generally more efficient than recomputing views from scratch. As we have shown, similar performance gains can be achieved in ETL and MapReduce.

Incremental view maintenance techniques have been adopted by all major commercial database systems today. However, we are not aware of any ETL tool that supports incremental recomputations. We do not claim that no data warehouse is incrementally maintained today – we rather feel that ETL programmers nowadays are in much the same situation as database programmers used to be in the early 1990s. While it was certainly possible to build custom solutions to maintain derived tables (or materialized views) incrementally, database systems did not support programmers in doing so. The same is true for today's ETL tools. While it is certainly possible to build incremental ETL jobs manually, this is complex and error-prone. Obviously, it was advantageous if ETL tools supported incremental maintenance just like today's database systems.

In this thesis we explored the idea of transferring incremental view maintenance techniques into the ETL environment to improve the efficiency of data integration. We followed the same fundamental approach as a view maintenance technique known as algebraic differencing. In a nutshell, algebraic differencing takes a view definition, which essentially describes how to fully compute a materialized view, and derives so-called maintenance expressions, which describe how to recompute the view incrementally. We proposed to follow the same basic approach and rewrite ETL jobs to obtain incremental counterparts.

We chose the algebraic differencing technique because it has some nice prop-

erties. It is transparent to the user, i.e. the rewriting process is completely automatic. Furthermore, both view definitions and maintenance expressions are expressed in relational algebra, i.e. the same language is used in either case. Similarly, we expressed incremental ETL jobs using just standard ETL operators. That way, incremental ETL jobs can be executed by standard ETL tools without modifications.

Bringing the algebraic differencing method into the world of ETL was not straightforward. We faced four major challenges that were each addressed in a separate chapter. In Chapter 3 we argued that native ETL operator models are not appropriate for algebraic differencing. There is no standard ETL transformation language but ETL tools use proprietary operator models. ETL operators should be thought of as composable, low-level data transformation utility programs rather than logical and sound operators as used in the relational algebra. We proposed a model to capture ETL transformation semantics in an abstract and platform-independent way. In doing so, we were able to build on and extend Orchid's Operator Hub Model. As we showed, our abstract ETL model is well suited for algebraic differencing.

In Chapter 4 we addressed the challenge of optimizing incremental ETL jobs. We showed that incremental ETL jobs derived using the standard rules of algebraic differencing may suffer from severe performance problems. Such incremental ETL jobs are often outperformed by jobs taking the naive full recomputation approach. This is because ETL puts a strong focus on data integration. ETL jobs typically involve a number of data standardization and cleansing operations. These operations are unique to the ETL environment and hence have not been considered in the context of incremental view maintenance before. We tackled the problem of optimizing ETL jobs by improving the rules for differencing in a step-wise process. In doing so, we were able to reuse concepts and techniques developed in other areas of database research including deductive databases and integrity checking.

In Chapter 5 we surveyed change data capture (CDC) techniques that are used at operational source systems to detect data changes, or deltas, as they are often referred to. Source deltas act as input for incremental ETL jobs. Our key observation was that common CDC techniques may not capture deltas completely. Incomplete (or partial) input deltas have not been considered in the context of incremental view maintenance before. In fact, established view maintenance techniques assume deltas to be fully known and, as we showed, produce wrong results if deltas happen to be partial. To allow incremental ETL jobs to benefit from any CDC technique, we proposed a generalized delta model that captures both complete and partial deltas. Furthermore, we generalized algebraic differencing to work for partial deltas. The original technique is a special case of ours, i.e. for non-partial deltas our technique degenerates to the original one.

In Chapter 6 we discussed anomalies that may occur when materialized views are incrementally maintained over distributed sources. Such maintenance anomalies may leave the materialized view in an inconsistent state and must thus be avoided. We showed that similar anomalies may occur in incremental ETL processing. A number of techniques to avoid maintenance anomalies have been proposed in the context of distributed view management. We argued that these techniques cannot be applied in ETL, however, because they do not fit into the data-flow oriented processing model and make strong assumption about the source systems. For this reason, we proposed ways to avoid maintenance anomalies tailored to incremental ETL processing. We considered a broad range of source systems with different properties regarding change capture and data access.

The primary strength of ETL is its ability to integrate data from very heterogeneous source systems. Better than other types of data integration middleware, such as replication systems for instance, ETL copes with autonomous, technically heterogeneous sources and problems related to data quality. It was an important objective to us to preserve these advanced integration capabilities in incremental ETL processing. For this reason, we considered a broad range of source systems with different properties, such as different CDC techniques, throughout our work.

We believe that our work provides the basis for future ETL tools that support incremental recomputations. Our approach has been designed to offer an easy migration path. It is not required to modify existing ETL tools to execute incremental ETL jobs, because these jobs are composed form standard ETL operators. Furthermore, our approach is completely transparent to the ETL programmer in the sense that incremental ETL jobs are derived in an automated fashion. In particular, this allows for existing ETL solutions to be "incrementalized" with very little effort.

Near real-time data warehousing is widely perceived as the next step in the evolution of data warehousing. The aim of near real-time data warehousing is to reduce the latency between business transaction at the operational sources and their appearance at the data warehouse. It thus facilitates the analysis of more recent data and thus timelier decision making. Near real-time data warehousing is achieved by shortening the warehouse maintenance cycles, i.e. running ETL jobs more frequently in so-called micro batches. For this to be possible, ETL computations must be highly efficient, because they are no longer performed in nightly batch windows. The efficiency gains of incremental ETL processing may thus pave the way for near real-time data warehousing.

In Chapter 7 we shifted our focus from traditional data integration supported by ETL to MapReduce, a more recent paradigm for data-intensive computations. MapReduce became very popular with the advent of cloud computing and is perceived as key technology in web-scale data management. Several other

research groups started to explore incremental recomputation approaches for MapReduce at about the same time as we did. We classified related work based on a notion of transparency, meaning that any aspects of incremental processing are hidden from the MapReduce programmer. We found the related work to fall into two categories: transparent approaches based on caching intermediate results and non-transparent approaches based on dedicated programming models.

We followed an approach that lies in between these extreme cases. We proposed a programming model on top of MapReduce that allows for expressing a class of computations that we called self-maintainable aggregates. We believe that this class is of high practical relevance and provided several use cases to support our point. Our programming model captures the essentials of the computation in a way such that MapReduce jobs for both, fully recomputing and incrementally recomputing may be composed automatically.

We argue that our approach strikes a balance between transparent and non-transparent approaches proposed in related work. Unlike caching, it is not fully transparent but requires programmers to use a dedicated programming model. At first, this may seem like a disadvantage. However, a well designed programming model may guide the programmer in expressing his problem in the most appropriate way. Much like the MapReduce programming model guides the programmer in the development of parallelizable algorithms, our programming model guides the programmer in the development of incrementally recomputable algorithms. A fully transparent approach cannot provide such guidance. Thus, MapReduce programmers are likely to come up with suboptimal solutions with regard to incremental recomputability.

While our approach is not fully transparent, it is neither fully non-transparent. Recall that non-transparent approaches such as Google Percolator and the CBP system leave the responsibility for the design and implementation of incremental algorithms to the programmer. Our approach offers a higher level of abstraction. It provides hooks for the programmer to plug-in custom code from which incremental MapReduce programs are generated automatically.

We believe that true transparency cannot be achieved without loosing the guiding influence of a well designed programming model, at least if general-purpose programming languages (such as Java in Hadoop) are used for implementing MapReduce jobs, because reasoning about imperative programs is hardly possible. However, several high-level programming languages have recently been build on top of MapReduce. Yahoo! Research developed a system called Pig that uses a scripting language named PigLatin, Facebook developed a system called Hive that uses a language known as HiveQL that lends itself to SQL, and IBM developed a language called Jaql that was inspired by the unix pipes syntax and is being used in the IBM BigInsights product. These high-level MapReduce programming languages are more declarative than the native

MapReduce API based on general-purpose programming languages. Their focus is on the specification of what should be computed, rather than how it should be done.

For future work, we believe that declarativity will be key to achieve true transparency for incremental recomputations in MapReduce and, at the same time, avoid the storage overhead of caching-based approaches. Similar to algebraic view maintenance, declarative MapReduce programs could be differentiated to automatically derive incremental MapReduce programs in the same language. Existing declarative programs may thus be transparently incrementalized. Furthermore, declarative programs may be reasoned about. Just like SQL view definitions in database systems, declarative MapReduce programs (view definitions) may be analyzed and, if needed, rewritten for better incremental maintainability, e.g. by turning set aggregations into multiset aggregations.

Our approach presented to incremental recomputations in MapReduce is limited to a certain class of MapReduce jobs we called self-maintainable aggregations. Though we believe that this class is of high practical relevance, future research should broaden the class of jobs that can be incrementalized. Again, considering high-level MapReduce languages may prove helpful, because their declarativity may allow for algebraic rewriting. It seems possible to formulate delta rules, similar to those used for algebraic differencing, for high-level MapReduce programming primitives.

The work presented in this thesis showed that view maintenance concepts can be transferred into the world of MapReduce to improve the efficiency of recurrent computations. As a next step, high-level MapReduce languages should be considered, which promise more transparency and the ability to handle a broader class of MapReduce jobs.

Bibliography

[AAM+02] Divyakant Agrawal, Amr El Abbadi, Achour Mostéfaoui, Michel Raynal, and Matthieu Roy. The Lord of the Rings: Efficient Maintenance of Views at Data Warehouses. In *DISC*, pages 33–47, 2002.

[AASY97] Divyakant Agrawal, Amr El Abbadi, Ambuj K. Singh, and Tolga Yurek. Efficient View Maintenance at Data Warehouses. In *SIGMOD Conference*, pages 417–427, 1997.

[Apaa] Apache Accumulo (Incubating). `http://incubator.apache.org/accumulo/`.

[Apab] Apache Hadoop. `http://hadoop.apache.org/`.

[BCL89] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.

[Beh09a] Andreas Behrend. A Classification Scheme for Update Propagation Methods in Deductive Databases. In *International Workshop on Logic in Databases (LID)*, 2009.

[Beh09b] Andreas Behrend. A magic approach to optimizing incremental relational expressions. In *IDEAS*, pages 12–22, 2009.

[BG04] Andreas Bauer and Holger Günzel. *Data-Warehouse-Systeme - Architektur, Entwicklung, Anwendung.* dpunkt, 2004.

[BJ10] Andreas Behrend and Thomas Jörg. Optimized incremental ETL jobs for maintaining data warehouses. In *IDEAS*, pages 216–224, 2010.

[BM04] Andreas Behrend and Rainer Manthey. Update Propagation in Deductive Databases Using Soft Stratification. In *ADBIS*, pages 22–36, 2004.

[BMSU86] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *PODS*, pages 1–15, 1986.

[Bro] Brown University Data Management Group. A Comparison of Approaches to Large-Scale Data Analysis. `http://database.cs.brown.edu/projects/mapreduce-vs-dbms/`.

[BWA+11] Pramod Bhatotia, Alexander Wieder, Istemi Ekin Akkus, Rodrigo Rodrigues, and Umut A. Acar. Large-scale Incremental Data Processing with Change Propagation. In *HotCloud*, 2011.

[BWR+11] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut Acar, and Rafael Pasquini. Incoop: MapReduce for Incremental Computations. In *SoCC*, 2011.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[CHRM02] Carmen Constantinescu, Uwe Heinkel, Ralf Rantzau, and Bernhard Mitschang. A System for Data Change Propagation in Heterogeneous Information Systems. In *ICEIS*, pages 73–80, 2002.

[DB2a] *IBM DB2 Database for Linux, UNIX, and Windows Information Center*. `http://publib.boulder.ibm.com/infocenter/db2luw/v9`.

[DB2b] *IBM DB2 Version 9.1 for z/OS Information Center*. `http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/`.

[DF09] Josep Domingo-Ferrer. Record Linkage. In Liu and Özsu [LÖ09], pages 2353–2354.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[DHW+08] Stefan Dessloch, Mauricio A. Hernández, Ryan Wisnesky, Ahmed Radwan, and Jindan Zhou. Orchid: Integrating Schema Mapping and ETL. In *ICDE*, pages 1307–1316, 2008.

[DJ11] Jens Dittrich and Alekh Jindal. Towards a One Size Fits All Database Architecture. In *CIDR*, pages 195–198, 2011.

[DS08a] David J. DeWitt and Michael Stonebraker. MapReduce: A major step backwards. 2008. `http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/`.

[DS08b] David J. DeWitt and Michael Stonebraker. MapReduce II. 2008. `http://databasecolumn.vertica.com/database-innovation/mapreduce-ii/`.

[FBT10] Ted Friedman, Mark A. Beyer, and Eric Thoo. Magic Quadrant for Data Integration Tools. Technical report, Gartner, 2010.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[GHOS96] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *SIGMOD Conference*, pages 173–182, 1996.

[GJM96] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data Integration using Self-Maintainable Views. In *EDBT*, pages 140–144, 1996.

[GL95] Timothy Griffin and Leonid Libkin. Incremental Maintenance of Views with Duplicates. In *SIGMOD Conference*, pages 328–339, 1995.

[GLT97] Timothy Griffin, Leonid Libkin, and Howard Trickey. An Improved Algorithm for the Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.*, 9(3):508–511, 1997.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[GM06] Himanshu Gupta and Inderpal Singh Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Inf. Syst.*, 31(6):435–464, 2006.

[Gör09] Jürgen Göres. *A Model Management Framework for Information Integration*. PhD thesis, TU Kaiserslautern, 2009.

[Hal09] Alon Y. Halevy. Information Integration. In Liu and Özsu [LÖ09], pages 1490–1496.

[HHH+05] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD Conference*, pages 805–810, 2005.

[HS98] Mauricio A. Hernández and Salvatore J. Stolfo. Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. *Data Min. Knowl. Discov.*, 2(1):9–37, 1998.

[HZ96] Richard Hull and Gang Zhou. A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches. In *SIGMOD Conference*, pages 481–492, 1996.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[IDU] IDUG: The Worldwide DB2 User Community. `http://www.idug.org`.

[Ina10] Muhammad Faisal Inam. A Comparison and Evaluation of Change Data Capture Techniques. Master's thesis, University of Kaiserslautern, November 2010.

[Inf] IBM InfoSphere Information Server. `http://www-01.ibm.com/software/data/integration/info_server/`.

[JD08] Thomas Jörg and Stefan Dessloch. Towards generating ETL processes for incremental loading. In *IDEAS*, pages 101–110, 2008.

[JD09a] Thomas Jörg and Stefan Deßloch. Formalizing ETL Jobs for Incremental Loading of Data Warehouses. In *BTW*, pages 327–346, 2009.

[JD09b] Thomas Jörg and Stefan Dessloch. Near Real-Time Data Warehousing Using State-of-the-Art ETL Tools. In *BIRTE*, pages 100–117, 2009.

[JD11] Thomas Jörg and Stefan Deßloch. View Maintenance using Partial Deltas. In *BTW*, pages 287–306, 2011.

[JMS09] Thomas Jörg, Albert Maier, and Oliver Suhre. Generating Extract, Transform, and Load (ETL) Jobs for Loading Data Incrementally. United States Patent, 3 2009.

[JPYD11a] Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Can MapReduce learn form Materialized Views? In *LADIS 2011*, pages 1–5, 9 2011.

[JPYD11b] Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Incremental recomputations in MapReduce. In *Proceedings of the third international workshop on Cloud data management*, CloudDB '11, pages 7–14, New York, NY, USA, 2011. ACM.

[KC04] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data.* John Wiley & Sons, Inc., 2004.

[Kle11] Joachim Klein. Stets Wertvollständig! - Snapshot Isolation für das Constraint-basierte Datenbank-Caching. In *BTW*, pages 247–266, 2011.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[KP81] Shaye Koenig and Robert Paige. A Transformational Framework for the Automatic Control of Derived Data. In *VLDB*, pages 306–318, 1981.

[KR02] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[LGM96] Wilburt Labio and Hector Garcia-Molina. Efficient Snapshot Differential Algorithms for Data Warehousing. In *VLDB*, pages 63–74, 1996.

[LHM+86] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Hamid Pirahesh, and Paul F. Wilms. A Snapshot Differential Refresh Algorithm. In *SIGMOD Conference*, pages 53–60, 1986.

[LN06] Ulf Leser and Felix Naumann. *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. dpunkt, 2006.

[LÖ09] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009.

[LOR+10] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful Bulk Processing for Incremental Analytics. In *SoCC*, pages 51–62, 2010.

[LSS01] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. SchemaSQL: An extension to SQL for multidatabase interoperability. *ACM Trans. Database Syst.*, 26(4):476–519, 2001.

[LYC+00] Wilburt Labio, Jun Yang, Yingwei Cui, Hector Garcia-Molina, and Jennifer Widom. Performance Issues in Incremental Warehouse Maintenance. In *VLDB*, pages 461–472, 2000.

[MFPR90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is Relevant. In *SIGMOD Conference*, pages 247–258, 1990.

[MP94] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of Magic-sets in a Relational Database System. In *SIGMOD Conference*, pages 103–114, 1994.

[MQM97] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *SIGMOD Conference*, pages 100–111, 1997.

[MTK06] Joy Mundy, Warren Thornthwaite, and Ralph Kimball. *The Microsoft Data Warehouse Toolkit: With SQL Server 2005 and the Microsoft Business Intelligence Toolset*. John Wiley & Sons, Inc., 2006.

[Myr05] Thomas Myrach. *Temporale Datenbanken in betrieblichen Informationssystemen: Prinzipien, Konzepte, Umsetzung*. Teubner-Reihe Wirtschaftsinformatik. B.G. Teubner Verlag, 2005.

[NYBR11] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. Distributed Cube Materialization on Holistic Measures. In *ICDE*, pages 183–194, 2011.

[OMSV11] Martin Oberhofer, Albert Maier, Thomas Schwarz, and Manfred Vodegel. Metadata-driven Data Migration for SAP Projects. In *BTW*, 2011.

[OOCR09] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*, pages 237–252, 2009.

[Par11] Roya Parvizi. Inkrementelle Neuberechung mit MapReduce (in German). Bachelor's thesis, TU Kaiserslautern, 2011.

[PBYI09] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. DryadInc: Reusing work in large-scale computations. In *HotCloud*, 2009.

[PD10] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*, 2010.

[PPR+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making Views Self-Maintainable for Data Warehousing. In *PDIS*, pages 158–169, 1996.

[Qua96] Dallan Quass. Maintenance Expressions for Views with Aggregation. In *VIEWS*, pages 110–118, 1996.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.*, 3(3):337–341, 1991.

[Rau05] John Rauscher. Is SQL Becoming the Industry Standard Language for Data Integration? Technical report, Sunopsis, 2005.

[RD00] Erhard Rahm and Hong Hai Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[RL11] Andreas Reuter and Wolfgang Lehner. Realtime Data Warehousing. In *DBTT*, 2011.

[RS97] Mary Tork Roth and Peter M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB*, pages 266–275, 1997.

[SB08] Ricardo Jorge Santos and Jorge Bernardino. Real-time data warehouse loading methodology. In *IDEAS*, pages 49–58, 2008.

[Sch12] Marc Schäfer. Inkrementelle Wartung von Data Cubes mit MapReduce (in German). Bachelor's thesis, TU Kaiserslautern, 2012.

[SF90] Arie Segev and Weiping Fang. Currency-Based Updates to Distributed Materialized Views. In *ICDE*, pages 512–520, 1990.

[She08] Rick Sherman. Back to the Basics of Data Warehousing. *Information Management Magazine*, October 2008.

[Sim03] Alkis Simitsis. Modeling and managing ETL processes. In *VLDB PhD Workshop*, 2003.

[Sim05] Alkis Simitsis. Mapping conceptual to logical models for ETL processes. In *DOLAP*, pages 67–76, 2005.

[SL90] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.

[SP89] Arie Segev and Jooseok Park. Updating Distributed Materialized Views. *IEEE Trans. Knowl. Data Eng.*, 1(2):173–184, 1989.

[SQL] *Microsoft SQL Server 2008*. `http://www.microsoft.com/sqlserver/2008/`.

[SVS05] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. Optimizing ETL Processes in Data Warehouses. In *ICDE*, pages 564–575, 2005.

[TBL09] Maik Thiele, Andreas Bader, and Wolfgang Lehner. Multi-Objective Scheduling for Real-Time Data Warehouses. *Computer Science - R&D*, 24(3):137–151, 2009.

[TL09] Maik Thiele and Wolfgang Lehner. Evaluation of Load Scheduling Strategies for Real-Time Data Warehouse Environments. In *BIRTE*, pages 84–99, 2009.

[TPL08] Christian Thomsen, Torben Bach Pedersen, and Wolfgang Lehner. RiTE: Providing On-Demand Data for Right-Time Data Warehousing. In *ICDE*, pages 456–465, 2008.

[Tra02] Transaction Processing Performance Council. TPC Benchmark W (Web Commerce). Version 1.8, 2002.

[Tra09] Transaction Processing Performance Council. TPC Benchmark H (Decision Support) Revision 2.9.0, 2009.

[TVS07] Vasiliki Tziovara, Panos Vassiliadis, and Alkis Simitsis. Deciding the physical implementation of ETL workflows. In *DOLAP*, pages 49–56, 2007.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.

[VSS02] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual modeling for ETL processes. In *DOLAP*, pages 14–21, 2002.

[WCP09] Len Wyatt, Brian Caufield, and Daniel Pol. Principles for an ETL Benchmark. In *TPCTC*, pages 183–198, 2009.

[WR05] Catharine M. Wyss and Edward L. Robertson. Relational languages for metadata integration. *ACM Trans. Database Syst.*, 30(2):624–660, 2005.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *SIGMOD Conference*, pages 316–327, 1995.

[ZGMW96] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *PDIS*, pages 146–157, 1996.

[ZGMW98] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency Algorithms for Multi-Source Warehouse View Maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998.

[ZSW+06] Xufeng Zhang, Weiwei Sun, Wei Wang, Yahui Feng, and Baile Shi. Generating Incremental ETL Processes Automatically. In *IMSCCS (2)*, pages 516–521, 2006.

Resume

Thomas Jörg – Augustenstraße 65 – 80333 München

Since 4/2012	Software Engineer Google Germany GmbH, München
03/2008 – 04/2012	Scientific Staff Member Heterogeneous Information Systems Group, Department of Computer Science, University of Kaiserslautern
03/2007 – 02/2008	Internship IBM Deutschland Research & Development, IBM Lab Böblingen
10/2001 – 02/2007	Studies of Applied Computer Science University of Kaiserslautern Degree: Dipl.-Inf.
03/2006 – 06/2006	Internship DaimlerChrysler AG, Research and Technology, Ulm
03/2005 – 07/2005	Participant of the AIESEC exchange program Quinnox Consultancy Services Limited, Bombay, India
07/2000 – 04/2001	Military Service, 3. Panzerartillerielehrbataillon 345, Kusel
07/1991 – 06/2000	Albert-Schweitzer-Gymnasium, Kaiserslautern
06/1987 – 06/1991	Grundschule Enkenbach-Alsenborn