

Formalizing ETL Jobs for Incremental Loading of Data Warehouses

Thomas Jörg and Stefan Dessloch
University of Kaiserslautern,
67653 Kaiserslautern, Germany
{joerg|dessloch}@informatik.uni-kl.de

Abstract: Extract-transform-load (ETL) tools are primarily designed for data warehouse loading, i.e. to perform physical data integration. When the operational data sources happen to change, the data warehouse gets stale. To ensure data timeliness, the data warehouse is refreshed on a periodical basis. The naive approach of simply reloading the data warehouse is obviously inefficient. Typically, only a small fraction of source data is changed during loading cycles. It is therefore desirable to capture these changes at the operational data sources and refresh the data warehouse incrementally. This approach is known as *incremental loading*. Dedicated ETL jobs are required to perform incremental loading. We are not aware of any ETL tool that helps to automate this task. In fact, incremental load jobs are handcrafted by ETL programmers so far. The development is thus costly and error-prone.

In this paper we present an approach to the automated derivation of incremental load jobs based on equational reasoning. We review existing Change Data Capture techniques and discuss limitations of different approaches. We further review existing loading facilities for data warehouse refreshment. We then provide *transformation rules* for the derivation of incremental load jobs. We stress that the derived jobs rely on existing Change Data Capture techniques, existing loading facilities, and existing ETL execution platforms.

1 Introduction

The Extract-Transform-Load (ETL) system is the foundation of any data warehouse [KR02, KC04, LN07]. The objective of the ETL system is extracting data from multiple, heterogeneous data sources, transforming and cleansing data, and finally loading data into the data warehouse where it is accessible to business intelligence applications.

The very first population of a data warehouse is referred to as *initial load*. During an initial load, data is typically extracted exhaustively from the sources and delivered to the data warehouse. As source data changes over time, the data warehouse gets stale, and hence, needs to be refreshed. Data warehouse refreshment is typically performed in batch mode on a periodical basis. The naive approach to data warehouse refreshment is referred to as *full reloading*. The idea is to simply rerun the initial load job, collect the resulting data, and compare it to the data warehouse content. In this way, the required changes for data warehouse refreshment can be retrieved. Note that it is impractical to drop and recreate the data warehouse since historic data has to be maintained. Full reloading is

obviously inefficient considering that most often only a small fraction of source data is changed during loading cycles. It is rather desirable to capture source data changes and propagate the mere changes to the data warehouse. This approach is known as *incremental loading*.

In general, incremental loading can be assumed to be more efficient than full reloading. However, the efficiency gain comes at the cost of additional development effort. While ETL jobs for initial loading can easily be reused for reloading the data warehouse, they cannot be applied for incremental loading. In fact, dedicated ETL jobs are required for this purpose, which tend to be far more complex. That is, ETL programmers need to create separate jobs for both, initial and incremental loading. Little advice on the design of incremental load jobs is found in literature and we are not aware of any ETL tool that helps to automate this task.

In this paper we explore the following problem: Given an ETL job that performs initial loading, how can an ETL job be derived that performs incremental loading, is executable on existing ETL platforms, utilizes existing Change Data Capture technologies, and relies on existing loading facilities? Our proposed approach to tackle this problem is based on previous work of ours [JD08]. We elaborate on this approach and provide formal models for ETL data and ETL data transformations. We further provide formal transformation rules that facilitate the derivation of incremental load jobs by equational reasoning.

The problem addressed in this paper is clearly related to maintenance of materialized views. Work in this area, however, is partly based on assumptions that do not hold in data warehouse environments and cannot directly be transferred to this domain. We highlight the differences in Section 2 on related work. We review existing techniques with relevance to the ETL system in Section 3. We then introduce a formal model for the description of common ETL transformation capabilities in Section 4. Based on this model we describe the derivation of incremental load jobs from initial load jobs based on equivalence preserving transformation rules in Section 5 and conclude in Section 6.

2 Related Work

ETL has received considerable attention in the data integration market; numerous commercial ETL tools are available today [DS, OWB, DWE, IPC]. According to Ralph Kimball seventy percent of the resources needed for the implementation and maintenance of a data warehouse are typically consumed by the ETL system [KC04]. However, the database research community did not give ETL the attention that it received from commercial vendors so far. We present academic efforts in this area in Section 2.1. We then discuss work on the maintenance of materialized views. This problem is clearly related to the problem of loading data warehouses incrementally. However, approaches to the maintenance of materialized views are based on assumptions that do not hold in data warehouse environments. We discuss the differences in Section 2.2.

2.1 Related Work on ETL

An extensive study on the modeling of ETL jobs is by Simitsis, Vassiliadis, et al. [VSS02, SVTS05, Sim05, SVS05]. The authors propose both, a conceptual model and a logical model for the representation of ETL jobs. The conceptual model maps attributes of the data sources to the attributes of the data warehouse tables. The logical model describes the data flow from the sources towards the data warehouse. In [Sim05] a method for the translation of conceptual models to logical models is presented. In [SVS05] the optimization of logical model instances with regard to their execution cost is discussed. However, data warehouse refreshment and, in particular, incremental loading has not been studied.

In [AN08] the vision of a generic approach for ETL management is presented. The work is inspired by research on generic model management. The authors introduce a set of high-level operators for ETL management tasks and recognize the need for a platform- and tool-independent model for ETL jobs. However, no details of such a model are provided.

Most relevant to our work is the Orchid project [DHW⁺08]. The Orchid system facilitates the conversion from schema mappings to executable ETL jobs and vice versa. Schema mappings capture correspondences between source and target schema items in an abstract manner. Business users well understand these correspondences but usually do not have the technical skills to design appropriate ETL jobs.

Orchid translates schema mappings into executable ETL jobs in a two-step process. First, the schema mappings are translated into an intermediate representation referred to as Operator Hub Model (OHM) instance. The OHM captures the transformation semantics common to both, schema mappings and ETL jobs. Second, the OHM instance is translated into an ETL job tailored for the ETL tool of choice. However, jobs generated by Orchid are suited for initial loading (or full reloading) only. We adopt Orchid's OHM to describe ETL jobs in an abstract manner and contribute an approach to derive incremental load jobs from initial load jobs. By doing so, we can reuse the Orchid system for the deployment of ETL jobs. We thus complement Orchid with the ability to create ETL jobs for incremental loading.

2.2 Related Work on Maintenance of Materialized Views

Incremental loading of data warehouses is related to incremental maintenance of materialized views, because, in either case, physically integrated data is updated incrementally. Approaches for maintaining materialized views are, however, not directly applicable to data warehouse environments for several reasons.

- Approaches for maintaining materialized views construct maintenance expressions in response to changes at transaction commit time [GL95, QW91, GMS93]. Maintenance expressions assume access to the unchanged state of all base relations and the net changes of the committing transactions. ETL jobs for incremental loading, in contrast, operate in batch mode for efficiency reasons and are repeated on a peri-

odical basis. Hence, any possible change has to be anticipated. More importantly, source data is only available in its most current state unless the former state is explicitly preserved.

- A materialized view and its source relations are managed by the same database system. In consequence, full information about changes to source relations is available. In data warehouse environments Change Data Capture techniques are applied that may suffer from limitations and miss certain changes as stated in Section 3.2. In Section 4 we will introduce a notion of *partial change data* to cope with this situation. We are not aware of any other approach that considers limited access to change data.
- In literature, a data warehouse is sometimes regarded as a set of materialized views defined on distributed data sources [AASY97, ZGMHW95, Yu06, QGMW96]. We argue that this notion disregards an important aspect of data warehousing since the data warehouse keeps a history of data changes. Materialized views, in contrast, reflect the most current state of their base relations only. Change propagation approaches in the context of materialized views typically distinguish two types of data modifications, i.e. insertions and deletions [AASY97, GL95, GMS93, QW91, ZGMHW95, BLT86, CW91]. Updates are treated as a combination of both, insertions and deletions. The initial state of updated tuples is propagated as deletion while the current state of updated tuples is propagated as insertion. Materialized views are maintained by at first performing deletions and subsequently performing insertions that have been propagated from the sources. Data warehouse dimensions, however, keep a history of data changes. In consequence, deleting and reinserting tuples will not lead to the same result as updating tuples in place, in terms of the data history. Therefore, our approach to incremental data warehouse loading handles updates separately from insertions and deletions.

3 The ETL Environment

In this Section we review existing techniques that shape the environment of the ETL system and are thus relevant for ETL job design. We first introduce the dimensional modeling methodology that dictates data warehouse design. We then discuss techniques with relevance to incremental loading. First, we review so called Change Data Capture techniques that allow for detecting source data changes. Second, we describe loading facilities for data warehouses. In absence of an established term, we refer to these techniques as Change Data Application techniques.

3.1 Dimensional Modeling

Dimensional modeling is an established methodology for data warehouse design and is widely used in practice [KR02]. The dimensional modeling methodology dictates both,

the logical schema design of the data warehouse and the strategy for keeping the history of data changes. Both parts are highly relevant to the design of ETL jobs.

A database schema designed according to the rules of dimensional modeling is referred to as a *star schema* [KR02]. A star schema is made up of so called fact tables and dimension tables. Fact tables store measures of business processes that are referred to as *facts*. Facts are usually numeric values that can be aggregated. Dimension tables contain rich textual descriptions of the business entities. Taking a retail sales scenario as an example, facts may represent sales transactions and provide measures like the sales quantity and dollar sales amount. Dimensions may describe the product being sold, the retail store where it was purchased, 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 typically are not in third normal form. In fact, dimensions often represent multiple hierarchical relationships in a single table. Products roll up into brands and then into categories, for instance. Information about products, brands, and categories is typically stored within the same dimension table of a star schema. That is, dimension tables are highly denormalized. The design goals are query performance, user understandability, and resilience to changes that come at the cost of data redundancy.

In addition to schema design, the dimensional modeling methodology dictates techniques for keeping the history of data changes. These techniques go by the name of *Slowly Changing Dimensions* [KR02, KC04]. The basic idea is to add a so called *surrogate key column* to dimension tables. Facts reference dimensions using the surrogate key to establish a foreign key relationship. Surrogate keys are exclusively controlled by the data warehouse. Their sole purpose is making dimension tuples uniquely identifiable while their value is meaningless by definition.

Operational data sources typically manage primary keys referred to as *business keys*. It is common to assign a unique number to each product in stock, for instance. Business keys are not replaced by surrogate keys. In fact, both, the business key and the surrogate key are included in the dimension table. In this way, dimension tuples can easily be traced back to the operational sources. This ability is known as *data lineage*.

When a new tuple is inserted into the dimension table a fresh surrogate key is assigned. The more interesting case occurs when a dimension tuple is updated. Different actions may be taken depending on the particular columns that have been modified. Changing a product name, for example, could be considered as an error correction; hence the corresponding name in the data warehouse is simply overwritten. This case is referred to as Slowly Changing Dimensions Type I. Increasing the retail price, in contrast, is likely considered to be a normal business activity and a history of retail prices is kept in the data warehouses. This case is referred to as Slowly Changing Dimensions Type II. The basic idea is to leave the outdated tuple in place and create a new one with the current data. The warehouse assigns a fresh surrogate key to the created tuple to distinguish it from its expired versions. Note that the complete history can be retrieved by means of the business key that remains constant throughout time. Besides the surrogate key there are further “special purpose” columns potentially involved in the update process. For instance, an effective timestamp and an expiration timestamp may be assigned, tuples may hold a

reference to their preceding version, and a flag may be used to indicate the most current version.

In summary, the dimensional modeling methodology impacts the ETL job design in two ways. First, it dictates the shape of the target schema, and more importantly it requires the ETL job to handle “special purpose” columns in the correct manner. We emphasize that the choice for Slowly Changing Dimensions Type I or II requires knowledge of the initial state and the current state of updated tuples.

3.2 Change Data Capture

Change Data Capture (CDC) is a generic term for techniques that monitor operational data sources with the objective of detecting and capturing data changes of interest [KC04, BT98]. CDC is of particular importance for data warehouse maintenance. With CDC techniques in place, the data warehouse can be maintained by propagating changes captured at the sources. CDC techniques applied in practice roughly follow three main approaches, namely log-based CDC, utilization of audit columns, and calculation of snapshot differentials [KC04, BT98, LGM96].

Log-based CDC techniques parse system logs and retrieve changes of interest. These techniques are typically employed in conjunction with database systems. Virtually all database systems record changes in transaction logs. This information can be leveraged for CDC. Alternatively, changes may be explicitly recorded using database triggers or application logic for instance.

Operational data sources often employ so called *audit columns*. Audit columns are appended to each tuple and indicate the time at which the tuple was modified for the last time. Usually timestamps or version numbers are used. Audit columns serve as the selection criteria to extract changes that occurred since the last incremental load process. Note that deletions remain undetected.

The *snapshot differential* technique is most appropriate for data that resides in unsophisticated data sources such as flat files or legacy applications. The latter typically offer mechanisms for dumping data into files but lack advanced query capabilities. In this case, changes can be inferred by comparing a current source snapshot with a snapshot taken at a previous point in time. A major drawback of the snapshot differential approach is the need for frequent extractions of large data volumes. However, it is applicable to virtually any type of data source.

The above mentioned CDC approaches differ not only in their technical realization but also in their ability to detect changes. We refer to the inability to detect certain types of changes as *CDC limitation*. As mentioned before deletions cannot be detected by means of audit columns. Often a single audit column is used to record the time of both, record creation and modification. In this case insertions and updates are indistinguishable with respect to CDC. Another limitation of the audit columns approach is the inability to retrieve the initial state of records that have been updated. Interestingly, existing snapshot differential implementations usually have the same limitation. They do not provide the initial state

of updated records while this would be feasible in principle, since the required data is available in the snapshot taken during the previous run. Log-based CDC approaches in practice typically capture all types of changes, i.e. insertions, deletions, and the initial and current state of updated records.

3.3 Change Data Application

We use the term *Change Data Application* to refer to any technique appropriate for updating the data warehouse content. Typically a database management system is used to host the data warehouse. Hence, available CDA techniques are DML statements issued via the SQL interface, proprietary bulk load utilities, or loading facilities provided by ETL tools. Note that CDA techniques have different requirements with regard to their input data.

Consider the SQL MERGE statement¹. The MERGE statement inserts or updates tuples; the choice depends on the tuples existing in the target table, i.e. if there is no existing tuple with equal primary key values the new tuple is inserted; if such a tuple is found it is updated. There are bulk load utilities and ETL loading facilities that are equally able to decide whether a tuple is to be inserted or updated depending on its primary key. These techniques ping the target table to determine the appropriate operation. Note that the input to any of these techniques is a single dataset that contains tuples to be inserted and updated in a joint manner.

From a performance perspective, ETL jobs should explicitly separate data that is to be updated from data that is to be inserted to eliminate the need for frequent table lookups [KC04]. The SQL INSERT and UPDATE statements do not incur this overhead. Again, there are bulk load utilities and ETL loading facilities with similar properties. Note that these CDA techniques differ in their requirements from the ones mentioned before. They expect separate datasets for insertions and updates. However, deletions have to be separated in either case to be applied to the target dataset using SQL DELETE statements, for instance.

In Section 3.1 we introduced the Slowly Changing Dimensions (SCD) approach that is the technique of choice to keep a history of data changes in the data warehouse. Recall, that the decision whether to apply SCD strategy type I or type II depends on the columns that have been updated within a tuple. That is, one has to consider both, the initial state and the current state of updated tuples. Hence, SCD techniques need to be provided with both datasets unless the costs for frequent dimension lookups are acceptable.

In summary, CDA techniques differ in their *requirements* with regard to input change data very much like CDC techniques face varying limitations. While some CDA techniques are able to process insertions and updates provided in an indistinguishable manner, others demand for separated data sets. Sophisticated CDA techniques such as SCD demand for both, the initial state and the current state of updates tuples.

¹The MERGE statement (also known as upsert) has been introduced with the SQL:2003 standard.

4 Modeling ETL Jobs

To tackle the problem of deriving incremental load jobs we introduce a model for ETL jobs in this section. We first specify a model for data and data changes and afterwards provide a model for data transformations. We adopt the relational data model here since the vast majority of operational systems organize data in a structured manner². In the following we use the term *relation* to refer to any structured datasets. We do not restrict this term to relational database tables but include other structured datasets such as flat files, for instance.

Formally, relations are defined as follows. Let \mathcal{R} be a set of relation names, \mathcal{A} a set of attribute names, and \mathcal{D} a set of domains, i.e. sets of atomic values. A relation is defined by a relation name $R \in \mathcal{R}$ along with a relation schema. A relation schema is a list of attribute names and denoted by $sch(R) = (A_1, A_2, \dots, A_n)$ with $A_i \in \mathcal{A}$. We use the function $dom : \mathcal{A} \rightarrow \mathcal{D}$ to map an attribute name to its domains. The domain of a relation R is defined as the Cartesian product of its attributes' domains and denoted by $dom(R) := dom(A_1) \times dom(A_2) \times \dots \times dom(A_n)$ with $A_i \in sch(R)$. The data content of a relation is referred to as the relation's state. The state r of a relation R is a subset of its domain, i.e. $r \subseteq dom(R)$. Note that the state of a relation may change over time as data is modified. In the following we use r_{new} and r_{old} to denote the current state of a relation and the state at the time of the previous incremental load, respectively.

During incremental loading the data changes that occurred at operational data sources are propagated towards the data warehouse. We introduce a formal model for the description of data changes referred to as *change data*. Change data specifies how the state of an operational data source has changed during one loading cycle. Change data is captured at tuple granularity and consists of four sets of tuples namely, the set of inserted tuples (insert), the set of deleted tuples (delete), the set of updated tuples in their current state (update new), and the set of updated tuples in their initial state (update old). We stress that change data serves as both, the model for the output of CDC techniques and the model for the input of CDA techniques. Below, we provide a formal definition of change data. We make use of the relational algebra projection operator denoted by π .

Definition 4.1. Given relation R , the current state r_{new} of R , the previous state r_{old} of R , the list of attribute names $S := sch(R)$, and the list of primary key attribute names $K \subseteq sch(R)$, *change data* is a four-tuple $(r_{ins}, r_{del}, r_{un}, r_{uo})$ such that

$$r_{ins} := \{s \mid s \in r_{new} \wedge t \in r_{old} \wedge \pi_K(s) \neq \pi_K(t)\} \quad (4.1)$$

$$r_{del} := \{s \mid s \in r_{old} \wedge t \in r_{new} \wedge \pi_K(s) \neq \pi_K(t)\} \quad (4.2)$$

$$r_{un} := \{s \mid s \in r_{new} \wedge t \in r_{old} \wedge (\pi_K(s) = \pi_K(t) \rightarrow \pi_{S \setminus K}(s) \neq \pi_{S \setminus K}(t))\} \quad (4.3)$$

$$r_{uo} := \{s \mid s \in r_{old} \wedge t \in r_{new} \wedge (\pi_K(s) = \pi_K(t) \rightarrow \pi_{S \setminus K}(s) \neq \pi_{S \setminus K}(t))\}. \quad (4.4)$$

²Data in the semi-structured XML format with relevance to data warehousing is typically data-centric, i.e. data is organized in repeating tree structures, mixed content is avoided, the document order does not contribute to the semantics, and a schema definition is available. The conversion of data-centric XML into a structured representation is usually straightforward.

In order to express changes of relations we introduce two partial set operators, namely the disjoint union and the contained difference.

- Let R and S be relations. The *disjoint union* $R \oplus S$ is equal to the set union $R \cup S$ if R and S are disjoint ($R \cap S = \emptyset$). Otherwise it is not defined.
- The *contained difference* $R \ominus S$ is equal to the set difference $R \setminus S$ if S is a subset of R . Otherwise it is not defined.

In an ETL environment the state of a relation at the time of the previous incremental load is typically not available. This state can however be calculated given the current state and change data.

Theorem 4.2. *Given relation R , the current state r_{new} of R , the previous state r_{old} of R , and change data r_{ins} , r_{del} , r_{un} , r_{uo} , the following equations hold*

$$r_{\text{new}} \ominus r_{\text{ins}} \ominus r_{\text{un}} = r_{\text{new}} \cap r_{\text{old}} \quad (4.5)$$

$$r_{\text{new}} \cap r_{\text{old}} \oplus r_{\text{del}} \oplus r_{\text{uo}} = r_{\text{old}} \quad (4.6)$$

$$r_{\text{new}} = r_{\text{old}} \ominus r_{\text{del}} \ominus r_{\text{uo}} \oplus r_{\text{ins}} \oplus r_{\text{un}}. \quad (4.7)$$

Proof. We show the correctness of 4.5. Equation 4.6 can be shown in a similarly way. Equation 4.7 follows from 4.5 and 4.6.

$$\begin{aligned} r_{\text{new}} \ominus r_{\text{ins}} \ominus r_{\text{un}} &= \\ r_{\text{new}} \ominus (r_{\text{ins}} \oplus r_{\text{un}}) &= \\ r_{\text{new}} \ominus \{s \mid s \in r_{\text{new}} \wedge t \in r_{\text{old}} \wedge (\pi_K(s) \neq \pi_K(t) \vee \pi_{S \setminus K}(s) \neq \pi_{S \setminus K}(t))\} &= \\ r_{\text{new}} \ominus \{s \mid s \in r_{\text{new}} \wedge t \in r_{\text{old}} \wedge s \neq t\} &= \\ r_{\text{new}} \ominus (r_{\text{new}} \setminus r_{\text{old}}) &= r_{\text{new}} \cap r_{\text{old}} \end{aligned}$$

□

The survey of CDC in Section 3.2 revealed that CDC techniques may suffer from limitations. While Log-based CDC techniques provide complete change data, both the snapshot differential and audit column approaches are unable to capture certain types of changes. We refer to the resulting change data as *partial* change data. Existing snapshot differential implementations often do not capture the initial state of updated tuples (update old). Consequently, this data is unavailable for change propagation; with regard to relation R we say that r_{uo} is unavailable. The audit column approach misses deletions and the initial state of updated records, i.e. r_{del} and r_{uo} are unavailable. In case a single audit column is used to record the time of tuple insertions and subsequent updates, insertions and updates cannot be distinguished. Thus, the best the CDC technique can provide is $r_{\text{ups}} := r_{\text{ins}} \oplus r_{\text{un}}$. Tuples that have been inserted or updated since the previous incremental load are jointly provided within a single dataset (upsert). Note that neither r_{ins} nor r_{un} are available though.

We further stressed in Section 3.3 that CDA techniques differ in their requirements with regard to change data. There are CDA techniques capable of consuming upsert sets, for

example. These techniques ping the data warehouse to decide whether to perform insertions or updates. Other techniques need to be provided with separated sets of insertions and updates but work more efficiently. Advanced techniques may additionally require updated tuples in their initial state. In summary, the notion of partial change data is key to the description of both, the output of CDC techniques and the input of CDA techniques. Below, we provide a definition of partial change data.

Definition 4.3. Given a relation R we refer to change data as *partial change data* if at least one component r_{ins} , r_{del} , r_{un} , or r_{uo} is unavailable.

Having established a model for data and change data we are still in need of a model for data transformations. While all major database management systems adhere to the SQL standard, no comparable standard exists in the area of ETL. In fact, ETL tools make use of proprietary scripting languages or visual user interfaces. We adopt OHM [DHW⁺08] to describe the transformational part of ETL jobs in a platform-independent manner. The OHM is based on a thorough analysis of ETL tools and captures common transformation capabilities. Roughly speaking, OHM operators are generalizations of relational algebra operators. We consider a subset of OHM operators, namely projection, selection, union, and join, which are described in-depth in Section 5.1, Section 5.2, Section 5.3, and Section 5.4, respectively.

Definition 4.4. An *ETL transformation expression* \mathcal{E} (or *ETL expression* for short) is generated by the grammar G with the following production rules.

$\mathcal{E} ::= R$	Relation name from \mathcal{R}
$\sigma_p(\mathcal{E})$	Selection
$\pi_A(\mathcal{E})$	Projection
$\mathcal{E} \oplus \mathcal{E}$	Disjoint union
$\mathcal{E} \ominus \mathcal{E}$	Contained difference
$\mathcal{E} \bowtie \mathcal{E}$	Join

We impose restrictions on the projection and the union operator, i.e. we consider key-preserving projections and key-disjoint unions only. We show that these restrictions are justifiable in the section on the respective operator. In contrast to [DHW⁺08] we favor an equational representation of ETL transformations over a graph-based representation. Advanced OHM operators for aggregation and restructuring of data are ignored for the moment and left for future work.

5 Incremental Loading

The basic idea of incremental loading is to infer changes required to refresh the data warehouse from changes captured at the data sources. The benefits of incremental loading as

compared to full reloading are twofold. First, the volume of changed data at the sources is typically very small compared to the overall data volume, i.e. less data needs to be extracted. Second, the vast majority of data within the warehouse remains untouched during incremental loading, since changes are only applied where necessary.

We propose a change propagation approach to incremental loading. That is, we construct an ETL job that inputs change data captured at the sources, transforms the change data, and ultimately outputs change data that specifies the changes required within the data warehouse. We refer to such an ETL job as *change data propagation job* or CDP job for short. We stress that CDP jobs are essentially conventional ETL jobs in the sense that they are executable on existing ETL platforms. Furthermore existing CDC techniques are used to provide the input change data and existing CDA techniques are used to refresh the data warehouse.

Recall that CDC techniques may have limitations and hence may provide partial change data (see Section 3.2). Similarly, CDA techniques have varying requirements with regard to change data they can consume (see Section 3.3). The input of CDP jobs is change data provided by CDC techniques; the output is again change data that is fed into CDA techniques. Thus, CDP jobs need to cope with CDC limitations and satisfy CDA requirements at the same time. This is, however, not always possible. Two questions arise. Given CDA requirements, what CDC limitations are acceptable? Or the other way round, given CDC limitations, what CDA requirements are satisfiable?

In the remainder of this section we describe our approach to derive CDP jobs from ETL jobs for initial loading. We assume that the initial load job is given as an ETL expression \mathcal{E} as defined in 4.4. We are thus interested in deriving ETL expressions \mathcal{E}_{ins} , \mathcal{E}_{del} , \mathcal{E}_{un} , and \mathcal{E}_{uo} that propagate insertions, deletions, the current state of updated tuples, and the initial state of updated tuples, respectively. We refer to these expressions as CDP expressions. In combination, CDP expressions form a CDP job³. Note that \mathcal{E}_{ins} , \mathcal{E}_{del} , \mathcal{E}_{un} , or \mathcal{E}_{uo} may depend on source change data that is unavailable due to CDC limitations. That is, the propagation of insertions, deletions, updated tuples in their current state, or updated tuples in their initial state may not be possible, thus, the overall CDP job may output *partial* change data. In this situation one may choose a CDA technique suitable for partial change data, migrate to more powerful CDC techniques, or simply refrain from propagating the unavailable changes. The dimensional modeling methodology, for instance, proposes to leave dimension tuples in place after the corresponding source tuples have been deleted. Hence, deletions can safely be ignored with regard to change propagation in such an environment.

For the derivation of CDP expressions we define a set of functions $f_{\text{ins}} : L(G) \rightarrow L(G)$, $f_{\text{del}} : L(G) \rightarrow L(G)$, $f_{\text{un}} : L(G) \rightarrow L(G)$, and $f_{\text{uo}} : L(G) \rightarrow L(G)$ that map an ETL expression \mathcal{E} to the CDP expressions \mathcal{E}_{ins} , \mathcal{E}_{del} , \mathcal{E}_{un} , and \mathcal{E}_{uo} , respectively. These functions can however not directly be evaluated. Instead, we define equivalence preserving transformation rules that “push” these functions into the ETL expression and thereby transform the expression appropriately. After repeatedly applying applicable transformation rules, these functions will eventually take a relation name as input argument instead

³CDP expressions typically share common subexpressions, hence it is desirable to combine them into a single ETL job.

of a complex ETL expression. At this point, the function simply denotes the output of the relation's CDC system.

We provide transformation rules for projection, selection, union, and join in the subsequent sections and explain the expression derivation process by an example in Section 5.5.

5.1 Key-preserving Projection

Our notion of an ETL projection operator generalizes the classical relational projection. Besides dropping columns, we allow for adding and renaming columns, assigning constant values, and performing value conversions. The latter may include sophisticated transformations, such as parsing and splitting up free-form address fields. Our only assumptions are that any value transformation function processes a single tuple at a time and computes its output in a deterministic manner.

We further restrict our considerations to *key-preserving* projections since dropping key columns is generally unwanted during ETL processing [KR02, KC04]. We highlighted the importance of business keys in Section 3.1 on dimensional modeling. Business keys are the primary means of maintaining data lineage. Moreover, in the absence of business keys, it is impractical to keep a history of data changes at the data warehouse. Hence, business keys must not be dropped by the ETL job. Whenever the ETL job joins data from multiple sources, the business key is composed of (a subset of) the key columns of the source datasets. Consider that the key columns of at least one dataset have been dropped. In this case, the join result will just as well lack key columns. That is, the propagation of business keys renders impossible if at any step of the ETL job a dataset without key columns is produced. We thus exclude any non-key-preserving projection from our considerations. For this reason implicit duplicate elimination is never performed by the projection operator.

In view of the above considerations we formulate the following transformation rules. The proofs are straightforward and therefore omitted.

$$f_{\text{ins}}(\pi_A(\mathcal{E})) \rightsquigarrow \pi_A(f_{\text{ins}}(\mathcal{E})) \quad (5.1)$$

$$f_{\text{del}}(\pi_A(\mathcal{E})) \rightsquigarrow \pi_A(f_{\text{del}}(\mathcal{E})) \quad (5.2)$$

$$f_{\text{un}}(\pi_A(\mathcal{E})) \rightsquigarrow \pi_A(f_{\text{un}}(\mathcal{E})) \quad (5.3)$$

$$f_{\text{uo}}(\pi_A(\mathcal{E})) \rightsquigarrow \pi_A(f_{\text{uo}}(\mathcal{E})) \quad (5.4)$$

$$f_{\text{ups}}(\pi_A(\mathcal{E})) \rightsquigarrow \pi_A(f_{\text{ups}}(\mathcal{E})) \quad (5.5)$$

$$f_{\text{new}}(\pi_A(\mathcal{E})) \rightsquigarrow \pi_A(f_{\text{new}}(\mathcal{E})) \quad (5.6)$$

5.2 Selection

The selection operator filters those tuples for which a given boolean predicate p holds and discards all others. In the light of change propagation three cases are to be distinguished.

- Inserts that satisfy the filter predicate are propagated while inserts that do not satisfy the filter predicate are dropped.
- Deletions that satisfy the filter predicate are propagated since the tuple to delete has been propagated towards the data sink before. Deletions that do not satisfy the filter predicate lack this counterpart and are dropped.
- Update pairs, i.e. the initial and the current value of updated tuples, are propagated if both values satisfy the filter predicate and dropped if neither of them does so. In case only the current value satisfies the filter predicate while the initial value does not, the update turns into an insertion. The updated tuple has not been propagated towards the data sink before and hence is to be created. Similarly, an update turns into a deletion if the initial value satisfies the filter predicate while the current value fails to do so.

If the initial value of updated records is unavailable due to partial change data it cannot be checked against the filter predicate. Thus, it is not possible to conclude whether the updated source tuple has been propagated to the data sinks before. It is thus unclear whether an insert or an update is to be issued, provided that the current value of the updated tuple satisfies the filter predicate. Hence, inserts and updates can only be propagated in a joint manner in this situation (see 5.11).

We formulate the following transformation rules for the select operator. We make use of the semijoin operator denoted by \ltimes and the antijoin operator denoted by $\overline{\ltimes}$ known from relational algebra.

$$f_{\text{ins}}(\sigma_p(\mathcal{E})) \rightsquigarrow \sigma_p(f_{\text{ins}}(\mathcal{E})) \oplus [\sigma_p(f_{\text{un}}(\mathcal{E})) \overline{\ltimes} \sigma_p(f_{\text{uo}}(\mathcal{E}))] \quad (5.7)$$

$$f_{\text{del}}(\sigma_p(\mathcal{E})) \rightsquigarrow \sigma_p(f_{\text{del}}(\mathcal{E})) \oplus [\sigma_p(f_{\text{uo}}(\mathcal{E})) \overline{\ltimes} \sigma_p(f_{\text{un}}(\mathcal{E}))] \quad (5.8)$$

$$f_{\text{un}}(\sigma_p(\mathcal{E})) \rightsquigarrow \sigma_p(f_{\text{un}}(\mathcal{E})) \ltimes \sigma_p(f_{\text{uo}}(\mathcal{E})) \quad (5.9)$$

$$f_{\text{uo}}(\sigma_p(\mathcal{E})) \rightsquigarrow \sigma_p(f_{\text{uo}}(\mathcal{E})) \ltimes \sigma_p(f_{\text{un}}(\mathcal{E})) \quad (5.10)$$

$$f_{\text{ups}}(\sigma_p(\mathcal{E})) \rightsquigarrow \sigma_p(f_{\text{ups}}(\mathcal{E})) \quad (5.11)$$

$$f_{\text{new}}(\sigma_p(\mathcal{E})) \rightsquigarrow \sigma_p(f_{\text{new}}(\mathcal{E})) \quad (5.12)$$

Proof. We show the correctness of (5.7). We make use of the fact that $\pi_K(\mathcal{E}_{\text{ins}})$ and $\pi_K(\mathcal{E}_{\text{old}})$ are disjoint and \mathcal{E}_{un} shares common keys with the updated tuples $\mathcal{E}_{\text{uo}} \subseteq \mathcal{E}_{\text{old}}$ only. Equation (5.8), (5.9), and (5.10) can be shown in a similar way.

$$\begin{aligned} f_{\text{ins}}(\sigma_p(\mathcal{E})) &\stackrel{4.1}{=} \\ \{r \mid r \in \sigma_p(\mathcal{E}_{\text{new}}) \wedge s \in \sigma_p(\mathcal{E}_{\text{old}}) \wedge \pi_K(r) \neq \pi_K(s)\} &\stackrel{4.7}{=} \\ \{r \mid r \in \sigma_p(\mathcal{E}_{\text{old}} \ominus \mathcal{E}_{\text{del}} \ominus \mathcal{E}_{\text{uo}} \oplus \mathcal{E}_{\text{ins}} \oplus \mathcal{E}_{\text{un}}) \wedge s \in \sigma_p(\mathcal{E}_{\text{old}}) \wedge \pi_K(r) \neq \pi_K(s)\} &= \\ \{r \mid r \in \sigma_p(\mathcal{E}_{\text{old}} \ominus \mathcal{E}_{\text{del}} \ominus \mathcal{E}_{\text{uo}}) \wedge s \in \sigma_p(\mathcal{E}_{\text{old}}) \wedge \pi_K(r) \neq \pi_K(s)\} \oplus & \\ \{r \mid r \in \sigma_p(\mathcal{E}_{\text{ins}}) \wedge s \in \sigma_p(\mathcal{E}_{\text{old}}) \wedge \pi_K(r) \neq \pi_K(s)\} \oplus & \\ \{r \mid r \in \sigma_p(\mathcal{E}_{\text{un}}) \wedge s \in \sigma_p(\mathcal{E}_{\text{old}}) \wedge \pi_K(r) \neq \pi_K(s)\} = & \\ \emptyset \oplus \sigma_p(\mathcal{E}_{\text{ins}}) \oplus (\sigma_p(\mathcal{E}_{\text{un}}) \overline{\ltimes} \sigma_p(\mathcal{E}_{\text{uo}})) & \end{aligned}$$

□

We omit proofs for (5.11) and (5.12).

5.3 Key-disjoint Union

We already emphasized the importance of business keys for maintaining data lineage and keeping a history of data changes at the data warehouse. Business keys must “survive” the ETL process and we therefore restrict our considerations to key-preserving unions. That is, the result of a union operation must include unique key values. In consequence we require the source relations to be disjoint with regard to their key values. This property can be achieved by prefixing each key value with a unique identifier of its respective data source. The key-disjoint union is a special case of the disjoint union.

$$f_{\text{ins}}(\mathcal{E} \oplus \mathcal{F}) \rightsquigarrow f_{\text{ins}}(\mathcal{E}) \oplus f_{\text{ins}}(\mathcal{F}) \quad (5.13)$$

$$f_{\text{del}}(\mathcal{E} \oplus \mathcal{F}) \rightsquigarrow f_{\text{del}}(\mathcal{E}) \oplus f_{\text{del}}(\mathcal{F}) \quad (5.14)$$

$$f_{\text{un}}(\mathcal{E} \oplus \mathcal{F}) \rightsquigarrow f_{\text{un}}(\mathcal{E}) \oplus f_{\text{un}}(\mathcal{F}) \quad (5.15)$$

$$f_{\text{uo}}(\mathcal{E} \oplus \mathcal{F}) \rightsquigarrow f_{\text{uo}}(\mathcal{E}) \oplus f_{\text{uo}}(\mathcal{F}) \quad (5.16)$$

$$f_{\text{ups}}(\mathcal{E} \oplus \mathcal{F}) \rightsquigarrow f_{\text{ups}}(\mathcal{E}) \oplus f_{\text{ups}}(\mathcal{F}) \quad (5.17)$$

$$f_{\text{new}}(\mathcal{E} \oplus \mathcal{F}) \rightsquigarrow f_{\text{new}}(\mathcal{E}) \oplus f_{\text{new}}(\mathcal{F}) \quad (5.18)$$

5.4 Join

The majority of transformation rules seen so far map input change data to output change data of the same type. Exceptions are (5.7), (5.8), (5.9), (5.10) on the selection operator. Here updates may give rise to insertions and deletions depending on the evaluation of the filter predicate. The situation is somewhat more complex for the join operator. We need to distinguish between one-to-many and many-to-many joins. Consider two (derived) relations \mathcal{E} and \mathcal{F} involved in a foreign key relationship where \mathcal{E} is the referencing relation and \mathcal{F} is the referenced relation. Say, \mathcal{E} and \mathcal{F} are joined using the foreign key of \mathcal{E} and the primary key of \mathcal{F} . Then tuples in $\mathcal{E} \bowtie \mathcal{F}$ are functionally dependent on \mathcal{E} 's primary key. In particular, for each tuple in \mathcal{E}_{un} at most one join partner is found in \mathcal{F} even if the foreign key column has been updated. It is therefore appropriate to propagate updates. The situation is different for many-to-many joins, i.e. no column in the join predicate is unique. Here, multiple new join partners may be found in response to an update of a column in the join predicate. Moreover, multiple former join partners may be lost at the same time. In consequence, multiple insertions are propagated, one for each new join partner and multiple deletions are propagated, one for each lost join partner. The number of insertions and deletions may differ.

	$E_{new} \cap E_{old}$	E_{ins}	E_{del}	E_{un}	E_{uo}
$F_{new} \cap F_{old}$		ins	del	un	uo
F_{ins}		ins		un	
F_{del}			del		uo
F_{un}	un	ins		un	
F_{uo}	uo		del		uo

Figure 1: Matrix Representation 1-to-n Joins

The matrix shown in Figure 1 depicts the interdependencies between change data of the referencing relation \mathcal{E} and the referenced relation \mathcal{F} assuming referential integrity. The matrix is filled in as follows. The column headers and row headers denote change data of relation \mathcal{E} and \mathcal{F} , respectively. From left to right, the column headers denote the unmodified tuples of \mathcal{E} ($\mathcal{E}_{new} \cap \mathcal{E}_{old}$), insertions \mathcal{E}_{ins} , deletions \mathcal{E}_{del} , updated records in their current state \mathcal{E}_{un} , and updated records in their initial state \mathcal{E}_{uo} . The row headers are organized in a similar way from top to bottom. Columns and rows intersect in cells. Each cell represents the join of two datasets indicated by the row and column headers. The caption of a cell shows whether the join contributes to either $f_{ins}(\mathcal{E} \bowtie \mathcal{F})$, $f_{del}(\mathcal{E} \bowtie \mathcal{F})$, $f_{un}(\mathcal{E} \bowtie \mathcal{F})$, $f_{uo}(\mathcal{E} \bowtie \mathcal{F})$, or to none of them (empty cell). Take the \mathcal{E}_{ins} column as an example. The column contains five cells out of which three are labeled as insertions and two are empty. That means the three joins $\mathcal{E}_{ins} \bowtie (\mathcal{F}_{new} \cap \mathcal{F}_{old})$, $\mathcal{E}_{ins} \bowtie \mathcal{F}_{ins}$, and $\mathcal{E}_{ins} \bowtie \mathcal{F}_{un}$ contribute to $f_{ins}(\mathcal{E} \bowtie \mathcal{F})$. Since no other cells are labeled as insertions $f_{ins}(\mathcal{E} \bowtie \mathcal{F})$ is given by the union of these joins.

$$\begin{aligned}
f_{ins}(\mathcal{E} \bowtie \mathcal{F}) &= \mathcal{E}_{ins} \bowtie (\mathcal{F}_{new} \cap \mathcal{F}_{old}) \oplus \mathcal{E}_{ins} \bowtie \mathcal{F}_{ins} \oplus \mathcal{E}_{ins} \bowtie \mathcal{F}_{un} \stackrel{4,5}{=} \\
&\mathcal{E}_{ins} \bowtie (\mathcal{F}_{new} \ominus \mathcal{F}_{ins} \ominus \mathcal{F}_{un}) \oplus \mathcal{E}_{ins} \bowtie \mathcal{F}_{ins} \oplus \mathcal{E}_{ins} \bowtie \mathcal{F}_{un} = \\
&\mathcal{E}_{ins} \bowtie \mathcal{F}_{new}
\end{aligned}$$

Note, that the above considerations lead to the transformation rule 5.19. The other transformation rules are derived in a similar way.

$$f_{ins}(\mathcal{E} \bowtie \mathcal{F}) \rightsquigarrow \mathcal{E}_{ins} \bowtie \mathcal{F}_{new} \quad (5.19)$$

$$f_{del}(\mathcal{E} \bowtie \mathcal{F}) \rightsquigarrow \mathcal{E}_{del} \bowtie \mathcal{F}_{new} \oplus \mathcal{E}_{del} \bowtie \mathcal{F}_{del} \oplus \mathcal{E}_{del} \bowtie \mathcal{F}_{uo} \ominus \quad (5.20)$$

$$\mathcal{E}_{del} \bowtie \mathcal{F}_{ins} \ominus \mathcal{E}_{del} \bowtie \mathcal{F}_{un}$$

$$f_{un}(\mathcal{E} \bowtie \mathcal{F}) \rightsquigarrow \mathcal{E}_{new} \bowtie \mathcal{F}_{un} \oplus \mathcal{E}_{un} \bowtie \mathcal{F}_{new} \ominus \mathcal{E}_{ins} \bowtie \mathcal{F}_{un} \ominus \mathcal{E}_{un} \bowtie \mathcal{F}_{un} \quad (5.21)$$

$$f_{uo}(\mathcal{E} \bowtie \mathcal{F}) \rightsquigarrow \mathcal{E}_{new} \bowtie \mathcal{F}_{uo} \ominus \mathcal{E}_{ins} \bowtie \mathcal{F}_{uo} \ominus \mathcal{E}_{un} \bowtie \mathcal{F}_{uo} \oplus \mathcal{E}_{uo} \bowtie \mathcal{F}_{new} \ominus \quad (5.22)$$

$$\mathcal{E}_{uo} \bowtie \mathcal{F}_{ins} \ominus \mathcal{E}_{uo} \bowtie \mathcal{F}_{un} \oplus \mathcal{E}_{uo} \bowtie \mathcal{F}_{del} \oplus \mathcal{E}_{uo} \bowtie \mathcal{F}_{uo}$$

$$f_{ups}(\mathcal{E} \bowtie \mathcal{F}) \rightsquigarrow \mathcal{E}_{new} \bowtie \mathcal{F}_{ups} \oplus \mathcal{E}_{ups} \bowtie \mathcal{F}_{new} \ominus \mathcal{E}_{ups} \bowtie \mathcal{F}_{ups} \quad (5.23)$$

$$f_{new}(\mathcal{E} \bowtie \mathcal{F}) \rightsquigarrow \mathcal{E}_{new} \bowtie \mathcal{F}_{new} \quad (5.24)$$

Though the right-hand sides of the above transformation rules look rather complex they can still be efficiently evaluated by ETL tools. It can be assumed that change data is

					Join predicate unaffected		Join predicate affected	
		$E_{new} \cap E_{old}$	E_{ins}	E_{del}	E_{un}	E_{uo}	E_{un}	E_{uo}
<div style="display: flex; flex-direction: column; align-items: center;"> <div style="margin-bottom: 5px;">Join predicate unaffected</div> <div style="margin-bottom: 5px;">Join predicate affected</div> </div>	$F_{new} \cap F_{old}$		ins	del	un	uo	ins	del
	F_{ins}	ins	ins		ins		ins	
	F_{del}	del		del		del		del
	F_{un}	un	ins		un		ins	
	F_{uo}	uo		del		uo		del
	F_{un}	ins	ins		ins		ins	
	F_{uo}	del		del		del		del

Figure 2: Matrix Representation n-to-m Joins

considerable smaller in volume than base relations. Furthermore ETL tools are able to process the joins involved in parallel.

For the case of many-to-many joins we again provide a matrix representation shown in Figure 2. Note that a distinction of update operation is necessary here. Updates that affect columns used in the join predicate have to be separated from updates that do not affect those columns. The reason is that the former may cause tuples to find new join partners or loose current ones while the latter may not. Consequently, updates that affect the join predicate give rise to insertions and deletions while updates that do not affect the join predicate can often be propagated as updates. For space limitations we omit the transformation rules for many-to-many joins.

5.5 Example

We explain the derivation process of incremental load jobs by an example. Assume that there are two operational data sources A and B that store customer information. Further assume that there is a dataset C that contains information about countries and regions together with a unique identifier, say ISO country codes. The country codes for each customer are available in relations A and B . All these information shall be integrated into the customer dimension of the data warehouse. That is, customer data extracted from relations A and B is transformed to conform to an integrated schema according to the expressions a and b . This step may involve converting date formats or standardizing address information, for instance. Furthermore a unique source identifier is appended to maintain data lineage. The resulting data is then merged (union) and joined to the country relation C . For the sake of the example, assume that customer data from A is filtered by means of some predicate p after being extracted, for example, to discard inactive customers. The ETL expression \mathcal{E} to describe the initial load is given below.

$$\mathcal{E} = [\pi_a(A) \oplus \pi_b(\sigma_p(B))] \bowtie C$$

CDP expressions are derived from \mathcal{E} step-by-step, by applying suitable transformation rules. The process is exemplified for the case of insertions below.

$$\begin{aligned}
f_{\text{ins}}(\mathcal{E}) &= \\
& f_{\text{ins}}([\pi_a(A) \oplus \pi_b(\sigma_p(B))] \bowtie C) \stackrel{5.19}{=} \\
& f_{\text{ins}}([\pi_a(A) \oplus \pi_b(\sigma_p(B))] \bowtie f_{\text{new}}(C)) \stackrel{5.13}{=} \\
& [f_{\text{ins}}(\pi_a(A)) \oplus f_{\text{ins}}(\pi_b(\sigma_p(B)))] \bowtie f_{\text{new}}(C) \stackrel{5.1}{=} \\
& [\pi_a(f_{\text{ins}}(A)) \oplus \pi_b(f_{\text{ins}}(\sigma_p(B)))] \bowtie f_{\text{new}}(C) \stackrel{5.7}{=} \\
& [\pi_a(f_{\text{ins}}(A)) \oplus \pi_b(\sigma_p(f_{\text{ins}}(B)) \oplus (\sigma_p(f_{\text{un}}(B)) \bar{\bowtie} \sigma_p(f_{\text{uo}}(B))))] \bowtie f_{\text{new}}(C) = \\
& [\pi_a(A_{\text{ins}}) \oplus \pi_b(\sigma_p(B_{\text{ins}}) \oplus (\sigma_p(B_{\text{un}}) \bar{\bowtie} \sigma_p(B_{\text{uo}})))] \bowtie C_{\text{new}}
\end{aligned}$$

CDP expressions can be translated into an executable ETL job as shown in [DHW⁺08]. The transformation rules are designed to “push” the function f_{ins} , f_{del} , f_{un} , or f_{uo} into the ETL expression. Eventually these functions are directly applied to source relations and thus denote the output of CDC techniques. At this point no further transformation rule is applicable and the derivation process terminates. The resulting CDP expression shows that the propagation of insertions is possible if A ’s CDC technique is capable of capturing insertions and B ’s CDC technique is capable of capturing insertions and updates in their initial state and in their current state. In case the available CDC techniques do not meet these requirements the propagation of insertions is impractical. However, it is possible to propagate upserts in this situation as the evaluation of $f_{\text{ups}}(\mathcal{E})$ would show.

5.6 Experimental Results

We provide some experimental results to demonstrate the advantage of incremental loading over full reloading. We stress that the measurement is exemplary. Nevertheless, the results clearly suggest performance benefits. Our experiment is based on the sample scenario described in the previous section. We chose the cardinality of the customer relation and the country relation to be 600,000 tuples and 300 tuples, respectively. During the experiment, we stepwise increased the number of changes to the source relations. At each step, equal numbers of tuples were inserted, deleted, and updated. We employed ETL jobs for initial loading, full reloading, and incremental loading. The jobs for initial loading and full reloading differ in the sense that the former expects the target relation to be empty while the latter does not. Instead, it performs lookups to decide whether tuples need to be inserted, updated, or remain unchanged. The incremental load job computes three separate datasets, i.e. tuples to be inserted and tuples to be updated in both, their initial state and their current state. Deletions are not propagated since historical data is kept in the data warehouse.

We measured the time to compute change data for data warehouse refreshment, i.e. we focused on CDP and excluded CDC and CDA from our considerations. The cost of CDC

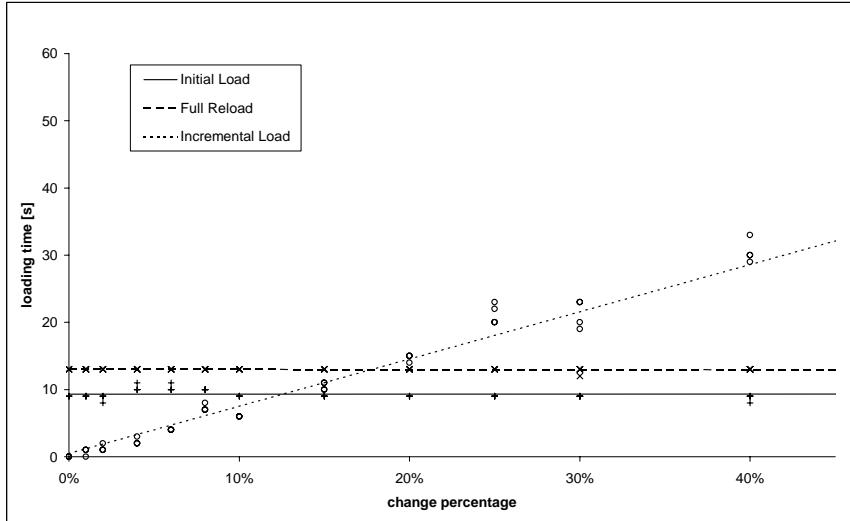


Figure 3: Change Data Propagation Time Comparisons

depends on the CDC technique used and the architecture of the CDC system. However, none of the CDC techniques described in Section 3.2 has a stronger impact on the data source than a full extraction required for full reloading. The cost of CDA again depends on the loading facility used. However, the CDA cost is the same for both, full reloading and incremental loading.

The results of the experiment are provided in Figure 3. Expectedly, the time for the initial load and the full reload are constant, i.e. the number of source data changes does not impact the runtime of these jobs. The full reload is considerably slower than the initial load. The reason is the overhead incurred by the frequent lookups. Incremental loading clearly outperforms full reloading unless the source relations happen to change dramatically.

6 Conclusion

In this paper we addressed the issue of data warehouse refreshment. We argued that incremental loading is more efficient than full reloading unless the operational data sources happen to change dramatically. Thus, incremental loading is generally preferable. However, the development of ETL jobs for incremental loading is ill-supported by existing ETL tools. In fact, separate ETL jobs for initial loading and incremental loading have to be created by ETL programmers so far. Since incremental load jobs are considerably more complex their development is more costly and error-prone.

To overcome this obstacle we proposed an approach to derive incremental load jobs from given initial load jobs based on equational reasoning. We therefore reviewed existing Change Data Capture (CDC) techniques that provide the input for incremental loading.

We further reviewed existing loading facilities to update the data warehouse incrementally, i.e. to perform the final step of incremental loading referred to as Change Data Application (CDA). Based on our analysis we introduced a formal model for *change data* to characterize both, the output of CDC techniques and the input of CDA techniques. Depending on the technical realization, CDC techniques suffer from limitations and are unable to detect certain changes. For this reason we introduced a notion of *partial* change data.

Our main contribution is a set of equivalence preserving transformation rules that allow for deriving incremental load jobs from initial load jobs. We emphasize that our approach works nicely in the presence of *partial* change data. The derived expressions immediately reveal the impact of partial input change data on the overall change propagation. That is, the interdependencies between tolerable CDC limitations and satisfiable CDA requirements become apparent. We are not aware of any other change propagation approach that considers limited knowledge of change data. Thus, we are confident that our work contributes to the improvement of ETL development tools.

Future work will focus on advanced transformation operators such as aggregation, outer joins, and data restructuring such as pivoting. We further plan to investigate the usage of the staging area, i.e. allow ETL jobs to persist data that serves as additional input for the subsequent runs. By utilizing the staging area, CDC limitations can be compensated to some extent, i.e. partial change data can be complemented while being propagated. Moreover, we expect performance improvements from persisting intermediary results.

References

- [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.
- [AN08] Alexander Albrecht and Felix Naumann. Managing ETL Processes. In *NTII*, pages 12–15, 2008.
- [BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently Updating Materialized Views. In *SIGMOD Conference*, pages 61–71, 1986.
- [BT98] Michele Bokun and Carmen Taglienti. Incremental Data Warehouse Updates. *DM Review Magazine*, May 1998.
- [CW91] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB*, pages 577–589, 1991.
- [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.
- [DS] IBM WebSphere DataStage. <http://www-306.ibm.com/software/data/integration/datastage/>.
- [DWE] IBM DB2 Data Warehouse Enterprise Edition. www.ibm.com/software/data/db2/dwe/.

- [GL95] Timothy Griffin and Leonid Libkin. Incremental Maintenance of Views with Duplicates. In *SIGMOD Conference*, pages 328–339, 1995.
- [GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD Conference*, pages 157–166, 1993.
- [IPC] Informatica PowerCenter. http://www.informatica.com/products_services/powercenter/.
- [JD08] Thomas Jörg and Stefan DeBloch. Towards Generating ETL Processes for Incremental Loading. In *IDEAS*, pages 101–110, 2008.
- [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.
- [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.
- [LN07] Ulf Leser and Felix Naumann. *Informationsintegration*. dpunkt.verlag, 2007.
- [OWB] Oracle Warehouse Builder. <http://www.oracle.com/technology/products/warehouse/index.html>.
- [QGMW96] Dallen Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making Views Self-Maintainable for Data Warehousing. In *PDIS*, pages 158–169, 1996.
- [QW91] Xiaolei Qian and Gio Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.*, 3(3):337–341, 1991.
- [Sim05] Alkis Simitsis. Mapping conceptual to logical models for ETL processes. In *DOLAP*, pages 67–76, 2005.
- [SVS05] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. Optimizing ETL Processes in Data Warehouses. In *ICDE*, pages 564–575, 2005.
- [SVTS05] Alkis Simitsis, Panos Vassiliadis, Manolis Terrovitis, and Spiros Skiadopoulos. Graph-Based Modeling of ETL Activities with Multi-level Transformations and Updates. In *DaWaK*, pages 43–52, 2005.
- [VSS02] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual modeling for ETL processes. In *DOLAP*, pages 14–21, 2002.
- [Yu06] Tsae-Feng Yu. A Materialized View-based Approach to Integrating ETL Process and Data Warehouse Applications. In *IKE*, pages 257–263, 2006.
- [ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *SIGMOD Conference*, pages 316–327, 1995.