

## On Deriving Net Change Information From Change Logs – The DELTALAYER-Algorithm –

Stefanie Rinderle<sup>1</sup>, Martin Jurisch<sup>1</sup>, Manfred Reichert<sup>2</sup>

<sup>1</sup>Institute DBIS, Ulm University, Germany, {stefanie.rinderle, martin.jurisch}@uni-ulm.de

<sup>2</sup>IS Group, University of Twente, The Netherlands, m.u.reichert@utwente.nl

**Abstract:** The management of change logs is crucial in different areas of information systems like data replication, data warehousing, and process management. One barrier that hampers the (intelligent) use of respective change logs is the possibly large amount of unnecessary and redundant data provided by them. In particular, change logs often contain information about changes which actually have had no effect on the original data source (e.g., due to subsequently applied, overriding change operations). Typically, such inflated logs lead to difficulties with respect to system performance, data quality or change comparability. In order to deal with this we introduce the DeltaLayer algorithm. It takes arbitrary change log information as input and produces a cleaned output which only contains the net change effects; i.e., the produced log only contains information about those changes which actually have had an effect on the original source. We formally prove the minimality of our algorithm, and we show how it can be applied in different domains; e.g., the post-processing of differential snapshots in data warehouses or the analysis of conflicting changes in process management systems. Altogether the ability to purge change logs from unnecessary information provides the basis for a more intelligent handling of these logs.

## 1 Introduction

### 1.1 Problem Description

The management of log information is crucial in different areas of information systems. In addition to, for example, transaction logs in database management systems the optimized processing of change (log) information (e.g., differential snapshots [LGM96]) is gaining more and more importance. In this context, one prominent example is the update of a data warehouse based on change information. More precisely, for such an update heterogeneous data sources are monitored. According to the particular monitoring strategy, updates within the data sources are then fed into the staging area of the data warehouse (cf. Figure 1a). How these data updates can be determined depends on the kind of data sources. For example, database systems offer mechanisms such as replication or triggers in this context which ease the extraction and processing of update information. However, very often data updates have to be extracted from sources (e.g., legacy systems) which do not offer any support for getting this update information. In such cases, the only way is to produce *snapshots* of the data sources before and after the update and to calculate the difference (i.e., the *differential snapshot*) between them [LGM96, JNS<sup>+</sup>97] (cf. Figure 1a). A differ-

ential snapshot can be expressed by a set of change operations (i.e., INSERT, UPDATE, DELETE operations) leading from the source data to the target data.

In addition to data warehouses, change information also plays an important role in the domain of adaptive process management technology [RRD04b]. Such systems enable process changes at different levels, which occur frequently in practice [Wes01]: Single process instances may have to be adapted, for example, to deal with exceptional situations (e.g., by adding, deleting, or shifting process steps, cf. Figure 1b). Furthermore process templates<sup>1</sup> may have to be changed, e.g., to react on new regulations or to implement process optimizations. The information about these changes has to be stored for several reasons. In [RRJK06] we discuss traceability (e.g., for the medical domain) and correctness checks in the context of concurrent process changes as important use cases.

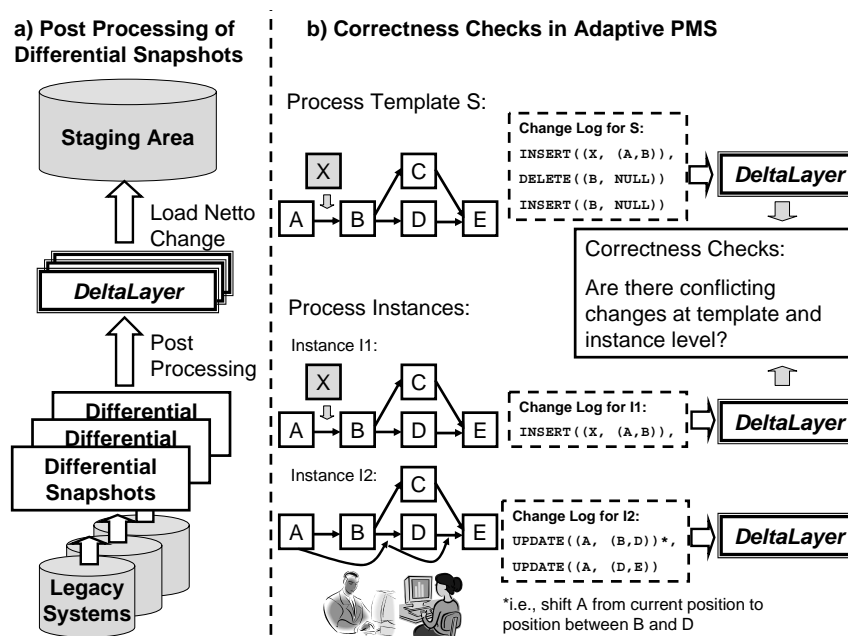


Figure 1: Different Use Cases for the DELTALAYER Approach

Both application domains, data warehouses and process management systems, show that change logging is very important. However, one problem hampers the management and the use of change logs: Very often, change logs consists of a possibly large amount of unnecessary information. In data warehouses, for example, this information comprises wasteful DELETE/INSERT or INSERT/DELETE pairs within the differential snapshots; these pairs are often produced when calculating the differential snapshots; a wasteful DELETE/INSERT pair expresses the deletion of a database entry X followed by the insertion of X (i.e., the same entry) in the sequel. The same problem arises in the context of

<sup>1</sup>Process templates describe the general structure of a process (e.g., control and data flow), cf. Figure 1b.

process management systems where change logs may contain change operations overriding the effects of previous changes. One example is illustrated for process instance I2 in Figure 1b: The first user updates the position of process activity A to (B, D) followed by an update of the second user to position (D, E) (i.e., the overall effect for both UPDATE operations is UPDATE (A, (D, E) ). The presence of such unnecessary information within change logs may cause a number of problems and difficulties:

- *Performance problems due to inflated logs:* In case of differential snapshots this may lead to performance problems in case of tight update windows [DDJ<sup>+</sup>98].
- *Data quality problem:* In data warehouses, redundant information from differential snapshots has to be cleaned within the staging area. Redundant change information might be desirable for traceability reasons on the one hand. On the other hand, DELETE/INSERT and INSERT/DELETE pairs do not have any real background and therefore should to be cleaned before analyzing the data.
- *Comparability problem:* In process management systems many correctness checks (e.g., dealing with concurrently applied changes) are based on comparing change logs. In this context, unnecessary information leads to non-comparable change logs, which significantly hampers correctness checks in the sequel [RRD04a, RRJK06].

This paper aims at tackling these problems which might be relevant for other use cases as well (e.g., when updating materialized views in data warehouses).

## 1.2 Contribution

In [RRJK06] we have already discussed requirements for capturing (process) change logs in adaptive process management systems. We have also shown how to purge unnecessary information from process change logs at the logical level. In this paper, we present the DELTALAYER algorithm which takes arbitrary change log information as input and produces a cleaned output which only contains the net change effects (i.e., information about changes which actually have had an effect on the database, the data source, or the process repository). The resulting net changes are considered as being *correct* if they are minimal with respect to the sets of inserted, deleted, and updates tuples. This constitutes an extension of the minimality notion as used in, for example, approaches for updating materialized views [GL95]. When compared to approaches on transaction equivalence [AV88] which mainly operate at the logical level, our approach tackles the challenge to efficiently realize minimized change logs at the physical level. Contrary to [AV88], for example, the *DeltaLayer* algorithm produces output (i.e., minimized change logs) which can be directly used for data extraction in data warehouses or correctness checks within a process management system. This also implies that our approach is independent of any transactional concepts (i.e., transactions and associated recovery mechanisms can be specified on top of the change logs information as considered in this paper).

Since the output of the DELTALAYER algorithm is *structured* it can be accessed and queried more easily, for example, to post-process the net changes: When updating materialized views in data warehouses, it might be useful to determine which attributes are affected by changes and which are not. Regarding adaptive process management systems the access to specific entities (such as process activities) support the efficient processing of the necessary correctness checks (e.g., whether two changes are conflicting) [RRD04a].

After presenting the DELTALAYER algorithm we will show how it can be implemented within commercial systems such as Oracle database or IBM DataPropagator as well as within differential snapshot algorithms (e.g., the *Window* algorithm as, for example, presented in [LGM96]) in order to purge the output from wasteful information. For applications such as data warehouses, it is sufficient that the DELTALAYER algorithm is able to process the basic operations DELETE, UPDATE, and INSERT [LGM96]. However, it offers the possibility to process more complex changes on top of these basic operations as well (e.g., transactions, set-oriented updates, or "high-level" process graph changes). We will show this by means of an example for a "high-level" operation in the area of adaptive process management systems. In this context we can also see that the minimality property of the DELTALAYER algorithm is crucial. Minimality guarantees that two arbitrary change logs become comparable after running the DELTALAYER algorithm on them. This, in turn, is essential for any check which requires comparison of change information.

The paper is organized as follows: In Section 2 fundamental notions are introduced. We present the DELTALAYER algorithm and prove its minimality property in Section 3. Section 4 discusses several applications of the DELTALAYER algorithm. In Section 5 we discuss related work, and we finish with a summary and outlook in Section 6.

## 2 Fundamentals

In this section we introduce fundamental notions which we use further on in the paper and which are necessary for a basic understanding.

First of all, we restrict our considerations to the following set of basic change operations:  $CO := \{\text{INSERT}(T), \text{UPDATE}(T), \text{DELETE}(T)\}$ <sup>2</sup>. We assume that tuple  $T$  has a key  $T.Key$  and a list of attributes  $T.A$  with  $A = [a_1, \dots, a_n]$ . Using this notion, for example, differential snapshots can be easily expressed:  $\text{INSERT}(T)$  refers to the insertion of a data field with key  $T.Key$  and attribute list  $T.A$  (cf. [LGM96]). Regarding change logs in adaptive process management systems,  $\text{INSERT}(T)$  expresses the insertion of a new activity node  $T.Key$  into the underlying process template (e.g., activity node  $X$  is inserted into process template  $S$ , cf. Figure 1b). The additional parameters stored within  $T.A$  might, for example, specify the position where  $T.Key$  is inserted. Furthermore,  $\text{UPDATE}(T)$  expresses an update of a data field having key  $T.Key$  with attributes  $T.A$  or an update of the position of a process activity node  $T.Key$  within a process template (e.g., updating the current position of activity node  $A$  to new position  $(D, E)$  for instance  $I2$ , cf. Figure 1b). Finally, for a delete operation it is sufficient to only specify the key of

---

<sup>2</sup>In Section 4.3 we show how our approach can be extended to other change operations as well.

the tuple to be deleted.

In the following we denote attribute values which remain non-specified as NULL. This might be used, for example, when applying UPDATE operations for which only some of the attributes are updated whereas for the others the old values are kept (cf. Table 1).

One or several change operations as described above can be used to express modifications on data sources (e.g., update of source tables contained in differential snapshots). More precisely, we denote an ordered sequence of change operations as *change log*. Formally:  $cL := \langle op_1, \dots, op_n \rangle$  where  $op_i \in CO$  ( $i = 1, \dots, n$ ) and  $op_i$  affects tuple T having key  $T.Key$  and attribute list  $T.A$ . Note that an operation  $op_i$  is performed before another operation  $op_j$  ( $op_i, op_j \in cL$ ) for  $i < j$ . As running example we use the following simple change log for the remainder of this paper:

**Change Log 1 (Example Change Log) <**

```
op1 = INSERT ((2, "HELLO")),  
op2 = INSERT ((3, "BYE")),  
op3 = DELETE ((2, NULL)),  
op4 = INSERT ((4, "BYEBYE")),  
op5 = UPDATE ((4, "HELLO")),  
op6 = UPDATE ((4, "BYE")),  
op7 = DELETE ((5, NULL)),  
op8 = INSERT ((5, "CIAO")),  
op9 = INSERT ((6, "HI")),  
op10 = INSERT ((7, "HIHI")) >
```

When analyzing this sample change log we can see that its actual effects on the original data source can be captured by a fraction of the original change operations, i.e.,

```
<INSERT ((3, "BYE")), INSERT ((4, "BYE")), UPDATE ((5, "CIAO"))3,  
INSERT ((6, "HI")), INSERT (7, "HIHI")>.
```

From literature studies and experiences with our process management system we learned that unnecessary change pairs within change logs occur rather often. Examples are differential snapshots [LGM96] and change logs in process management systems [RRJK06]. Regarding the first case, wasteful change pairs are produced by the algorithms to compute the differential snapshots; in the latter case the unnecessary information might be caused by users trying out the best solutions for change operations (see, for example, instance I2 in Figure 1b). In any case such wasteful change pairs might cause performance problems [DDJ<sup>+</sup>98] and / or hamper correctness checks based on the change logs [RRJK06]. In Table 1 we summarize all different kinds of wasteful change pairs which might occur within change logs. Table 1 also analyzes their actual effects on the data source.

---

<sup>3</sup>In this case we have to check if attribute value CIAO means an update on the original table, otherwise the INSERT ((5, "CIAO")) operation would be unnecessary too.

Table 1: *Wasteful Change Pairs within Change Logs*

Change Pair	Effect	Example
1. INSERT (T) /DELETE (T)	neutral	INSERT ( (2, "HELLO") ), DELETE ( (2, NULL) )
2. UPDATE (T) /DELETE (T)	delete	UPDATE ( (2, "HELLO") ), DELETE ( (2, NULL) ) ⇒ DELETE ( (2, NULL) )
3. DELETE (T) /INSERT (T)	a) same attribute values as tuple (2, ..) had in source before ⇒ neutral b) different attribute values as tuple (2, ..) had in source before ⇒ update	DELETE ( (2, NULL) ), INSERT ( (2, "HELLO") ) DELETE ( (2, NULL) ), INSERT ( (2, "BYE") ) ⇒ UPDATE ( (2, "BYE") )
4. UPDATE (T) /UPDATE (T)	a) second update has same attribute values as T in source ⇒ neutral  b) updates attribute values compared to attribute values of T in source ⇒ update  c) value update for more than one attribute ⇒ merge into one update	original attribute values of T in source (2, "HELLO") UPDATE ( (2, "BYE") ), UPDATE ( (2, "HELLO") ) original attribute values of T in source (2, "HELLO") UPDATE ( (2, "BYE") ), UPDATE ( (2, "BYEBYE") ) ⇒ UPDATE ( (2, "BYEBYE") ) original attribute values of T in source (2, "HELLO", 100) UPDATE ( (2, "BYE", NULL) ) UPDATE ( (2, NULL, 300) ) ⇒ UPDATE ( (2, "BYE", 300) )
5. INSERT (T) /UPDATE (T)	a) for one attribute ⇒ insert  b) for more than one attribute ⇒ merge into insert operation	INSERT ( (2, "HELLO") ), UPDATE ( (2, "BYE") ) ⇒ INSERT ( (2, "BYE") ) INSERT ( (2, "BYE", NULL) ) UPDATE ( (2, NULL, 11) ) ⇒ INSERT ( (2, "BYE", 11) )

### 3 On Calculating the Minimal Effect of Change Logs – the DELTALAYER Algorithm

So far we have discussed different kinds of wasteful change combinations within change logs (cf. Section 2). In this section we present the DELTALAYER algorithm which purges a change log from change combinations as summarized in Table 1 and produces a "net information" output (i.e., an output only containing change operations which reflect the delta between target and source table).

#### 3.1 The DELTALAYER Algorithm

Generally, the DELTALAYER algorithm (cf. Algorithm 1) receives a change log as input (such as Example 1) and produces an output within the *DeltaLayer* format<sup>4</sup> [RRJK06]

<sup>4</sup>We abstract from a concrete representation since different applications usually require different approaches in this context. One possibility for a *DeltaLayer* representation is presented in Figure 2. We will discuss other approaches as used in commercial systems (e.g., the condensed change data table format in IBM Data Propagator) in Section 4.1. Here, for example, the entities affected by change operations  $op_i$  are stored within one

which only contains the net information of the input change log. In (1) the *DeltaLayer* structure is initialized; either a new *DeltaLayer* structure is created or an already existing one is used. Then the algorithm steps through the change log (i.e., an ordered sequence of change operations).

**Algorithm 1 (DELTALAYER: ChangeLog  $cL := \langle op_1, \dots, op_n \rangle \mapsto \text{DeltaLayer } D_{Net}$ )**

```

DeltaLayer  $D_{Net}$  := newDeltaLayer
forall (i = 1, ..., n) do
  if  $op_i = \text{INSERT}(T)$  then
     $D_{Net} = \text{insertTuple}(D_{Net}, T)$ 
  else if  $op_i = \text{DELETE}(T)$  then
     $D_{Net} = \text{deleteTuple}(D_{Net}, T)$ 
  else if  $op_i = \text{UPDATE}(T)$  then
     $D_{Net} = \text{updateTuple}(D_{Net}, T)$ ;
fi
od

```

For each change operation  $op_i$  its type is determined (i.e., INSERT, UPDATE, DELETE). Based on the current change type, Algorithm 1 calls associated functions `insertTuple(...)`, `deleteTuple(...)`, or `updateTuple(...)`. Basically, these functions check if and what effect current operation  $op_i$  has on the target data table. For this decision the effects of already applied change operations have to be taken into account (cf. Table 1).

Let us assume that Algorithm 1 calls function `insertTuple(...)` with the current *DeltaLayer* and the current change operation  $op_i = \text{INSERT}(T)$ . First of all, function `insertTuple(...)` checks whether tuple T has already been deleted before (i.e., by a change operation  $op_k$  with  $k < i$ ). This corresponds to case 3 in Table 1. As we can see from this table, function `insertTuple(...)` now has to distinguish whether the attribute values used when applying  $op_i$  are the same or different from the attribute values having been used for prior  $op_k = \text{DELETE}(T)$  operation (i.e., the original values in the source table in case of a DELETE operation). This is done in lines (\*). If the attribute values used by  $op_i$  and  $op_k$  are different, obviously,  $op_i$  has to be stored within the *DeltaLayer* as UPDATE(T) operation (i.e., if `update = true` (◇)). In any case, T is deleted from the set of deleted tuples and therefore the previous DELETE(T) operation  $op_k$  is correctly purged from *DeltaLayer*. Except the DELETE(T)/INSERT(T) combination, according to Table 1, there is no other possibly wasteful change combination where an INSERT operation is applied in the second place. Therefore, if  $T \notin D_{old}.\text{deletedTuples}$ , T can be added to the set of inserted tuples and INSERT(T) is currently stored within *DeltaLayer*.

**Function 1 (insertTuple (DeltaLayer  $D_{old}$ , newTupel  $T_{new}$ )  $\mapsto$  DeltaLayer  $D_{Net}$ )**

```

 $D_{Net} := D_{old}$ 
boolean contains := false

```

table including a column where the type of the particular change operation is indicated (e.g., I for an INSERT operation). Another possibility would be to keep an own table for INSERT, UPDATE, and DELETE operations. The choice may depend on factors such as performance or query optimization.

```

forall (T ∈ Dold.deletedTuples) do //†
if (T.Key = Tnew.Key) then //‡
  boolean update := false
  Tupdate = new T<Tnew.Key, {}>
  n = |T.A|
  forall (k = 1, ..., n) do
    if (T.ak ≠ Tnew.ak) then //★
      Tupdate.ak = Tnew.ak //★
      update = true
    fi
  od
  if (update = true) then //◇
    DNet.updatedTuples := DNet.updatedTuples ∪ Tupdate//◇
  fi
  DNet.deletedTuples := DNet.deletedTuples \ T
  contains := true
  break
  fi
od
if (contains = false)
  DNet.insertedTuples := DNet.insertedTuples ∪ Tnew
fi
return DNet

```

We omit the code for functions `deleteTuple(...)` and `updateTuple(...)` due to space restrictions, but explain their essence in the following: For  $op_i = \text{DELETE}(T)$ , if there has been a previous operation  $op_k$  with  $op_k = \text{INSERT}(T)$  ( $k < i$ ),  $op_k$  and  $op_i$  are purged from *DeltaLayer* by function `deleteTuple(...)`. Reason is that combination  $\text{INSERT}(T)/\text{DELETE}(T)$  has no actual effect according to case 1 in Table 1. By contrast, case 2 in Table 1 is more interesting: Although there has been a previous  $\text{UPDATE}(T)$  operation, we have to insert the original attribute values into the *DeltaLayer* which have been contained within the source table before any change operation has been applied. Otherwise the *DeltaLayer* does not reflect correctly applicable changes. To get correct values it is not sufficient to check the source table due to intermediate (and already purged)  $\text{UPDATE}$  operations. Therefore, one alternative is to look up the original values from the target table which might decrease the performance of the algorithm (see Section 3.2). However doing so does not cause a blocking of the target table and therefore does not lead to an increase of the update window. Alternatively, the original attributes can be stored in an auxiliary data structure as soon as an  $\text{UPDATE}$  operation is applied. This raises storage needs but increases performance. The mechanisms used in function `updateTuple(...)` can be seen as a combination of those ones used for functions `insertTuples(...)` and `deleteTuples(...)`.

Figure 2 shows how the DELTALAYER algorithm works on the input changes presented in Example 1. The crossed-out entries reflect the purged entries within the *DeltaLayer*.

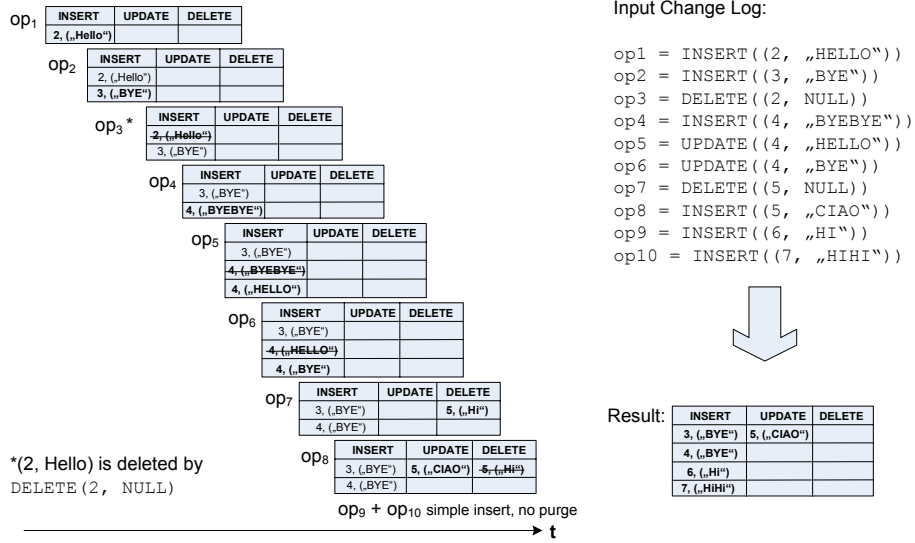


Figure 2: Applying DELTALAYER to Example Change Log

Note that the algorithm can also be used if the Delta Layer is updated each time a change operation occurs, i.e., the input of algorithm 1 becomes a one-element change log. This approach is, for example, applied in our adaptive process management system ADEPT2, i.e., each time a change operation is applied either to a process instance or to a process template, the according Delta table is updated using the DELTALAYER algorithm. In Section 4.2 we will show how the same principle can be directly integrated within the *Window* algorithm producing differential snapshots.

### 3.2 Performance Considerations for the DELTALAYER Algorithm

In this section, we provide some performance considerations for the DELTALAYER algorithm. Obviously, the complexity of Algorithm 1 without considering functions `insertTuple(...)`, `deleteTuple(...)`, and `updateTuple(...)` is  $O(n)$  with  $n$  equals the number of change operations contained in the input change log. More interesting are the complexity considerations for functions `insertTuple(...)`, `deleteTuple(...)`, and `updateTuple(...)`.

For function `insertTuple(...)` we obtain a complexity of  $O(n * c) = O(n)$  since this function consists of a scan of set `deletedTuples` with  $n$  elements  $\implies O(n)$  (cf line †), a simple comparison in line (‡), and a check of all  $c$  attributes of  $T \implies O(c)$  with  $c$  is constant. Furthermore, adding  $T$  to the set of updated or inserted tuples as well as deleting  $T$  from the set of `deletedTuples` (if necessary) is all of complexity  $O(1)$ .

Function `deleteTuple(...)` also has a complexity of  $O(n)$ : The check of the set of inserted tuples has a complexity of  $O(n)$  ( $n$  equals the number of inserted tuples). The scan of the set of updated tuples is of complexity  $O(m)$  with  $m$  equals the number of updated tuples. Therefore we obtain a complexity of  $O(n + m) = O(n)$  for `deleteTuple(...)` (delete and insert into the sets of inserted, deleted, and updates tuples are again of a complexity of  $O(n)$ ).

For function `updateTuple(...)` we obtain a worst case complexity of  $O(n^2)$ : First of all there is a scan of the set of inserted tuples having complexity of  $O(n)$  with  $n$  equals the number of inserted tuples. The following check of the attribute set of a tuple  $T$  has complexity of  $O(c)$  with  $c$  equals the number of attributes of  $T$ . The conditional scan of the set of updated tuples is of complexity  $O(m)$  ( $m$  equals the number of updated tuples). Then the number of attributes of tuple  $T$  is checked again with complexity of  $O(k)$  with  $k$  equals the number of attributes of tuple  $T$ . Then the target table has to be accessed with complexity of  $O(t)$  with  $t$  equals the number of tuples in the target table. Note that here no locks on the target table become necessary. Furthermore we can decrease the complexity by storing the original attribute values of the affected tuples. Finally, a scan of the attributes of tuple  $T$  in the target table becomes necessary having complexity  $O(l)$  with  $l$  equals the number of attributes of tuple  $T$ . Altogether, the complexity of `updateTuple(...)` turns out as  $O(n * c) + O(m * k * t * l)$  where  $c$ ,  $k$ , and  $l$  are constant. Consequently, the complexity is  $O(n) + O(m * t) = O(n) + O(n^2) = O(n^2)$ .

Generally speaking, the number of the tuple in the *DeltaLayer* is quite small when compared to the number of tuples in source and target tables since only the minimal set of change operations is stored. Therefore scanning the sets of inserted, updated, and deleted tuples can be accomplished rather quickly. Generally, processing DELTALAYER algorithm is not the part potentially causing performance problems. The critical part is feeding the data into the data warehouse afterwards since the target tables have to be locked during this time (update window). The DELTALAYER algorithm minimizes the volume of the data to be fed into the data warehouse significantly and therefore can increase the update window quite dramatically. Furthermore, as we will show in Section 4.2, the DELTALAYER algorithm has not to be applied once for all (and possibly large) change logs. It is possible to directly integrate the DELTALAYER algorithm into, for example, differential snapshot algorithms. Therefore the different data mechanisms can work with the DELTALAYER algorithm asynchronously regarding the time updates on the data tables occur.

### 3.3 Correctness of the DELTALAYER Algorithm

In this section we show that the output produced by Algorithm 1 is correct, i.e., the output captured within the *DeltaLayer* is minimal. Minimality has been considered as important requirement in the context of updating materialized views as well. For example, in [GL95] minimality of the bag algebra expressions for updating a derived view has been proven for INSERT and DELETE operations. UPDATE operations applied to the base view have not been directly taken into account since they have been always transformed into associated DELETE/INSERT pairs. This is contrary to minimality itself.

In the following, we will adopt the minimality requirements for INSERT and DELETE operations for our algorithmic approach but will extend it by considering UPDATES.

**Definition 1 (Minimality of the DeltaLayer)** Let  $S$  be the tuple set before the change and let  $S'$  be the tuple set after applying change operations  $op_i$  ( $i = 1, \dots, n$ ) captured within change log  $cL = \langle op_1, \dots, op_n \rangle$ . Let further the sets of actually inserted, deleted, and updated tuples be defined as follows:

1. Actually deleted tuples:  $\nabla cL := \{t \mid t \in S \setminus S': \nexists t' \in S' \text{ with } t'.Key = t.Key\}$
2. Actually inserted tuples:  $\Delta cL := \{t \mid t \in S' \setminus S: \nexists t' \in S \text{ with } t'.Key = t.Key\}$
3. Actually updated tuples:  $\triangleright cL := \{t \mid t \in S \setminus S' \vee t \in S' \setminus S: \exists t' \in S' \text{ with } t'.Key = t.Key\}$

Then we call a change log or related data structure (e.g., a delta layer)  $D$  minimal if and only if the following conditions hold:

1.  $D.deletedTuples = \nabla cL$
2.  $D.insertedTuples = \Delta cL$
3.  $D.updatedTuples = \triangleright cL$

The definitions of  $\nabla cL$ ,  $\Delta cL$ , and  $\triangleright cL$  are illustrated in Figure 3a from a set-based point of view. As it can be seen from Figure 3b the distinction between updated and inserted or updated and deleted tuples is done by comparing keys.

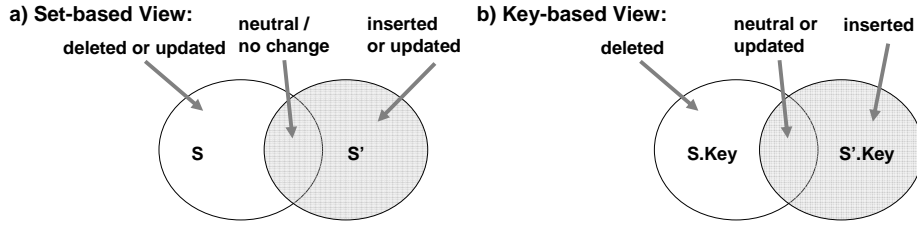


Figure 3: Sets of Inserted, Deleted, and Updated Tuples

**Theorem 1 (Minimality of DeltaLayer)** Let  $S$  be a tuple set and let  $cL$  be a change log transforming tuple set  $S$  into tuple set  $S'$ . Let further DeltaLayer  $D$  be the output resulting from the application of Algorithm 1 to  $cL$ . Then  $D$  is minimal according to Definition 1.

Due to lack of space we only sketch the proof of Theorem 1 for the set of inserted tuples  $D.insertedTuples$ . The proofs for deleted and updated tuples, however, can be accomplished in a similar way by contradiction.

**Proof Sketch:**

Proof by contradiction  $\implies$  we have to show

1.  $D.deletedTuples \neq \nabla cL \vee$
2.  $D.insertedTuples \neq \Delta cL \vee (*)$
3.  $D.updatedTuples \neq \triangleright cL$

$\implies \neg$  (D produced by Algorithm 1)

We proof  $(*) \implies \neg$  (D produced by Algorithm 1).

Auxiliary assumptions: a)  $\forall t$  used in the following  $t \in S \vee t \in S'$  holds, b) INSERT ( $t$ ) is the last entry in  $cL$  for  $t$ , c) it is not possible to apply an INSERT ( $t$ ) operation after an UPDATE ( $t$ ) operation (would be rejected by the database system,

$D.insertedTuples \neq \Delta cL \equiv$

$D.insertedTuples \neq \{t \mid t \in S' \setminus S: \nexists t' \in S \text{ with } t'.Key = t.Key\} \implies$   
 $\exists t1 \in D.insertedTuples$  with

1.  $t1 \in S \setminus S' \vee$
2.  $t1 \in S \cap S' \vee$
3.  $(t1 \in S' \setminus S \wedge \exists t2 \in S \text{ with } t1.Key = t2.Key)$

$\implies$

1)  $t1 \in S \setminus S' \implies t1$  has been deleted or updated  $\implies$  contradiction to (34) of Algorithm 1 and to b) and c) of auxiliary assumptions

2)  $t1 \in S \cap S' \implies (t1$  has not been affected by any change)  $\vee$  INSERT ( $t1$ )/DELETE ( $t1$ ) pair in  $cL$  (with same attributes)  $\vee$  DELETE ( $t1$ )/INSERT ( $t1$ ) pair in  $cL \vee$  UPDATE ( $t1$ ) / UPDATE ( $t1$ ) pair in  $cL$  (with original values of  $t1$  in  $S$  for last update)<sup>5</sup>  
 $\implies$  contradiction to 1)  $\vee$  contradiction to (14,28,33)  $\vee$  contradiction to (40,42,56,57)  $\vee$  contradiction to (76,77,97) of Algorithm 1

3)  $(t1 \in S' \setminus S \wedge \exists t2 \in S \text{ with } t1.Key = t2.Key) \implies$  UPDATE ( $t1$ ) in  $cL \implies$  contradiction to (63) of Algorithm 1 and to b) and c) of auxiliary assumptions  $\square$

## 4 On Applying the DELTALAYER Algorithm

In this section we provide different application scenarios for the DELTALAYER algorithm. They range from commercial systems such as Oracle or IBM DataPropagator to an extension of the used change operations within process management systems.

<sup>5</sup>For all change combinations see Table 1.

## 4.1 Application within Commercial Systems

In the following, we show how the DELTALAYER algorithm can be applied in the context of two main commercial systems which offer support for data warehouses:

*Oracle Database:* Within the data warehousing guide of Oracle [Dat03] different possibilities for data change capture are summarized. Closely related to the problems described in this paper is the *merge* statement which is an extension of SQL. Basically, the merge statement is used to transform data from source tables into the format of the target table.

```
(1) MERGE INTO products t USING products_delta s
(2) ON (t.prod_id=s.prod_id)
(3) WHEN MATCHED THEN UPDATE SET
(4) t.prod_list_price=s.prod_list_price, t.prod_min_price=s.prod_min_price
(5) WHEN NOT MATCHED THEN INSERT (prod_id, prod_name, prod_desc,
prod_subcategory,
(6) prod_subcategory_desc, prod_category, prod_category_desc, prod_status,
(7) prod_list_price, prod_min_price)
(8) VALUES (s.prod_id, s.prod_name, s.prod_desc, s.prod_subcategory,
(9) s.prod_subcategory_desc, s.prod_category, s.prod_category_desc,
(10) s.prod_status, s.prod_list_price, s.prod_min_price);
```

Figure 4: MERGE Statement provided by Oracle Data Warehousing Guide [Dat03]

In the above statement `products t` refers to the target table and `products_delta` refers either to the source or to the delta table. In (2) the merge criteria are stated. If the merge criteria are fulfilled then the associated tuples within the source / delta tables are updated (3-4). Otherwise new tuples are inserted (5-10) (where column name of source and target table do not necessarily have to match). The *merge*-statement offers the possibility of expressing an "if-then-else"-like semantics (cf. Figure 4.1). Therefore it could be used in a slightly modified way to implement the "if-then-else"-constructs of the DELTALAYER algorithm. Using a table representing the changes of an unpurged log as input, the modified *merge*-statements (one for each change operation) produce a correct net delta in the output table.

Another, maybe even simpler possibility is to implement the DELTALAYER algorithm within JAVA and make it accessible using the mechanism of user-defined functions of Oracle database.

*IBM DataPropagator:* Another commercial system dealing with delta information in the context of replicating distributed data sources is IBM DataPropagator [IBM95]. Using IBM DataPropagator, basically, it is possible to produce so called condensed change data tables which are supposed to contain net update information. However, condensed change data tables are neither minimal nor correct in most cases. Let us apply the triggers presented in [IBM95] on the change combinations summarized in Table 1. If, for example, a change of type DELETE (T) is applied, the before trigger, first of all, deletes any entry for which key equals T.Key holds (regardless whether it is an update or insert operation) and the after trigger stores the DELETE (T) operation afterwards. Doing so, in case of unnec-

essary `INSERT (T) /DELETE (T)` pairs the `DELETE (T)` operation remains within the condensed change data table (therefore the table is not minimal). Furthermore, this result is even not correct if `T.Key` is not key of the data warehouse schema afterwards (otherwise the database system would simply reject the application of the `DELETE (T)` operation). In case of unnecessary `INSERT (T) /UPDATE (T)` pairs the triggers always produce an incorrect output since tuple `T` having key `T.Key` is no longer present when the `UPDATE (T)` operation is applied. Consequently, the triggers provided for IBM DataPropagator so far could be improved by implementing them according to the `DELTALAYER` algorithm. Due to lack of space we abstain from details here.

## 4.2 Post Processing of Differential Snapshots

Another application for the `DELTALAYER` algorithm is post processing the output of differential snapshot algorithms. As already mentioned in [LGM96] this would constitute a valuable extension in order to minimize update windows within the data warehouse [DDJ<sup>+</sup>98]. One alternative in this context is to take the whole differential snapshot as input for the `DELTALAYER` algorithm (real post processing). However, we can also think of directly integrating the `DELTALAYER` algorithm into the differential snapshot algorithms. For the Window algorithm [LGM96], for example, this can be accomplished rather easily: instead of writing entries from *AgingBuffer1* into the `DELETE` queue and entries from *AgingBuffer2* into the `INSERT` queue all entries from the aging buffers can be directly written into the *DeltaLayer*. There an immediate purge of wasteful `INSERT/DELETE` pairs takes place.

## 4.3 An Extension Towards Arbitrary Change Operations

`INSERT/DELETE/UPDATE` operations are rather simple changes. In applications such as data warehouses [GM95] or process management systems more advanced change operations become necessary in practical applications [RRJK06]. In our ADEPT process management system, for example, we offer the change framework depicted in Figure 5a. Here, first of all, we distinguish between primitive and high-level change operations in order to offer better user support. Thereby the high-level change operations are constructed by combining an arbitrary amount of change primitives. Furthermore, high-level change operations are equipped with a set of (formal) pre- and post-conditions which guarantee the correctness of the resulting process when applying a high-level change operation.

The main differences between high-level operations and change primitives are their properties and the intention they are used for. High-level operations act as an interface provided to the end-user. Furthermore – contrary to the set of change primitives – the set of high-level change operations does not necessarily have to be closed (i.e. the process management system can be extended by new high-level change operations any point in time). Change primitives are mainly used to create the data structures (in particular the

*DeltaLayer* based on which (high-level) changes are represented at system-level. More precisely, there is a mapping for each high-level change operation to associated change primitives (including those high-level operations which newly defined within the process management system). Based on the concept of mapping high-level change operations at system level to a change primitive representation, the associated change primitives can be used for manipulating the internal representation, i.e. the *DeltaLayer*. Therefore, for high-level change operations the DELTALAYER algorithm can again be used in order to keep the *DeltaLayer* correct and minimal. Minimality of the change information, in turn, is crucial for correctness checks in the context of concurrently applied changes which are mainly based on comparing the associated change logs.

In Figure 5b, for example, a process template change is illustrated which is accomplished by applying two high-level change operations  $sInsert(\dots)$  and  $sMove(\dots)$  at the user level. At the system level, the high-level operation have been automatically transformed into a change primitive representation. Note that the DELTALAYER algorithm presented in this paper can be easily adapted to the change primitives used in adaptive process management systems (e.g.,  $addNode(\dots)$ ,  $addEdge(\dots)$ , or  $addDataElement(\dots)$ ) as it has been shown in [RRJK06]. Based on this primitive representation, the DELTALAYER algorithm produces the purged *DeltaLayer* which automatically keeps a minimal change information for correctness checks at any time.

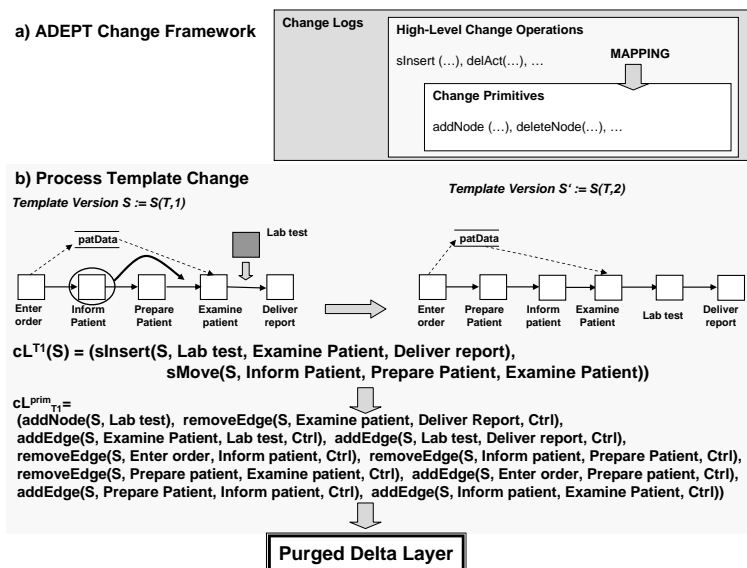


Figure 5: a) ADEPT Change Framework and Example Process Template Change

## 5 Related Work

There are several suggestions how to extract and feed updates from data sources into a data warehouse. One complete architecture has been proposed within the WHIPS project [HGMW<sup>+</sup>95] ranging from data extraction to the update of materialized views.

*Snapshot-Based Approaches:* If data sources do not offer any support for extracting data updates (typical for legacy systems) the only way to do so is to build snapshots of the files before and after the change. Then the difference between those snapshots is computed, e.g., by the algorithm proposed in [LGM96]. Commercial systems such as Oracle database offer the possibility to create different snapshots [BG04]. The resulting change log is loaded into the data warehouse afterwards. However, such algorithms produce wasteful change combinations as well, i.e., unnecessary INSERT (T) /DELETE (T) or DELETE (T) /INSERT (T) pairs as depicted in Table 1. This may cause overhead when loading the files afterwards [LGM96] as well as unnecessary checks during the transformation phase within the staging area later on. Therefore a post processing as proposed in this paper would be beneficiary for working with differential snapshots.

*Replication-Based Approaches:* Replication approaches range from copying database tables to storing the updates tuples within so called delta tables. In particular the latter is related to our approach.

*Materialized Views:* The principles of net changes and delta tables may be also interesting for the maintenance of views in data warehouses [LYG99, LYC<sup>+</sup>00, ZGHW95, GL95], i.e., the problem of propagating changes of a base table  $B$  to a view  $V$  derived from  $B$ . Here the incremental update of views is of particular interest, i.e., updated view  $V'$  is calculated as original view  $V$  minus the set of deleted tuples  $\nabla V$  plus the set of inserted tuples  $\Delta V$ . The challenge is to determine sets  $\nabla V$  and  $\Delta V$ . This comprises two aspects: first of all, it has to be determined which tuples have been changed in  $B$  and secondly, how to calculate the sets  $\nabla V$  and  $\Delta V$  from that (e.g., the necessary aggregations). Our approach can be taken to optimize the result of step one, the determination of changes applied to  $B$ : Since UPDATES on  $B$  are expressed as INSERT and DELETE pairs [LYG99] the minimality requirement for the view updates only refer to INSERT and DELETE operations [GL95]. However, our algorithm additionally considers minimality regarding UPDATE operations.

*Object-Oriented Approaches:* Delta objects have been proposed for object-oriented databases [SBDU97]. Here the changes are associated with the objects, not stored within a global delta log. Furthermore, the purpose of managing the delta information is different. In this context, delta information is, for example, used for debugging and run-time testing.

*Adaptive Process Management:* In general, adaptivity in process management systems has been a hot topic in literature for many years. However, there are only few approaches dealing with an efficient implementation of advanced process management functionality [Wes98, KAS<sup>+</sup>03]. So far, they have neglected issues related to change log management. Our ADEPT system is one of the very few available research prototypes for adaptive, high-performance process management [RRD04b, RRKD05, RRJK06].

## 6 Summary and Outlook

In this paper we have introduced the DELTALAYER algorithm which receives a change log as input and produces a structured net change within the *DeltaLayer* format as output. We have shown that the algorithm produces a correct (i.e., minimal) net change with respect to the set of inserted, deleted, and updated tuples. Furthermore in the paper different application areas for the DELTALAYER algorithm have been provided such as replication mechanisms in commercial systems (e.g., Oracle or IBM DataPropagator), post processing of differential snapshots to feed data updates into data warehouses, or correctness checks for adaptive process management systems.

At the moment we are integrating the DELTALAYER algorithm within the implementation of our process management engine ADEPT2 (for more information see [www.aristaflow.de](http://www.aristaflow.de)). This is also particularly interesting since maintaining a *DeltaLayer* provides transparency to any change operations which is added to the change framework afterwards. Based on the implementation we will conduct performance studies within different scenarios.

In future work we want to study more application areas for the DELTALAYER algorithm. One example is the cleaning of data within the staging area of a data warehouse (cf. Figure 6). The idea is to purge data (if desired!) after necessary transformation using metadata within a *DeltaLayer*. Doing so, for example, redundancies can be purged.

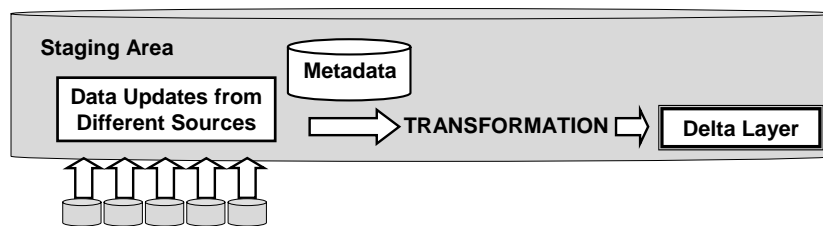


Figure 6: Cleansing Data from Different Sources withing Staging Area

## References

- [AV88] S. Abiteboul and V. Vianu. Equivalence and Optimization of Relational Transactions. *Journal of the Association of Computing Machinery*, 35(1):70–120, 1988.
- [BG04] A. Bauer and H. Günzel. *Data Warehouse Systems*. dpunkt, 2004.
- [Dat03] Oracle Database. *Data Warehousing Guide, 10g Release 1 (10.1)*, December 2003.
- [DDJ<sup>+</sup>98] L. Do, P. Drew, W. Jin, V. Jumanı, and D. Van Rossum. Issues in Developing Very Large Data Warehouses. In *Proc. Int'l Conf. VLDB*, pages 633–636, 1998.
- [GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proc. Int'l SIGMOD Conf.*, pages 328–339, 1995.

- [GM95] A. Gupta and I. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.
- [HGMW<sup>+</sup>95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin*, June, 1995.
- [IBM95] IBM. Data Where You Need It, The DPropR Way! DataPropagator Relational Solutions Guide. Technical Report GG24-4492-00, IBM International Technical Support Organization, San Jose Center, 1995.
- [JNS<sup>+</sup>97] H. Jagadish, P. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proc. Int'l Conf. VLDB*, pages 16–25, 1997.
- [KAS<sup>+</sup>03] K. Kochut, J. Arnold, A. Sheth, J. Miller, E. Kraemer, B. Arpinar, and J. Cardoso. IntelliGEN: A Distributed Workflow System for Discovering Protein-Protein Interactions. *DPD*, 13(1):43–72, 2003.
- [LGM96] W. Labio and H. Garcia-Molina. Efficient Snapshot Differential Algorithms for Data Warehousing. In *Proc. Int'l Conf. VLDB*, pages 63–74, 1996.
- [LYC<sup>+</sup>00] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance Issues in Incremental Warehouse Maintenance. In *Proc. Int'l Conf. on Very Large Databases*, pages 461–472, 2000.
- [LYG99] W. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the Warehouse Update Window. In *Proc. Int'l SIGMOD Conference*, pages 383–394, 1999.
- [RRD04a] S. Rinderle, M. Reichert, and P. Dadam. Disjoint And Overlapping Process Changes: Challenges, Solutions, Applications. In *CoopIS'04*, pages 101–120, 2004.
- [RRD04b] S. Rinderle, M. Reichert, and P. Dadam. Flexible Support Of Team Processes By Adaptive Workflow Systems. *DPD*, 16(1):91–116, 2004.
- [RRJK06] S. Rinderle, M. Reichert, M. Jurisch, and U. Kreher. On Representing, Purging, and Utilizing Change Logs in Process Management Systems. In *Proc. Int'l Conf. BPM*, pages 241–256, 2006.
- [RRKD05] M. Reichert, S. Rinderle, U. Kreher, and P. Dadam. Adaptive process management with ADEPT2. In *ICDE'05*, pages 1113–1114, 2005.
- [SBDU97] A. Sundermier, T. Ben Abdellatif, S.W. Dietrich, and S.D. Urban. Object Deltas in an Active Database Development Environment. In *Proc. Int'l Conf. DOOD-97*, pages 211–229, 1997.
- [Wes98] M. Weske. Object-Oriented Design of a Flexible Workflow Management System. In *ADBS98*, pages 119–131, 1998.
- [Wes01] M. Weske. Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In *HICSS-34*, 2001.
- [ZGHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proc. Int'l SIGMOD Conf.*, pages 316–327, 1995.