

Towards Generating ETL Processes for Incremental Loading

Thomas Jörg
University of Kaiserslautern
67653 Kaiserslautern, Germany
joerg@informatik.uni-kl.de

Stefan DeBloch
University of Kaiserslautern
67653 Kaiserslautern, Germany
dessloch@informatik.uni-kl.de

ABSTRACT

Extract, Transform, and Load (ETL) processes physically integrate data from multiple, heterogeneous sources in a central repository referred to as data warehouse. Physically integrated data gets stale when source data is changed, hence periodic refreshes are required. For efficiency reasons data warehouses are typically refreshed incrementally, i.e. changes are captured at the sources and propagated to the data warehouse on a regular basis. Dedicated ETL processes referred to as incremental load processes are employed to extract changes from the sources, propagate the changes, and refresh the data warehouse incrementally. Changes required in the data warehouse are inferred from changes captured at the sources during change propagation. The creation of incremental load processes is a complex task reserved to trained ETL programmers. In this paper we review existing Change Data Capture (CDC) techniques and discuss limitations of different approaches. We further review existing techniques for refreshing data warehouses. We then present an approach for generating incremental load processes from abstract schema mappings.

Categories and Subject Descriptors

H.1 [Models and Principles]: Miscellaneous
; H.2.1 [Database Management]: Logical Design

General Terms

Design

Keywords

Data Integration, Data Warehousing, Extract, Transform, and Load (ETL), Information Integration

1. INTRODUCTION

Extract, Transform, and Load (ETL) processes play a vital role in data warehousing [16, 15, 19]. ETL processes

involve extracting data from multiple, heterogeneous data sources, transforming and cleansing data, and ultimately loading data into the data warehouse. The creation of ETL processes is a labor-intensive task. Various systems with data transformation capabilities, such as dedicated ETL tools, (federated) database systems, database replication systems, message-oriented-middleware, or operating system utilities may participate in a joint ETL process. ETL programmers need to be familiar with a multitude of different programming and processing paradigms. Clearly, a higher level of abstraction is desirable. To meet this need, ETL development typically starts with the definition of a logical data map [15]. This document declares how source data relates to data in the warehouse by specifying *schema mappings*. Schema mappings are widely used in tools that support information integration tasks in general. Users are presented with source and target schemas and specify correspondences between them, often by drawing arrows between interrelated schema elements. These correspondences may be annotated to indicate the intended transformation semantics. A sample schema mapping is depicted in figure 1. On the left-hand side, two source schemata are found. They hold information about products and product packaging. The target schema on the right-hand side belongs to a product dimension of a data warehouse. Correlated schema elements are connected by arrows. Two annotations have been added to the mapping. Annotation (1) is used to specify a value conversion. In the example, cryptic codes are translated to rather self-explanatory terms. Annotation (2) specifies a predicate with regard to the product dataset. It prevents product data from being loaded to the data warehouse before it has been approved.

The logical data map is used as a functional specification by ETL developers. ETL programming, however, is done manually, since mapping tools and ETL development tools do not directly interoperate. This “gap” has only recently been bridged by the Orchid system [11], which facilitates the conversion from schema mappings to ETL processes and vice versa. Orchid provides an abstract operator model, referred to as Operator Hub Model (OHM), which captures the transformation semantics common to both, schema mappings and ETL processes. For this purpose, a set of basic transformation operators is defined. Roughly speaking, OHM operators are generalizations of relational algebra operators: FILTER, PROJECT, UNION, and JOIN correspond to the relational selection, projection, union, and join operator, respectively. The OHM GROUP operator borrows from the SQL GROUP BY clause.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IDEAS08 2008, September 10-12, Coimbra [Portugal]

Editor: Bipin C. DESAI

Copyright 2008 ACM 978-1-60558-188-0/08/09 ...\$5.00.

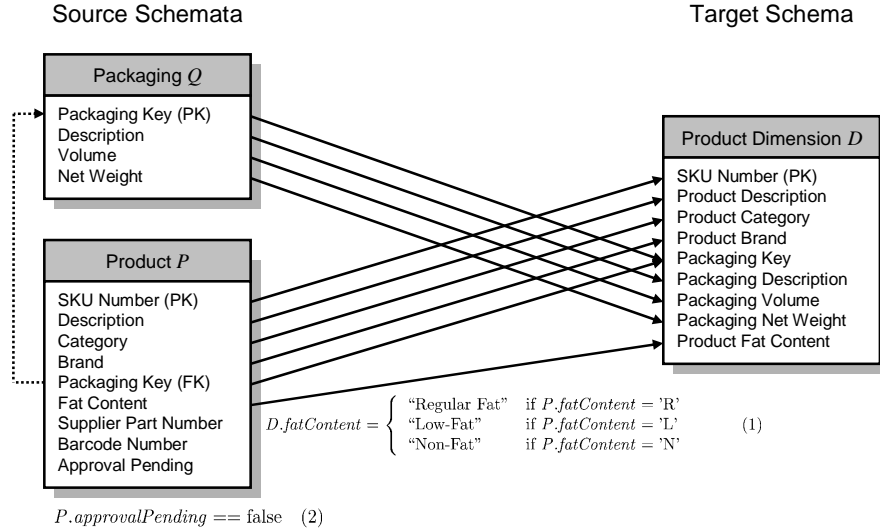


Figure 1: Sample Schema Mapping

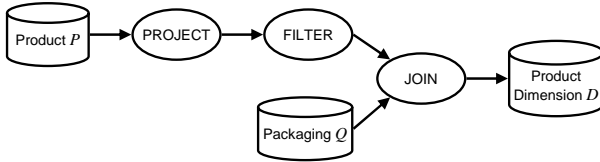


Figure 2: Sample OHM Instance

Complex data transformations are modeled by assembling basic OHM operators to a directed acyclic graph referred to as *OHM instance*. Each OHM operator acts as a node within this graph. Edges route datasets through the graph, i.e. the output of an operator is input to subsequent operators. Consider the sample schema mapping depicted in figure 1. Orchid can automatically translate this mapping into the OHM instance depicted in figure 2 and vice versa. Both, the sample mapping and the sample OHM instance model equal data transformations: The PROJECT operator inputs the records from the product dataset. It drops the supplier part number and barcode number columns because they are not mapped to the target schema and converts the value of the fat content column. The results are passed to the FILTER operator. Here all records that do not satisfy the filter predicate are dropped. Finally, the remaining product records are joined to packaging records by the JOIN operator and the result is delivered to the product dimension.

Orchid can automatically deploy OHM instances, i.e. generate platform specific code such as ETL scripts or SQL statements from a given OHM instance. Obviously, Orchid greatly eases ETL development by translating abstract schema mappings to executable ETL processes. However, there is an important aspect of ETL development that has not been addressed: Physically integrated data gets stale when the source data is changed. Orchid-generated ETL processes extract source data exhaustively and (re)build the data warehouse within each loading cycle. This approach obviously does not scale well as data volumes increase and loading intervals are shortened. It is desirable to capture

changes of source data and propagate these changes to the data warehouse. This approach is known as *incremental loading*. The volume of changed data is usually small compared to the entire datasets, thus, incremental loads can be assumed to work more efficiently than full reloads. Dedicated ETL processes are required to perform incremental loads that are handcrafted by ETL programmers as yet. The creation of such processes is a complex task since results of joins, aggregations, or complex data cleansing operations need to be updated incrementally during change propagation. This motivated us to explore the automatic generation of incremental load processes from abstract schema mappings. This problem 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 5 on related work.

In this paper we present an approach to rewrite OHM instances to incorporate incremental update semantics. We explore how the resulting OHM instances can be simplified by taking properties of data sources into account. If, for instance, referential integrity is enforced by the source system, several combinations of source changes must not happen and can be excluded from considerations. This allows reducing the complexity of change propagation processes. In section 2 we present Change Data Capture techniques used to detect and capture changes of interest in operational data sources. These techniques deliver the input for incremental load processes. In section 3 we discuss loading facilities used to incrementally update persistent datasets, i.e. to perform the very last step of an incremental load. The main contribution of this work is presented in section 4. Here we describe our approach to rewrite Orchid-generated OHM instances to incorporate incremental update semantics. We discuss related work in section 5 and conclude in section 6.

2. CHANGE DATA CAPTURE

Change Data Capture (CDC) is a generic term for techniques that monitor operational data sources with the ob-

jective of detecting and capturing data changes of interest [15, 9]. 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. Different CDC techniques are found in practice. We provide an overview in subsection 2.1. In subsection 2.2 we present a model for *change data* delivered by CDC techniques. It allows for a uniform treatment of CDC techniques though the technical realizations differ.

2.1 Change Data Capture Techniques

CDC techniques applied in practice roughly follow three main approaches, namely log-based CDC, utilization of audit columns, and calculation of snapshot differentials [15, 9, 18, 17]. Below, each approach is described briefly.

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. While this information is intended for recovery it can also 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 record and indicate the time at which the record 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. As soon as a record is removed from the operational data source the associated audit column is lost. Deletions, however, are often not reflected in data warehouses and may safely be ignored. Furthermore deletions never occur on certain data structures like append-only tables.

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 an earlier one. Each time a snapshot is extracted it is preserved for future use. During the subsequent extraction step, it is compared against the recent snapshot and changes are inferred. A major drawback of the snapshot differential approach is the need for frequent extractions of large quantities of data, yet 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. This information is overwritten in the data source during the update operations and therefore no longer available. Interestingly, snapshot differential implementations in practice usually have the same limitation. They do not provide the initial state of updated records while this would be feasible in principle. Note that the required data is available in the snapshot taken during the previous run. Log-based CDC ap-

proaches in practice typically capture all types of changes, i.e. insertions, deletions, and the initial and current state of updated records. The same applies to CDC techniques that instantly record changes by means of triggers or application logic. Table 1 summarizes the limitations of the CDC techniques mentioned.

2.2 Change Data

In the preceding subsection we have shown the diversity of CDC techniques with respect to their technical realization and, more importantly, their ability to detect different types of changes. We now present a model for the output of CDC techniques referred to as *change data*. This allows us to treat any CDC implementations as a black box with a defined interface.

First, we define our data model, which is a generalization of the relational model introduced by Codd [10]. We refer to instances of our data model as *datasets*¹. The relational model is set-based, i.e. it does not allow for duplicate tuples within one relation. We drop this restriction since it hinders the description of ETL dataflows. In particular no implicit duplicate elimination is performed by ETL tools, databases, or other data integration systems in practice. Following Codd’s definition of relations we define datasets as below.

DEFINITION 1. *Given sets S_1, \dots, S_n (not necessarily distinct) of atomic values, a dataset D on these n sets is a multiset of n -tuples each of which has its first element from S_1 , its second element from S_2 , and so on. We refer to the n -tuple (S_1, \dots, S_n) as the schema of D . More formally, D is a pair (D', count) such that*

$$\begin{aligned} D' &\subseteq S_1 \times \dots \times S_n \\ &\text{and} \\ \text{count} : D' &\rightarrow \mathbb{N}^+ \end{aligned}$$

where D' is the underlying set of tuples contained in D and the function count maps each tuple in D' to its multiplicity, i.e. the number of its occurrences in D .

Below, we make use of two multiset operators, namely the additive union denoted by \uplus and the bag difference denoted by \ominus . The additive union is a generalization of the set union that adds up the multiplicities of elements in multisets. That is, if an element x appears n times in multiset S and m times in multiset T , the number of occurrences of x in $S \uplus T$ is $(n + m)$. The bag difference subtracts multiplicities. Hence, the number of occurrences of x in $S \ominus T$ is $(n - m)$ if $n \geq m$ and 0 otherwise.

Data sources can be perceived as their content at a particular point in time, which is essentially a dataset. The Orchid system follows this notion, i.e. the generated ETL jobs extract all data contained in the sources for further processing. The perception of data sources changes in the light of CDC. In fact, CDC techniques determine how the content of data sources has changed over time. We refer to the output of CDC techniques as *change data*. A definition is provided below. Change data is typically captured at tuple granularity. Hence each change data tuple relates to an inserted, deleted, or updated source data tuple.

¹Though our data model is based on multisets we favor the established term “dataset” over “data multiset” or “data bag”.

	Insert	Delete	Update (new state)	Update (old state)
Audit Columns	yes	no	yes ¹⁾	no
Snapshot Differential	yes	yes	yes	no ²⁾
Log scraping or sniffing	yes	yes	yes	yes

¹⁾ Possibly indistinguishable from insert

²⁾ Though feasible in principle

Table 1: Limitations of Change Data Capture Techniques

DEFINITION 2. Given a dataset D^{new} of schema S that represents the current content of a data source and a dataset D^{old} of schema S that represents the content of the same data source at a previous point in time, change data is a set of datasets $\Delta D, \nabla D, \boxplus D, \boxminus D$ with $\Delta D \subseteq D^{new}, \nabla D \subseteq D^{old}, \boxplus D \subseteq D^{new}$, and $\boxminus D \subseteq D^{old}$ where

- ΔD denotes tuples that have been inserted,
- ∇D denotes tuples that have been deleted,
- $\boxplus D$ denotes the current state of tuples that have been updated (referred to as update new),
- $\boxminus D$ denotes the initial state of tuples that have been updated (referred to as update old)

such that the following conditions hold.

$$\begin{aligned}
D^{new} &= D^{old} \boxplus \Delta D \boxplus \boxplus D \ominus \nabla D \ominus \boxminus D \\
D^{old} &= D^{new} \ominus \Delta D \ominus \boxplus D \boxplus \nabla D \boxplus \boxminus D \\
|\boxplus D| &= |\boxminus D|
\end{aligned}$$

In particular, the above definition assures that each record in D^{new} occurs at most once in either ΔD or $\boxplus D$. Similarly, each record in D^{old} occurs at most once in either ∇D or $\boxminus D$. In other words, change data defines at most one modification per record. Note that records may be modified several times between one loading cycle. In such a case, all modifications are merged into one change data tuple. This property allows for change data application with no particular order, thereby avoiding the risk of “lost updates”.

As discussed in subsection 2.1 CDC techniques may suffer from various limitations. Non-capturable changes can obviously not contribute to change data. Therefore only a subset of $\{\Delta D, \nabla D, \boxplus D, \boxminus D\}$ may be delivered by a CDC technique. We refer to change data of this kind as *partial* change data.

In related work on incremental maintenance of materialized views, updates are typically modeled as delete-insert pairs [12, 20, 22]. While updates have been considered independently in theoretical considerations on properties of materialized views [13, 8, 28] we are not aware of any change propagation approach that processes updates separately. We argue that our model naturally matches the output of existing CDC techniques [18, 15, 9, 17] and easily extends to handle CDC limitations, CDA requirements, and partial change data.

3. CHANGE DATA APPLICATION

Physically integrated data becomes stale when the underlying sources are updated; periodic refreshes are performed

to catch up with source changes. For efficiency reasons, the refreshment is typically performed *incrementally*, meaning that changes are applied where necessary while the rest of the dataset is unaffected. The input required for incremental refreshes is effectively *change data* as defined in subsection 2.2. Therefore, we refer to the process of refreshing physically integrated data as Change Data Application (CDA).

Numerous techniques for CDA are used in practice. They differ not only in terms of implementation and target platform but also in the change data required. Consider the SQL MERGE statement². The MERGE statement refreshes a target table using change data. Records in the target that match change data records are updated while records that do not exist in the target are inserted. Database systems usually come along with bulk load utilities which are also able to detect whether insert or update operations are appropriate on a per-record level. ETL tools furthermore offer comparable loading facilities. What these CDA techniques have in common is the demand for one dataset that contains records to be inserted or updated in a joint manner, i.e. a form of partial change data. These CDA techniques perform lookups to check for the existence of target records. In [15] the authors state that this approach may result in poor performance. They argue that the ETL process should explicitly separate data that is to be updated from data that is to be inserted, thereby eliminating the need for frequent lookups when loading the target dataset. This again corresponds to partial change data where insertions and updates are available as separate datasets. Note that deletions have to be segregated from insertions and updates in either of the two cases to be applied to the target dataset. Complete change data is required for more sophisticated refresh strategies commonly used in data warehousing. Here, different actions may be taken depending on the particular columns that have been updated. Changing a product name, for example, could be considered as an error correction; hence the corresponding name in the data warehouse can simply be overwritten. 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. Note that a distinction of cases is only possible if both, the initial state and the current state of updated records are available unless lookups to the target dataset are acceptable.

In summary, CDA techniques differ in their *requirements* with regard to consumable change data. While some techniques are able to process insertions and updates provided in an indistinguishable manner, others demand for complete change data. Note that CDA requirements can be charac-

²The MERGE statement has been introduced with the SQL:2003 standard.

terized by the change data model introduced in section 2.2. In the next section we discuss the propagation of change data originating from CDC techniques and ultimately fed into CDA techniques.

4. CHANGE DATA PROPAGATION

In section 1 we stated that mappings can be used to describe data transformations in an abstract manner. The Orchid system allows translating such mappings into OHM instances with equal transformation semantics. OHM instances can be deployed to an executable ETL process hereafter. Orchid-generated OHM instances describe data flows such that the entire content of the sources is extracted, transformed, and loaded to the data sinks. We refer to Orchid-generated OHM instances as *static*. Static OHM instances are inadequate for the generation of incremental load processes. Incremental load processes utilize CDC techniques to capture changes in data sources and CDA techniques to incrementally update the data sinks. The heart and soul of incremental loads is change data propagation (CDP). A CDP process consumes change data acquired from the data sources and outputs change data ready for application to the data sinks. Note, that there are numerous ways to implement CDP processes. Besides dedicated ETL tools, the data transformation capabilities of database systems, operating systems, mainframe systems, or middleware can be leveraged, possibly in a combined manner [15]. The implementation of CDP processes is not in the scope of this paper. We instead make use of the OHM to describe CDP processes in a platform-independent manner. Refer to [11] for a description of the deployment of OHM instances into multiple target platforms.

CDP acts as the link between CDC and CDA. Thus, CDP processes need to cope with CDC limitations and satisfy CDA requirements at the same time. This, however, is not always possible. Two questions arise.

- Given CDA requirements for each data sink, what CDC limitations are acceptable for each data source?
- Given CDC limitations for each data source, what CDA requirements are satisfiable for each data sink?

Even with “powerful” CDC techniques in place it is desirable to be aware of acceptable CDC limitations. CDC techniques can typically be configured to ignore certain types of changes, thereby improving performance. Data warehouses, for instance, usually retain data even if it has been deleted in operational data sources, hence deletions can safely be ignored by CDC techniques. Similarly, it is sometimes advantageous to apply CDA techniques with lower requirements than potentially satisfiable. As a general rule, CDP processes become more complex the higher the CDA requirements are. It might be desirable to minimize the complexity of CDP processes if manual rework or maintenance is intended.

We now describe our approach to the generation of CDP processes. We therefore derive so called *incremental* OHM instances from static OHM instances such that equivalent data transformations are described. Note that both, the static and incremental OHM instance correspond to the same mapping. As a first step we provide *incremental* variants of conventional OHM operators. Incremental operator variants consume change data instead of datasets. Similarly,

they output change data and are hence interconnectable. Static OHM instances are translated to incremental OHM instances by replacing each contained operator with its incremental variant. An incremental variant of an OHM operator is specified as a composition of OHM operators, i.e. an incremental operator variant is an OHM instance. This approach allows us to leverage the Orchid system in two ways. First Orchid allows for the translation of mappings to static OHM instances which provide the basis for the creation of incremental OHM instances. Secondly, Orchid provides means for the deployment of OHM instances. Since incremental OHM instances are made up from conventional OHM operators the deployment can be performed by existing infrastructure.

We now exemplify the creation of CDP processes with our initial example given in section 1. The static OHM instance generated by Orchid is depicted in figure 2. It provides the starting point for our considerations. Assume that the product dataset contains audit columns that are utilized for CDC. So we face several CDC limitations: Inserts and updates are indistinguishable, deletions remain undetected, and only the current states of updated records are available. Further assume that non-partial change data is available for the packaging dataset. Two questions arise. What CDA requirements are satisfiable, i.e. what techniques can be applied for incremental loads in practice? What transformations are required to propagate change data from source to target and how can the propagation process be described by an OHM instance?

In the following subsections we present incremental variants of OHM operators contained in the sample scenario, namely PROJECT, FILTER, and JOIN. We discuss the impact of partial change data and certain properties of source systems and show how incremental operator variants need to be adapted. We then discuss how CDA requirements follow from given CDC limitations in the sample scenario. Finally, we briefly address OHM operators not considered until then.

4.1 Project Operator

Since given CDC limitations are our starting point, the replacement of operators within the static OHM instance is performed in a source-to-target direction. In the sample OHM instance depicted in figure 2 the PROJECT operator is to be replaced first. Incremental operator variants in general propagate (and thereby transform) change data in a way such that the resulting change data indicates target data changes that occur in response to source data changes. The PROJECT operator allows dropping, adding, and renaming columns, to perform type casts, value assignments, and value conversions. In particular, no duplicate elimination is performed. The construction of an incremental PROJECT variant is straightforward. The transformations enumerated before directly carry over from dataset processing in static OHM instances to change data processing in incremental OHM instances. That is, PROJECT operators in static OHM instances are replaced by multiple similar PROJECT operators during the generation of incremental OHM instances. Each component of change data, i.e. insert, delete, update new, and update old is processed separately, thus an incremental PROJECT variant comprises several PROJECT operators, as depicted in figure 4.

The PROJECT operator in the sample scenario is used to drop several columns and to convert abbreviated values to

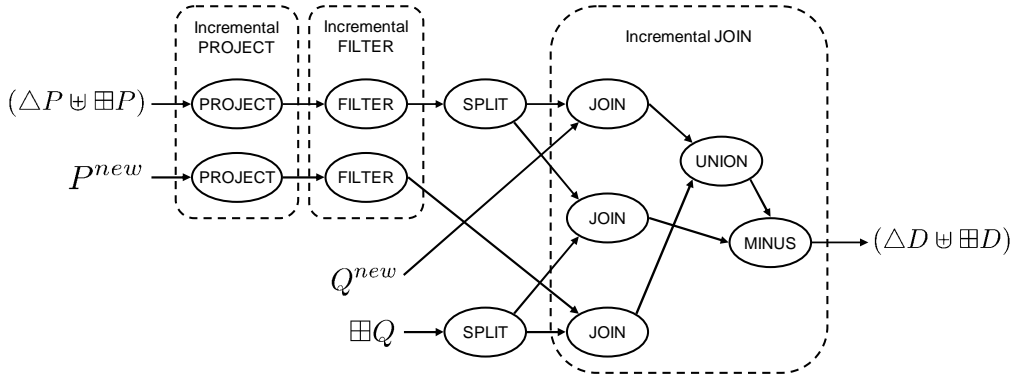


Figure 3: Incremental OHM Instance

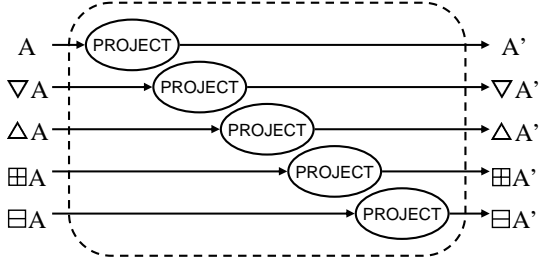


Figure 4: Incremental PROJECT Variant

rather self-explanatory terms (see mapping 1). Since audit columns are used for CDC change data is partial. Only insertions and the current state of updated records are available and cannot be distinguished from each other. The incremental PROJECT variant is adapted according to its input by omitting all operators dedicated to unavailable change data. In the sample scenario the incremental PROJECT variant hence comprises two basic PROJECT operators (see figure 3).

Further aspects of incremental PROJECT variants are worth mentioning here. Consider that non-partial change data is consumed, i.e. update new and update old are available. Dropping columns may then lead to superfluous update pairs because all modified columns might have been discarded. It is, however, possible to detect and discard superfluous update pairs at the cost of additional processing (not shown in figure 4).

Furthermore there are CDC techniques [6, 4] that can be configured to ignore certain columns, sometimes referred to as column subsetting. Updates that affect only such columns are not captured neither are these columns included in the output change data. Thus, PROJECT operators that drop columns may be “pushed down” to CDC techniques. For this purpose, CDC techniques are registered as deployment platforms with the Orchid system. During the deployment step an appropriate CDC subscription is generated.

4.2 Filter Operator

We proceed with the next operator in the sample OHM instance, namely the FILTER operator. An OHM FILTER selects records from a dataset that satisfy a given boolean predicate and discards all others. An incremental FILTER variant processes change data as follows.

- 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 record 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 records, 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 source record 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 change data is only partially available, the incremental FILTER needs to be adapted. If the initial value of updated records (update old) is unavailable it cannot be checked against the filter predicate. Thus, it is not possible to conclude whether the updated source record has been propagated to the data sinks before. Consequently, it cannot be decided if an insert or an update is to be issued, provided that the current value of the updated source record satisfies the filter predicate. Thus, an incremental FILTER variant unaware of the initial value of updated records may only output change data with indistinguishable inserts and updates.

Consider the sample scenario again. The change data fed into the incremental FILTER is partial in such a way that inserts and updates are indistinguishable. Each record is checked against the filter predicate and propagated if it is satisfied. The incremental FILTER variant is depicted in figure 3.

4.3 Join Operator

The OHM JOIN operator models n-way inner equi-joins, left outer equi-joins, full outer equi-joins, and cross joins. We restrict ourselves to two-way inner equi-joins for now and discuss the design of incremental JOIN variants for this restricted case. The objective is to infer change data that is to be propagated towards the data sinks from change data













































		join predicate unaffected				join predicate affected		
								
join predicate unaffected								
								
								
								
join predicate affected								
								
								

Figure 5: Matrix Representation of an Incremental Join

originating from two sources. Complex interdependencies between the source change data have to be taken into account. Consider the sample scenario. Here a join is used to relate product information with details concerning the packaging. Consider that a packaging record has been updated. The impact on the data sink now depends on changes within the product dataset that occurred in the meantime. For each unmodified product record that references the updated packaging record an update is to be issued. The same applies to product records that have been updated. If no referencing product record exists the update operation is without effect. Each newly inserted product record that references the updated packaging record gives rise to an insert. Similarly, for each deleted product record that used to reference the updated packaging record an deletion is to be issued. Note that the sample scenario does not present the most general form of a join since a primary key acts as one join key. Primary keys are assumed to be unique identifiers, hence functional dependencies exist between columns of target records. We further elaborate on this thought in subsection 4.4. First, we introduce the most general form of an incremental inner equi-join. We then discuss properties of data sources that allow for simplifications.

We use a matrix representation to depict the interdependencies between the two input change data sets and the output change data as shown in figure 5. The basic idea is to identify the impact of each combination of different types of input change data (i.e. insert, delete, update new, and update old) on the output change data. The matrix is interpreted as follows. The column headers and row headers denote input datasets for data source A and data source B , respectively. From left to right, the column headers denote the unmodified subset of source data ($A^{unchanged} = A^{new} \ominus \triangle A \ominus \boxplus A$), insertions $\triangle A$, deletions ∇A , updated records in their current state $\boxplus A$, and updated records in their initial state $\boxminus A$. The last two column headers again denote updates. The reason is that updates leaving the join key columns unaltered have to be distinguished from updates changing these columns. The row headers are arranged in a similar way.

We refer to the intersections of columns and rows as cells. Cells may be labeled. Consider the insert column and the insert row, for instance. The cell at the intersection point is labeled as an insert. That is, $\triangle A \bowtie \triangle B$ is part of the insert dataset $\triangle R$ of the resulting change data. All other cells

labeled as insertions also contribute to $\triangle R$. Each of them indicates a join between two datasets. $\triangle R$ is the additive union of all join results. ∇R , $\boxplus R$, and $\boxminus R$ are computed in a similar manner. For space restrictions, we omit the proof of correctness.

If change data is only partially available, cells within the corresponding column or row cannot be evaluated. Thus, they cannot contribute to the resulting change data and the overall calculation gets impractical. Assume that $\boxminus B$ is unavailable, for instance. This makes the calculation of ∇R and $\boxminus R$ impractical as evident from the matrix depicted in figure 5.

4.4 Join Matrix Transformations

The matrix depicted in figure 5 represents the most general incremental join. No assumptions with regard to the data sources are made. There are, however, certain properties of source data that allow for simplifications of the matrix.

In *append-only datasets*, for instance, deletions and updates never occur. Hence all labels within related columns or rows can be erased. Since delete and update datasets are surely empty each cell evaluates to a join with one empty input dataset. These joins will not produce any results and can hence be excluded from the computation.

Another interesting source property is the presence of *primary keys*. Primary keys are guaranteed to be unique. Furthermore they are typically not modified³. If a primary key acts as a join key no updates that affect the join predicate may ever happen. This situation is found in the sample scenario. The lower two rows of the join matrix can hence be cleared as indicated by gray labels in figure 6 (1).

The fact that a unique column acts as a join key establishes *functional dependencies* within records of the join result. Clearly the packaging details are functionally dependent on the SKU number (the primary key of products) in the product dimension in the sample scenario. Thus, changing the foreign key column that references the packaging of a product results in an *update* of the product dimension; the reason is that at most one new join partner is found since the packaging key is unique. Note that this is not true in the general case where an arbitrary amount of newly matching records may be found, which results in varying quantities of inserts and deletions to be propagated. In terms of the join matrix the functional dependencies cause the replacement of the insert and delete labels in the two rightmost columns by update new and update old labels, respectively. The adaptation of the join matrix is shown in figure 6 (2).

If data resides in database systems, *referential integrity* is typically enforced automatically. We assume that this is the case in the sample scenario. Under these circumstances the join matrix can again be simplified. There must not exist any product that references a non-existing packaging record. Thus, the deletion of a packaging record may never leave a product with a “dangling” foreign key reference. In fact, the database system assures referential integrity by either prohibiting the operation or forcing the deletion of dependent records. Consequently, a product record will not find a new join partner or loose its present join partner because of insertions or deletions within the packaging dataset, respectively.

³Though primary keys may be updatable from a technical perspective, changes are usually prohibited by business rules.

The related cells in the matrix can therefore be cleared as shown in figure 6 (3). On the right-hand side of the figure 6 the simplified join matrix is depicted. The distinction between updates with and without impact on the join predicate is no longer required. Taking referential integrity into account we could rule out several cases thereby eliminating the “conflicting” cell labels. These labels have either been replaced (2) or removed (3) from the join matrix. Hence the differentiation is needless.

The right-hand side join matrix has been further modified to adapt it to *partial change data*: The insert and update columns have been merged (4) since product change data is partial and the two datasets are provided in a joint manner. Merging columns or rows in the join matrix is feasible since inner joins are distributive over additive union⁴. Merging columns or rows makes sense only if the resulting dataset can be processed by subsequent incremental operators or CDA techniques. Such techniques exist for joint inserts and updates as we have shown in section 3. There is, however, no other meaningful combination of different types of change data.

We now exemplify the design of an incremental JOIN variant on basis of the join matrix. Recall that product change data is partial. Neither deletions nor the initial state of updated records are available as indicated by the gray columns. The join matrix is interpreted as follows: Cell labels appearing within unavailable columns hinder the calculation of the corresponding change data as a whole. Consequently, neither ∇D nor $\boxminus D$ can be calculated (recall that D denotes the target product dimension). All cells with insert or update labels are available. Some of the labels have been merged into the same cells as discussed above. This indicates that the union of these datasets can be calculated while the individual datasets cannot. Note that insertions are not distinguishable from updates within the resulting dataset. To compute $(\Delta D \uplus \boxplus D)$, all cells with either insert or update labels (or both) have to be considered. This results in the equation below. Note that each of the labeled cells in the matrix corresponds to one join within the equation.

$$\begin{aligned}
 (\Delta D \uplus \boxplus D) = & \left[Q^{unchanged} \bowtie (\Delta P \uplus \boxplus P) \right] \uplus \\
 & \left[\Delta Q \bowtie (\Delta P \uplus \boxplus P) \right] \uplus \\
 & \left[\boxplus Q \bowtie P^{unchanged} \right] \uplus \\
 & \left[\boxplus Q \bowtie (\Delta P \uplus \boxplus P) \right]
 \end{aligned}$$

The equation above can not directly be evaluated because the available input values are P^{new} , Q^{new} , $(\Delta P \uplus \boxplus P)$, ΔQ , ∇Q , $\boxplus Q$, and $\boxminus Q$. It can, however, be rewritten to the equation below (with $P^{unchanged} = P^{new} \ominus \Delta P \ominus \boxplus P$ and $Q^{unchanged} = Q^{new} \ominus \Delta Q \ominus \boxplus Q$).

$$\begin{aligned}
 (\Delta D \uplus \boxplus D) = & \left[(\Delta P \uplus \boxplus P) \bowtie Q^{new} \right] \uplus \\
 & \left[P^{new} \bowtie \boxplus Q \right] \ominus \\
 & \left[(\Delta P \uplus \boxplus P) \bowtie \boxplus Q \right]
 \end{aligned}$$

The translation of the above equation into an incremental JOIN variant is straightforward. The resulting OHM instance is depicted in figure 7. Here the three joins, the addi-

⁴The equation $A \bowtie (B \uplus C) = (A \bowtie B) \uplus (A \bowtie C)$ holds.

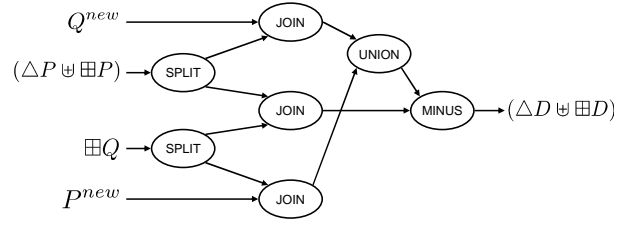


Figure 7: Incremental JOIN variant

tive union and the bag difference are expressed in terms of OHM operators. The complete incremental OHM instance results from assembling the previously discussed incremental operator variants as shown in figure 3.

Our analysis entails the answer to our initial question concerning acceptable CDA requirements: Only such CDA techniques capable of processing inserts and updates in a joint manner can be employed in the sample scenario (see section 3). Deletions cannot be propagated to the target dataset. It is particularly interesting that the required change data of the packaging dataset covers only $\boxplus Q$ while ΔQ , ∇Q , and $\boxminus Q$ can safely be ignored by the CDC technique.

4.5 Summary

In this section we exemplified rewriting a static OHM instances by replacing each basic OHM operator with its incremental variant. In each subsection, we first presented the most general incremental operator variant and subsequently discussed adaptations to partial change data. We further exploited knowledge about source data to reduce the complexity of the incremental JOIN variant. The rewrite process was performed in a source-to-target direction since CDC limitations were provided as a starting point. The resulting incremental OHM instance reveals that change data propagation yields to partial change data: Inserts and updates are provided in an indistinguishable manner while deletions are unavailable. This has to be taken into account when choosing an appropriate CDA technique. Note that the rewrite process can also be performed in a target-to-source direction with CDA requirements given as a starting point. In this case the resulting incremental OHM instance reveals acceptable CDC limitations with regard to each data source.

5. RELATED WORK

Incremental warehouse loading is clearly related to incremental maintenance of materialized views [14, 20, 12, 21, 22] as both areas address incremental updates of physically integrated data. The approach presented in this paper is similar to the *algebraic* approaches to incremental maintenance of materialized views in the sense that we use the same “language” to express both, static and incremental transformation semantics. While incremental view maintenance relies on the relational algebra, we make use of OHM operators. The benefit is that existing infrastructure can be reused in either case. Traditional database query optimizers are exploited for incremental view maintenance while we leverage Orchid’s deployment facility. Our model for change data, however, differs significantly from approaches to incremental view maintenance. The latter treat updates as pairs of deletions and insertions during change propagation. We opt to model and propagate updates explicitly for two reasons.

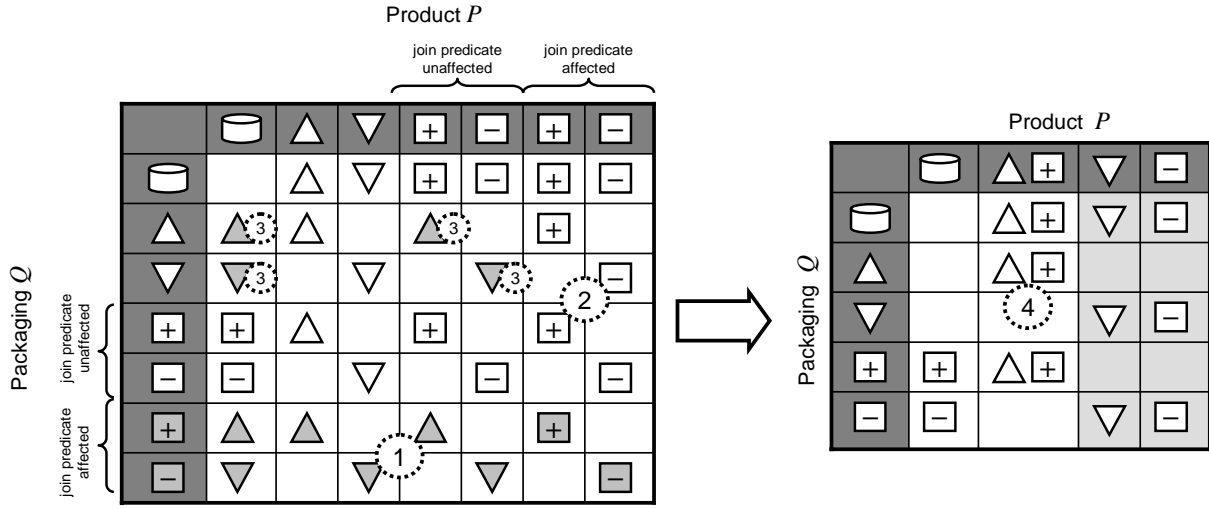


Figure 6: Join Matrix Simplifications

- A materialized view and its source relations are managed by the same database system. In consequence, full information about changes to source relation is available. In data warehouse environments CDC techniques are applied that may suffer from limitations and miss certain changes. Our notion of partial change data naturally covers this case. It is unclear how partial change data could be modeled in terms of insertions and deletions only.
- Furthermore, data warehouses typically retain *historical data* while materialized views reflect the most current state of the base relations at any time. Sophisticated update strategies exist for data warehouses to maintain a history of data. An update generally affects the data history differently than a delete-insert pair leading to an incorrect state of the warehouse.

Work on *view maintenance in data warehouse environments* [13, 23, 7] effectively studies maintenance of materialized views derived from distributed autonomous data sources. This notion of data warehouse environments significantly differs from ours. It is assumed that data warehouses are refreshed instantly in response to source changes rather than on a periodical basis. Data sources are assumed to offer sophisticated query capabilities. They are further assumed to answer frequent queries from the data warehouse and tolerate the additional workload. The fact that data warehouses store historical data is not taken into considerations. We argue that this notion of data warehousing substantially differs from real-world scenarios we are addressing.

The most extensive study on the modeling of ETL processes is by Simitsis, Vassiliadis, et al. [29, 24, 27]. The authors propose both, a conceptual model and a logical model for the representation of ETL processes. The conceptual model maps attributes of the data sources to the attributes of the data warehouse tables. The logical model describes the flow of data from the sources towards the data warehouse. Thus, the conceptual model and the logical model roughly correspond to schema mappings and OHM instances, respectively. In [25] a method for the translation of conceptual models to logical models is presented. In [26]

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 fact, the authors focus on initial load scenarios only.

In the last decade the market for ETL software has steadily grown. Numerous commercial ETL tools are available today [2, 5, 1, 3]. However, a standardized ETL language does not exist. ETL tools rely on proprietary scripting languages or visual user interfaces. With Orchid’s Operator Hub Model, a platform-independent model has been introduced that captures data transformation capabilities common to various ETL tools. We stress that neither Orchid nor our approach competes with existing ETL tools. In fact, we perceive them as potential deployment platforms for OHM instances. We focus on the generation of ETL jobs rather than on their execution.

We further emphasize that existing ETL tools are quite capable of performing incremental loads. This is obvious, considering that both, initial load processes and incremental load processes are assembled using the same transformation primitives. However, we are not aware of any ETL tool that supports incremental loading explicitly and shields the developer from the complexity. In fact, ETL jobs tailored for incremental loading need to be handcrafted as yet. This is a complex task reserved to trained ETL developers. Our approach allows for generating incremental load processes from high-level schema mappings. Compared to known solutions, it is significantly easier to use and addresses a much wider range of users. Most important, the generated jobs are guaranteed to work correctly.

6. CONCLUSION

We presented an approach for generating *incremental load processes* for data warehouse refreshment from *declarative schema mappings*. The basis of our work has been provided by Orchid, a prototype system developed at IBM Almaden Research Center, that translates *schema mappings* into ETL processes and vice versa. Orchid-generated ETL processes, however, are limited to *initial load scenarios*, i.e. source data is exhaustively extracted and the warehouse is com-

pletely (re)built. Incremental load processes, in contrast, propagate changes from the sources to the data warehouse. This approach has clear performance benefits. Change Data Capture (CDC) and Change Data Application (CDA) techniques are used to capture changes at the sources and refresh the data warehouse, respectively. We defined a model for *change data* to characterize both, the output of CDC techniques and the input of CDA techniques. Since CDC techniques may suffer from limitations we introduced a notion of *partial change data*. We discussed the propagation of partial change data within incremental load processes. Our approach allows for reasoning on how limitations of CDC techniques determine the set of applicable CDA techniques. That is, it allows inferring satisfiable CDA requirements from given CDC limitations and, the other way round, acceptable CDC limitations from given CDA requirements. We further, demonstrated the exploitation of properties of data sources (such as schema constraints) to reduce the complexity of incremental load processes. By leveraging Orchid's deployment facilities, we are able to generate executable ETL processes. We are confident that our work contributes to the improvement of ETL development tools.

Future work will focus on the generation of incremental variants of aggregation operators. We further plan to investigate the usage of the staging area, i.e. allow ETL processes to persist data that serves as additional input for 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, non-distributive aggregate functions like minimum or maximum over deletions depend on the staging area since an incremental computation based on the previous results and change data alone is impractical. Using the staging area, however, comes at the cost of additional IO operations. We plan to extend our approach to involve the staging area where appropriate.

7. REFERENCES

- [1] IBM DB2 Data Warehouse Enterprise Edition. www.ibm.com/software/data/db2/dwe/.
- [2] IBM WebSphere DataStage. <http://www-306.ibm.com/software/data/integration/datastage/>.
- [3] Informatica PowerCenter. http://www.informatica.com/products_services/powercenter/.
- [4] Oracle Database Change Data Capture. <http://www.oracle.com/database>.
- [5] Oracle Warehouse Builder. <http://www.oracle.com/technology/products/warehouse/index.html>.
- [6] WebSphere Replication Server (SQL replication). http://www-306.ibm.com/software/data/integration/replication_server/.
- [7] D. Agrawal, A. E. Abbadi, A. K. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 417–427. ACM Press, 1997.
- [8] J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.
- [9] M. Bokun and C. Taglienti. Incremental Data Warehouse Updates. *DM Review Magazine*, May 1998.
- [10] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.
- [11] S. Dessloch, M. A. Hernández, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: Integrating Schema Mapping and ETL. In *ICDE*, pages 1307–1316, 2008.
- [12] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *SIGMOD Conference*, pages 328–339, 1995.
- [13] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data Integration using Self-Maintainable Views. In *EDBT*, pages 140–144, 1996.
- [14] H. Gupta and I. S. Mumick. Incremental Maintenance of Aggregate and Outerjoin Expressions. *Inf. Syst.*, 31(6):435–464, 2006.
- [15] R. Kimball and J. Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. John Wiley & Sons, Inc., 2004.
- [16] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
- [17] W. Labio and H. Garcia-Molina. Comparing Very Large Database Snapshots. Technical Report STAN-CS-TN-95-27, Computer Science Department, Stanford University, June 1995.
- [18] W. Labio and H. Garcia-Molina. Efficient Snapshot Differential Algorithms for Data Warehousing. In *VLDB*, pages 63–74, 1996.
- [19] U. Leser and F. Naumann. *Informationsintegration*. dpunkt.verlag, 2007.
- [20] M. K. Mohania, S. Konomi, and Y. Kambayashi. Incremental Maintenance of Materialized Views. In *DEXA*, pages 551–560, 1997.
- [21] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental Maintenance for Non-Distributive Aggregate Functions. In *VLDB*, pages 802–813, 2002.
- [22] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.*, 3(3):337–341, 1991.
- [23] D. Quass. Maintenance Expressions for Views with Aggregation. In *VIEWS*, pages 110–118, 1996.
- [24] A. Simitsis. Modeling and managing ETL processes. In *VLDB PhD Workshop*, 2003.
- [25] A. Simitsis. Mapping conceptual to logical models for etl processes. In *DOLAP*, pages 67–76, 2005.
- [26] A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing ETL Processes in Data Warehouses. In *ICDE*, pages 564–575, 2005.
- [27] A. Simitsis, P. Vassiliadis, M. Terrovitis, and S. Skiadopoulos. Graph-Based Modeling of ETL Activities with Multi-level Transformations and Updates. In *DaWaK*, pages 43–52, 2005.
- [28] T. Urpí and A. Olivé. A Method for Change Computation in Deductive Databases. In *VLDB*, pages 225–237, 1992.
- [29] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Conceptual modeling for ETL processes. In *DOLAP*, pages 14–21, 2002.