# Online, Non-blocking Relational Schema Changes

Jørgen Løland and Svein-Olaf Hvasshovd

Dept. of Computer Science, NTNU, Trondheim, Norway
{jorgen.loland, svein-olaf.hvasshovd}@idi.ntnu.no

**Abstract.** A database schema should be able to evolve to reflect changes to the universe it represents. In existing systems, user transactions get blocked during complex schema transformations. Blocking user transactions is not an option in systems with very high availability requirements, like operational telecom databases. A non-blocking transformation framework is therefore needed.

A method for performing non-blocking full outer join and split transformations, suitable for highly available databases, is presented in this paper. Only the log is used for change propagation, and this makes the method easy to integrate into existing DBMSs. Because the involved tables are not locked, the transformation may run as a low priority background process. As a result, the transformation has little impact on concurrent user transactions.

## 1 Introduction

Database schemas are typically designed to model the world as understood at design time. At this point in time, the schema design may be excellent for the intended usage. Many applications change over time, however. In a study of seven applications, Marche [18] reports of significant changes to relational database schemas over time. Only one of the studied schemas had less than 50% of their attributes changed. Furthermore, 16% of all changes were due to changes in the degree of normalization. The evolution of the schemas continued after the development period had ended. A similar study of a health management system [25] came to the same conclusion. This indicates the need for non-trivial schema transformations.

A schema transformation can easily be made if the involved tables can be locked while the transformation is performed. Most databases can do this by issuing an *insert into select* command, where the select statement can be any valid SQL select statement, e.g. join or union.

Databases with very high availability requirements should not be unavailable for long periods of time. For tables with large amounts of data, the *insert into select* method could easily take tens of minutes or more. Such databases, often found in e.g. the telecom industry, would clearly benefit from a mechanism to change the schema without being blocked.

In this paper we suggest schema transformation methods for the full outer join (FOJ) and split relational operators. The methods are non-blocking and are based on log redo. FOJ and split are considered important operators by the authors because they are used to change the normalization degree of the schema.

We assume that both redo and undo log records are produced, and that undo operations produce Compensating Log Records (CLR) [6] as described in the ARIES method [20]. It is also assumed that a log sequence number (LSN) is associated with each record [12].

The paper is organized as follows: Section 2 describes other methods and research areas related to non-blocking transformations. An overview of our transformation framework is presented in 3. Details for how to apply the framework to FOJ and split transformations are presented in Sections 4 and 5, respectively. The framework has been implemented in a prototype and test results from this prototype are discussed in Section 6. Finally, in Section 7, we conclude and suggest further work.

## 2 Related Work

Little research has been published on non-blocking schema transformations in relational databases. Our method does, however, use techniques from both fuzzy copy and materialized views (MVs), as described in the following sections.

### 2.1 Ronströms' Method

Ronström [23] presents a framework that uses both a reorganizer and triggers within user transactions to perform schema transformations. Sagas [7] are used to organize the transformation. New tables, constraints, indices, and triggers are first added to the schema. The reorganizer then scans the old tables, while triggers make sure that updates to the old tables are executed immediately to the transformed table. When the scan is complete, the old and transformed tables are consistent due to the triggered updates.

No implementation or test results have been published on Ronströms method. Triggers are, however, used in a similar way to keep immediate Materialized Views (MVs) up to date. The extra workload incurred with using triggers to update MVs is significant, and deferred MVs are therefore recommended whenever possible (see e.g. [5, 16]).

With our method, there is no need for the transformed table to be consistent with the old table before the very end of the transformation. Updates can therefore be propagated to the transformed tables during low workloads.We also expect our method to be much more efficient in a distributed DBMS where user transactions have to wait for triggers to access other nodes. Finally, our method does not require the use of Sagas.

### 2.2 Fuzzy Copy

Our transformation framework has to make a copy of the source tables without setting locks to satisfy the non-blocking requirement. To do this, we use a modified fuzzy copy technique.

Hvasshovd et al. [4, 13] presents fuzzy copy as a way to copy a table to another node in a cluster without blocking. A *begin-fuzzy mark* is first written to the log. The records in the source table are then read without setting locks, resulting in a *fuzzy copy* where some of the updates that were made during the scan may not be reflected. The log is then redone to the copy in a similar way as ARIES [20] to make it up to date. LSNs on records ensure that the log propagation is idempotent. When all log records have been redone to the copy in ascending order, it is in the same state as the source table. An *end-fuzzy mark* is then written to the log, and the copy process is complete. The method requires CLR to be used for undo processing.

### 2.3 Materialized Views

Materialized views (MVs) store the result of a query. They are used to speed up query processing and must therefore be consistent with the source tables. Methods to propagate changes from the source tables to an MV is an area of extensive research (e.g. [3, 5, 8, 9, 10, 11, 16, 21, 24, 27]). All these propagation methods require the MVs to be consistent with a previous state of the source tables. This incurs that an MV must initially be consistent, i.e. populated with the result of a blocking read.

At first glance, MVs have much in common with our schema transformation framework. Blocking read operations are, however, not allowed in the transformation framework, so fuzzy copies of the source tables are used to create the initial images of transformed tables. Since a fuzzy copy is not consistent with the source table, the MV update methods are not applicable. Further more, schema transformations only require the transformed tables to *converge* to the source tables (i.e. to be consistent when all operations are propagated [27]), whereas MVs require consistency for all intermediate states as well.

### 2.4 Existing Transformations

Existing database systems, including IBM DB2 v8 [14, 15], Microsoft SQL Server 2000 [19], MySQL 4.0 [26] and Oracle 9*i* [1], offer some simple transformation functionality. These include removal of and adding one or more attributes to a table, renaming attributes and the like. Removal of an attribute can be performed by changing the table description only, thus leaving the physical records unchanged for an unspecified period of time. Complex tranformations like join are not supported.

## 3 General Framework

The goal of the transformation framework is to provide methods that transform the schema without blocking other transactions. The transformations are based on relational operators for two reasons: the effect of the transformation is easy to understand for the database administrator (DBA) that initiates it, and it enables us to make use of existing, optimized code (like join algorithms) for parts of the process.

The framework operates in four steps that are common to both the FOJ and split transformations. These steps are briefly explained below.

### 3.1 Preparation Step

Before the transformation starts, the new tables that are to be used after the transformation have to be created. They may include any subset of attributes from the source tables, but must include at least one candidate key from each. The reason for this is that the transformation method needs a way to uniquely identify which records are affected by an operation on a source table record. In the case that the included candidate key attributes are not wanted in the transformed tables, they must be deleted *after* the FOJ or split transformation completes.

Constraints, both new and from the source tables, may be added to the new tables. This should, however, be done with great care since constraint violations may force the transformation to abort.

Any indices that are needed on the new tables should also be created before the transformation starts. These indices will be up to date when the transformation is complete.

## 3.2 Initial Population Step

The newly created transformed tables have to be populated with records from the source tables. The first step of populating the new table is to write a *fuzzy mark* in the log. This log record must include the transaction identifiers of all transactions that are active on the source tables, i.e. a subset of the active transaction table. The source tables are then read fuzzily, returning an inconsistent result since locks are ignored [13]. Once the source tables have been read, the transformation operator is applied and the result, called the *initial image*, is inserted into the transformed tables.

## 3.3 Log Propagation

When the initial image(s) have been inserted into the transformed table(s), another fuzzy mark is written to the log. This log record marks the end of the current log propagation cycle and the beginning of the next one.

Log records of operations that may not be reflected in the transformed tables are now inspected. In the first iteration, the oldest log record that may contain such an operation is the oldest log record of any transaction that was active when the first fuzzy mark was written. Later log propagation iterations only have to read the log after the previous fuzzy mark.

Propagation rules for update, insert and delete of records in a source table differ for each transformation type, and are explained in detail in Sections 4 and 5.

To speed up the synchronization step, locks are maintained on records in the transformed tables during the entire transformation. The locks are likely to conflict during the transformation. Since they are only needed when user transactions access both source and transformed tables, i.e. during synchronization, they are ignored for now.

The synchronization step should not be started if a significant portion of the log remains to be propagated because it involves latching of tables. Each log propagation iteration therefore ends with an analysis of the remaining work. Based on the analysis, either another log propagation iteration or the synchronization step is started. The analysis could be based on, e.g. the time used to complete the current iteration, a count of the remaining log records to be propagated, or an estimated remaining propagation time.

If more log records are produced than the propagator is able to process, the synchronization is never started. If this is the case, the transformation should either be aborted or get higher priority.

The transformed tables of both FOJ and split of consistent data are self-maintainable [22], i.e does not need more information than the log and the transformed tables themselves. This makes them highly suitable for distributed databases as well.

### 3.4 Synchronization

When synchronization is initiated, the state of the transformed tables should be very close to the state of the source tables. This is because the source tables have to be latched during one final log propagation iteration that makes the transformed table consistent with the source tables.

We suggest three ways to synchronize the transformed tables to the source tables and thereby complete the transformation process. These are called blocking commit, non-blocking abort and non-blocking commit synchronization.

*Blocking commit* synchronization starts by blocking all new transactions that try to access any of the tables involved in the transformation. Transactions that already have locks on the source tables are then allowed to complete before a final log propagation iteration is performed. The transformed tables are now consistent with the source tables. New transactions are then given access to the new tables only. This method does not follow the non-blocking requirement.

*The non-blocking abort* strategy begins by placing table latches on the source tables for the duration of one final log propagation. Latching these tables effectively pauses ongoing transactions that work on them, but since there are only a few log records to propagate, the pause should be very brief (less than 1 ms in our current implementation). Once the log propagation is complete, the transformed tables are in the same state as the source tables. Recall from Section 3.3 that locks have been maintained on the transformed tables since the first fuzzy log mark. Records that are locked in the source tables are therefore also locked in the transformed tables. New transactions are now allowed to access the unlocked parts of the transformed table while transactions that were active on the source tables are forced to abort. The log propagation continues as a background process as long as old transactions are alive. Source table locks held in the transformed tables are released as soon as the propagator has processed the abort log record of the lock owner transaction.

*Non-blocking commit* synchronization works much like the previous strategy in that latches are placed on the source tables during one final log propagation. But in contrast to the previous strategy, transactions on the source tables are allowed to continue processing once the tables have been synchronized. This is called a soft transformation in [23]. The drawback of this method is that as long as any of the old transactions are alive, all locks on source tables have to be acquired on the corresponding records in the transformed tables. However, nonconflicting transactions are not aborted due to the transformation.

Finally, the source tables are dropped from the schema, and the transformation is complete.

## 4 Full Outer Join Transformations

The method for FOJ transforms two source tables, $R$ and $S$, into one table $T$ by applying the FOJ operator. An example transformation is shown in Figure 1. The general

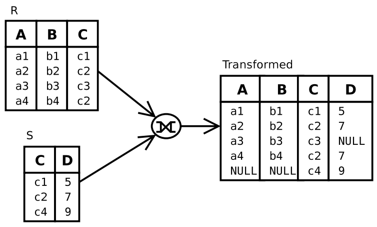


**Fig. 1.** Example full outer join transformation

transformation steps explained in Section 3 are discussed for FOJ below. For readability it is assumed that the join attribute of table $S$ (attribute $c$ in Figure 1) is unique, i.e. there is a one-to-many relation between the source tables. A solution for the many-to-many case is sketched in Section 4.2.

## 4.1 Preparation and Initial Population Steps

In the *preparation* step, the transformed table $T$ is created. As a minimum, $T$ must contain an identifying attribute set from both tables in addition to the join attributes. Constraints may be added, but unique constraints on attributes in $S$ should be avoided since a record in $S$ is likely to occur multiple times in $T$.

Without lack of generality, we assume that the key attributes of $R$ are also the key attributes of $T$. As long as there is a unique way to identify the $T$ records to update, the method will work without this assumption.

To improve transformation performance, an index should be created on the join attributes of $T$. If the join attributes of $S$ are not the same attributes as the primary key, an index should also be created on the primary key attributes of $S$ in the transformed table. These indexes provide fast lookup on all $T-$records that are affected by an operation on an $S-$record.

During the *initial population* step, the source tables are first read fuzzily. The FOJ of the results are then inserted into $T$. Special $R-$ and $S-$ NULL records, denoted $r^{null}$ and $s^{null}$, are joined with records that otherwise would not have a join match, as illustrated in Figure 1.

## 4.2 Log Propagation

The fuzzy copy method of Hvasshovd et al. [13] use a record state identifier, typically the Log Sequence Number (LSN), to make logged operations idempotent. Logged operations are applied to records only if the LSN of the log record is greater than the LSN of the record.

In our framework, there are no valid state identifiers for the records in the newly created $T$. This is because records in $T$ consist of two records, one from each source table. The records from the source tables have an LSN each, while the resulting record may only have one LSN. The LSN of a record in $T$ is therefore not a correct state identifier.

The rest of this section describes how to apply the log to the initial image, i.e. the join of the fuzzy read source tables, without using state identifiers. It works under the assumption that all write operations on the source tables use exclusive locks; i.e. delta updates [17] are not allowed.

When fuzzy read starts, the two source tables are in state 0, denoted $R_0$ and $S_0$. After the initial image has been inserted, $T$ is in an inconsistent state $i$, denoted $T_i$, where all records are in the same or newer state than they had in $R_0$ and $S_0$. All operations on the source tables that happened after state 0 are now applied sequentially to $T$. At some future point in time, during synchronization, all operations in the log have been redone to $T$, making $T$ up to date with $R$ and $S$. The states of the tables at that point in time is called $c$, denoted $R_c$, $S_c$ and $T_c$. $c$ is an action consistent state since both $R$ and $S$ are latched for the final synchronization.

At any point in time during log propagation, $R$ and $S$ have the same or newer state than $T$ for all records. This is a consequence of the fact that all operations reflected in $T$ are simply redoes of operations on $R$ and $S$. In addition, the current state $t$ (denoted $R_t$ and $S_t$) of the source tables precedes the state $c$ for all records. Thus, $0 \leq i \leq t \leq c$, where $a \leq b$ means that $b$ contains at least all operations reflected in $a$, but may also reflect newer operations.

**A Basic Property.** Without valid state identifiers, the log propagator does not know if a log record is already reflected in $T$. The rules are idempotent, i.e., a log record may be redone multiple times. The rules can not handle lost updates, but as shown in the following theorem lost updates never appear:

**Theorem 1.** *(***Records in $T_i$ are up to date***)*
*Assume that the log propagator is currently processing a log record describing an operation to a source table record. The appropriate records in the transformed tables are then either in the same state as the source table record was in when the operation was originally executed, or in a newer state.*

Assuming that the concurrency controller enforces serializability, the record must have been up to date when the operation was originally executed in the source table [2]. The original sequence of operations on that record is in the same order in the log because the log is sequential and the operations are serializable.

A fuzzy read of a table catches all updates that happened before the read started. As a consequence of the fact that fuzzy read ignores locks, it may also include some updates that happened during the read.

Since all updates that happened before the fuzzy read started are guaranteed to be reflected in the initial image, a lost update must have been introduced after that point in the log. The log propagator starts with the first log record of any active transaction at the time of the first fuzzy mark. This is the first operation that may not be reflected in the initial image of the transformed table.

Assume that the log propagation rules are correct, i.e. all records in the transformed table that should be affected by a logged operation on a source table record, are updated correctly by the propagator. Then, since no lost updates existed in the initial image and because the log is propagated sequentially, no lost updates can exist after the first log record has been applied. By induction, the transformed table has no lost updates when the current log record is encountered.

In other words: as long as the log is applied in sequential order to the initial fuzzy copy, all records in $T_i$ are in the same or a newer state than the source table records were in when the operation was originally executed.

**Insert Operations.** The log propagator may encounter log records describing insert, update and delete operations on records in the base tables. In what follows, rules for how to propagate insert log records are described.

The notation $r_x^y$ means a record from table $R$ where $y$ is the primary key value and $x$ is the join attribute value. By $t_x^y$, we refer to the record in $T$ resulting from the join of $r_x^y$ and $s_x^x$ (abbreviated $s^x$). As previously assumed, the join attribute of $S$ is unique. Records with no join match in the opposite source table are joined with the $R-$ or $S-$ null record ($r^{null}$ and $s^{null}$), as described in Section 4.1. $\mathbf{A}$ and $\mathbf{B}$ are the sets of all primary key and join attribute values allowed, respectively.

**Rule 1 (Insert $r_x^y$ into $R$)**
*Check if a record with the key $y$, $t_x^y$, exists in $T_i$. If so, ignore the log record. Otherwise, use the join attribute index of $T$ to find a record with the join attribute value $x$. There are three possible results: If $t_x^{null}$ is found, it is updated with the attribute values of $r_x^y$ to form $t_x^y$. If $t_x^v$ is found ($v \in \mathbf{A}, v \neq y$), a new $t_x^y-$record is inserted after joining $r_x^y$ with the $s^x-$part of $t_x^v$. If no record with this join attribute exists in $T_i$, $t_{null}^y$ is inserted after joining $r_x^y$ with $s^{null}$ .*

Theorem 1 states that all records in $T_i$ are up to date with or in a newer state than the log record. For this reason, if $t_x^y$ is found, the log record is already reflected in $T_i$ and can safely be ignored. If this was not the case, two records with the same key $y$ existed in $R$ at the same time.

The other cases are straightforward; by searching the index, the log propagator finds all information necessary to insert $t_x^y$.

Even if $t_w^y$ ($w \in \mathbf{B}$) is not found in $T_i$, it is possible that $T_i$ has a newer state for $t_w^y$ than that of the log. This can only be the case if $t_w^y$ is later deleted, leaving no trace of its existence. If so, the insertion of $t_x^y$ will be corrected when the log record of the delete is encountered later.

**Rule 2 (Insert $s^x$ into $S$)**
*Use the $S-$key index to find all records with the join attribute value $x$ in $T_i$. If any of these records are joined with $s^{null}$, they are updated with the new $s^x$ values. $T$-records joined with an $S-$record other than $s^{null}$ are not updated. Otherwise, if no records have $x$ as the join attribute, $t_x^{null}$ is inserted after joining $r^{null}$ with $s^x$.*

$s^x-$records found in $T_i$ are not modified since Theorem 1 guarantees that they are up to date. For both insert rules, FOJ requires that records with no join match are still present in the result.

**Delete Operations**

**Rule 3 (Delete $r^y$ from $R$)**
*Check if $t^y$ exists in $T_i$, and ignore the log record if not. If $t_{null}^y$ is found, it is simply*

*deleted. If $t_x^y$ is found, the index is used to see if $t_x^y$ is the only record in $T_i$ containing $s^x$. If so, $t_x^{null}$ is inserted after joining $s^x$ with $t^{null}$. $t_x^y$ is then deleted.*

**Rule 4 (Delete $s^x$ from $S$)**
*Use the join attribute index to identify all records with $x$ as join attribute value. If $t_x^{null}$ is found, it is deleted. All other records $t_x^v$ ($v \in \mathbf{A}$) that are found are joined with $s^{null}$.*

These rules are simply delete operations that guarantee the continued existence of their joined counterparts.

**Update Operations.** Insert and delete log records contain all the information needed to propagate the log. For insert log records, this information includes all attribute values. For delete log records, the primary key of the record to delete is all the information needed.

Update log records are less informative since they typically contain the primary key and updated attribute values only. The information not found in the log record is, however, available in $T_i$ as described next.

**Rule 5 (Update join attribute of $r_x^y$ to $z$)**
*The record with key attribute value $y$, $t_w^y$ ($w \in \mathbf{B}$), is first read from $T_i$. If $t_w^y$ is not found in $T_i$, or if $w \neq x$, the log record is ignored. Assuming that $t_x^y$ is found, the join attribute index of $T_i$ is searched to find if $s^x$ is represented in at least one more record. If not, $t_x^{null}$ is inserted by joining $r^{null}$ with $s^x$.*

*Next, the join attribute index is searched for a record with $z$ as the join attribute. If $t_z^{null}$ is found, it is updated with the attribute values of $r_z^y$ to form $t_z^y$. If $t_z^v$ ($v \in \mathbf{A}, v \neq y$) is found, a new $t_z^y-$record is inserted after joining $r_z^y$ with the $s^z-$ part of $t_z^v$. If no record with this join attribute exists in $T_i$, $t_{null}^y$ is inserted after joining $r_z^y$ with $s^{null}$.*

Again, Theorem 1 guarantees that the record $t_w^y$ found in $T_i$ is at least up to date with the log. If $w \neq x$, an operation representing a newer state than that of the log record is already reflected in $t_w^y$. Applying the logged update would not lead to inconsistency in the future state $T_c$ since the log record leading to that newer state will be found in the log before $c$ is reached. Doing so does, however, incur extra work.

Even though the join attribute is guaranteed to be unique in $S$, it is not necessarily the primary key. It may therefore be updated:

**Rule 6 (Update join attribute of $s^x$ to $z$)**
*All records in $T_i$ that have $x$ as the join attribute value, are first identified. If no record is found, the log record is ignored. If $t_x^{null}$ is found, it is deleted. If found, all records $t_x^v$ ($v \in \mathbf{A}$) in $T_i$ are joined with $s^{null}$ to form $t_{null}^v$.*

*Next, all records in $T_i$ that have the new join attribute value, $z$, are identified. If $t_{null}^v$ is found, it is updated with $s^z$ to form $t_z^v$. Any $t_z^v$ record already joined with an $s^z$ record stays unmodified. If no other $T-$record is joined with $s^z$, $r^{null}$ is joined with $s^z$ to form $t_z^{null}$.*

This rule operates like delete of $s^x$ followed by insert of $s^z$. Like in propagation rule 5, $s^x$ is used to extract the attribute values of $s^z$ since the log does not include this information.

**Rule 7 (Update other attribute of $r^y$ or $s^x$)**
*If an update of $r^y$ is described, the record $t^y$ with the same primary key is updated with the new attribute values. Similarly, if $s^x$ is updated, the index of $T_i$ is used to identify all records $t_x^v$ ($v \in \mathbf{A}$) with $x$ as the join attribute value. All records found are updated as described in the log. If no records are found, the log record is ignored.*

The rule should be intuitive: all records in $T_i$ that partly consist of the updated record must be updated with the new values. If no records match the key, the log record can safely be ignored since Theorem 1 guarantees that $T_i$ has a newer state for that record when this happens.

**Sketch of Log Propagation for Many-to-Many Relationships.** The described log propagation rules work under the assumption that the join attribute of $S$ is unique. In this section, we sketch what needs to be done when this assumption does not hold.

In many-to-many relationships, each $R-$record can be joined with multiple $S-$records. Because of this, the primary key of $R$ cannot be used as the primary key in $T$ alone. Instead, one or more identifying attributes from both source tables, e.g. their primary keys, should be used together to form the primary key of $T$. In what follows, $t_z^{yx}$ means a record in $T_i$ that consists of a record $r_z^y$ joined on attribute value $z$ with $s_z^x$.

The one-to-many rules for operations on $S-$records does not need modification to work in many-to-many transformations. Operations on $R-$records, however, need to be modified so that all records in $T_i$ that consist of the described $R-$part are affected. An index should be created to speed up the search for these.

For update and deletion of a record $r_z^y$, the modified rules simply has to identify all $T-$records consisting of $r_z^y$ and apply the operation described for the one-to-many case. For every deletion of a $T-$record, the existence of other $S-$records with the same primary key has to be checked to ensure full outer join.

When a log record describes an insert of $r_z^y$, a $t_z^{yv}-$record ($v \in \mathbf{A}$) has to be inserted for every matching record $s_x^v$. When the join attribute of an $r_z^y$ is updated, all existing $T-$records that the $r_z^y$ contributed to must be deleted. The continued existence of the deleted records‘ $S-$counterparts must be enshured as well. New join-matches are then inserted into $T$.

## 4.3 Synchronization

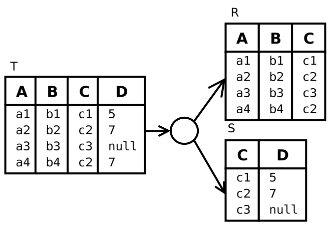
Synchronization of FOJ transformations are performed as described in Section 3.4. Lock propagation between the old and new tables must, however, be described in more detail for the non-blocking strategies.

Since locks from two source tables $R$ and $S$ are transferred to one new table $T$, the source table locks may conflict in $T$. This is, however, only a consequence of the lock granularity being *record* as opposed to *attribute*. Clearly, operations on $R$ and $S$ do not modify the same attributes. New lock compatibility rules for $T$ are needed to avoid the conflict. Note that this is only needed for the non-blocking strategies.

**Lock Compatibility.** A transaction being aborted cannot aquire new locks, so the *non-blocking abort* strategy only needs lock releases to be transferred from the source tables

|  | R.r | S.r | T.r | R.w | S.w | T.w |
|---|---|---|---|---|---|---|
| R.r | y | y | y | y | y | n |
| S.r | y | y | y | y | y | n |
| T.r | y | y | y | n | n | n |
| R.w | y | y | n | y | y | n |
| S.w | y | y | n | y | y | n |
| T.w | n | n | n | n | n | n |

**Fig. 2.** Lock compatibility matrix for locks in T for the non-blocking strategies.



| A | B | C | D |
|---|---|---|---|
| a1 | b1 | c1 | 5 |
| a2 | b2 | c2 | 7 |
| a3 | b3 | c3 | null |
| a4 | b4 | c2 | 7 |

| A | B | C |
|---|---|---|
| a1 | b1 | c1 |
| a2 | b2 | c2 |
| a3 | b3 | c3 |
| a4 | b4 | c2 |

| C | D |
|---|---|
| c1 | 5 |
| c2 | 7 |
| c3 | null |

**Fig. 3.** Example Split transformation

to $T$. When a *transaction aborted* log record is encountered in the log, the propagator releases the locks of that transaction in $T$.

With *non-blocking commit*, transactions are active on both the source tables and the transformed table at the same time. All transactions may acquire new locks, but to prevent inconsistencies, locks must be transferred both from $T$ to $R$ and $S$ and vice versa. If a transaction cannot get a lock on all implicated records in all tables, it is not allowed to go forward with the operation.

Because locks from two non-conflicting operations in $R$ and $S$ could conflict in $T$, new lock compatibility rules have been developed for locks that are transferred from the source tables to $T$. As can be seen in Figure 2, the new rules ensure that locks from operations executed by transactions on the source tables do not conflict in $T$, whereas they conflict with operations executed by transactions in $T$. The compatibility matrix can easily be extended to multigranularity locking [2].

As for the non-blocking abort case, locks are released when the propagator encounters a *transaction aborted* or *commited* log record.

## 5 Split Transformation

The (vertical) split transformation takes one source table, $T$, and transforms it into two tables $R$ and $S$. This is the reverse of the FOJ transformation described in the previous section, as illustrated by Figure 3. It follows the four steps described in Section 3.

When a table $T$ is split, multiple records may have equal $S-$parts. These records should be represented by only one record in $S$. Further more, a record in $S$ should only be deleted when there are no more records in $T$ with that $S-$part. To be able to decide if this is the case, a *counter*, similar to that of Gupta et al. [10], is associated with each $S$ record. When an $S$ record is first inserted, it has a counter of 1. After that, the counter is increased every time a record with the same primary key is inserted, and decreased every time one is deleted. If the counter of a record reaches zero, the record is removed from $S$.

The notation is the same as in Section 4: each record $t_x^y$ in $T$ is split into two records, $r_x^y$ and $s^x$ where $y \in \mathbf{A}$ and $x \in \mathbf{B}$. $\mathbf{A}$ and $\mathbf{B}$ are the sets of valid primary key values in $R$ and valid values for the attribute used to split, respectively. As for join, the states $R_c$ and $S_c$ is reached at some future point in time when the log propagator has applied the entire log to the transformed tables and the synchronization step is complete.

For readability, we assume that the split attribute is also the primary key in $S$, although this is not required for the method to work. The method does, however, require that the split attribute is a candidate key in $S$, i.e. can be used to identify $S-$records.

## 5.1 Data Consistency

Before the split method is described in detail, we show that inconsistencies that make it impossible to process the transformation may be found in $T$. Consider the following example:

*Example 1.* A company maintains a database of customer contact information, as shown in the table:

| **Customer ID** | Name | Postal Code | City |
|---|---|---|---|
| **001** | Peter | 7050 | Trondheim |
| **002** | Mark | 5020 | Bergen |
| **003** | Gary | 0050 | Oslo |
| ... | ... | ... | ... |
| **134** | Jen | 7050 | Trnodheim |

Customer ID is used as the primary key of this table. There is also a functional dependency in that postal code determines city.

Notice that there is an inconsistency between customers 001 and 134 since the postal codes are the same, whereas the city names differ. Nothing prevents such inconsistencies from occuring in this table, and the schema transformation framework has no means to decide whether "Trondheim" or "Trnodheim" is correct if we were to split this table on postal code. ♦

If inconsistencies like the one in Example 1 exist in $T$, we are not able to perform a split transformation without fixing them.

The log propagation rules are divided into two parts. Section 5.2 describes rules working under the assumption that inconsistencies does not appear in $T$. Section 5.3 describes additional rules needed when such assumptions are not made.

## 5.2 Split of Consistent Data

In this scenario, it is assumed that the DBMS applies measures that guarantee consistency. The method provides an easy-to-understand basis for the scenario where inconsistencies may occur.

During the *preparation and initial population* steps, $S$ and $R$ are simply created and populated as described in Section 3.

An alternative strategy is to create and populate the $S-$table only. Since all attributes needed in $R$ are already present in $T$, $T$ can be renamed to $R$ during synchronization if attributes that are not part of $R$ are removed first. By utilizing this, the transformation would require less space, and updates that would not affect attributes in $S$ could be ignored. Unfortunately, the log propagator needs information on both the LSN and the split attribute value of each $R-$record in the current intermediate state. A temporary table $P$ would be needed to keep track of this information during propagation.

Although $P$ may potentially be much smaller than $R$, this section describes how the method works when $R$ is created as a separate table. Only minor adjustments are needed for the temporary table method to work.

The *synchronization step* works as described in Section 3.4.

## Insert

After the initial images have been inserted into $R$ and $S$, log propagation can start. When a log record for an insert into $T$ is found in the log, $R$ and $S$ are updated using the following rule:

**Rule 8 (Insert $t_x^y$ into $T$)** *The existence of a record with the primary key value $y$, $r^y$, in $R_i$ is first checked. There are two scenarios: if $r^y$ is found, the log record is ignored. If $r^y$ is not found, the $R-$part of $t_x^y$, $r_x^y$, is inserted into $R_i$.*

*Assuming that $r^y$ did not previously exist in $R_i$, the $S-$part of $t_x^y$, $s^x$, is now inserted into $S_i$. First, the existence of a record with the same primary key $x$ is checked. If found, the counter of that record is increased by one. The LSN is then updated if the LSN of the log record is higher than that of $s^x$. If $s^x$ does not exist in $S_i$, $s^x$ is inserted with a counter of one and the LSN of the log record.*

By Theorem 1, if a record with the key $y$ is found in $R_i$, the log record is guaranteed to be reflected in the transformed tables. Both insertion into $R_i$ and $S_i$ are therefore ignored. With guaranteed consistency, the inserted $s^x$ record is either equal to an existing record in $S_i$, or the transaction that generated this log record also updated all other $T-$records contributing to $s^x$ consistently. Changing nothing but the counter and possibly the LSN is therefore correct.

## Delete

**Rule 9 (Delete $t^y$ from $T$)** *If no record with the primary key value $y$, $r^y$, exists in R, or if one exists that has a higher LSN than that of the log record, the log record is ignored.*

*If a record $r_v^y$ ($v \in \mathbf{B}$) exists and has a lower LSN, it is deleted from $R_i$. The counter of $s^v$ is then decreased, and the LSN is changed if the log record has a higher LSN. If the counter reaches zero, the record is completely removed from $S$.*

Using the LSN of the delete operation appears erroneous since it represents the state of a record that does not exist in $T$ anymore. This is not a problem for the transformation framework because the log is propagated sequentially. Changing the LSN of the $S-$record has therefore no consequence on whether future log records will we applied to the table.

We could have chosen not to update the LSN. The same problem would, however, occur in related situations: Assume that the records $t_c^a$ and $t_c^b$ are the only records in $T$ that contribute to the record $s^c$. Also assume that $t_c^a$ is updated and later deleted. Even if the LSN of the $S-$record is not changed by the delete, it still has the LSN value of the update of the record $t_c^a$ that no longer exists.

**Update**

Updating the $R-$part of a record in $T$ is straightforward:

**Rule 10 (Update $t^y$: the $R-$part)** *The existence of a record with the same primary key $y, r^y$, in $R_i$ is first checked. If $r^y$ is not found, or if it has a higher LSN than the logged operation, the log record is ignored.*

*Assuming that $r^y$ is found and that it has an LSN lower than the log record, the record and its LSN is simply updated. The LSN is changed even if no attribute values in $r_x^y$ are updated.*

There are two cases of updates propagated to $S_i$ that must be considered: the split attribute is either updated or not. Note that updates are only applied to $S_i$ if $r^y$ was updated in Rule 10. The reason for this is that the LSN values in $R_i$ uniquely identifies which operations in $T$ are already reflected on existing records in the transformed tables. If a logged operation is reflected in $R_i$, it must also be reflected in $S_i$.

**Rule 11 (Update $t_x^y$: the $S-$part)** *The record $s^x$ with the split attribute value $x$, read from $r_x^y$, is first identified. If the LSN of that record is lower than the log record's, the update is propagated as follows: assuming that only non-split attributes are updated, $s^x$ is simply updated with the new attribute values. Otherwise, if the split attribute is updated, the update is treated as a deletion of $s^x$, followed by the insertion of $s^v$ ($v$ being the new split attribute). Following the argument for insert of $S-$records, only the counter and possibly the LSN of the record with the new key is updated.*

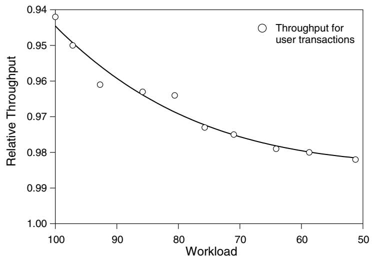
## 5.3 Split of Possibly Inconsistent Data

If consistency is not guaranteed by the DBMS, the transformation framework has to make sure that errors like the one in Example 1 are corrected. Performing this check comes with an overhead to the log propagator. The overhead is, however, not present *within* user transactions since the log propagation, and therefore the overhead, runs as a low priority background process.

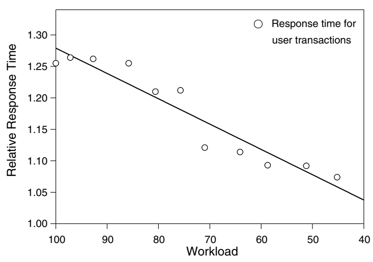
A flag is associated with each record in $S$. Two values are allowed: Consistent ($C$) and Unknown ($U$). A $C$ flag is used when an $S-$record is known to be consistent, and the $U$ flag is used when an $S-$record is known to be inconsistent or has an unknown consistency state.

Every $S-$record that was consistent in the fuzzy read gets a $C-$flag. All other records get a $U-$flag. During log propagation, inserting a record $s^x$ that is not equal to an existing record with the same split value changes a $C-$flag into $U$. The same happens when an update is applied to an $S-$record with a counter greater than 1. A $U-$flag is changed to $C$ only if the operation updates all non-key attributes of a record with a counter of 1.

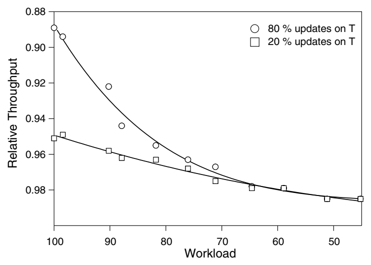
A "concistency checker" (CC) is run regularly. A $U-$flagged record, say $s^v$, is first chosen. The CC then writes a "Begin CC on $v$" record to the log. All records in $T$ contributing to $s^v$ are then read without using locks. If they are consistent in $T$, a "CC: $v$ is ok" record is written to the log together with the correct image of $s^v$. The log propagator keeps track of the records being checked: if $s^v$ is not changed in any way between the two log records, $s^v$ is guaranteed to be consistent and is changed accordingly. Note that all records in $S$ should have a $C-$flag before synchronization is started. Because $T$ has to be read during CC, the split of tables with inconsistent data is not self-maintainable.



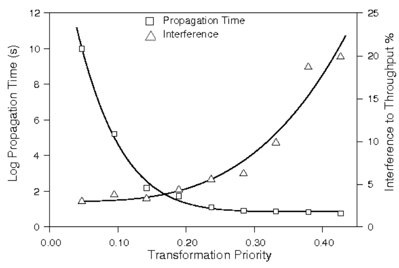
(a) Interference on throughput by initial population with 20% updates on $T$



(b) Interference on resp. time by initial population with 20% updates on $T$



(c) Interference on throughput by log propagation for two update scenarios



(d) Time and interference vs transformation priority at 75% workload

**Fig. 4.** Test results of Split Transformation

## 6 Prototype Implementation

A prototype that performs the described non-blocking transformations has been implemented in Java. It is simplified in that it keeps all data in main memory. This is realistic for databases requiring very fast response time (e.g. [12]), but not for most traditional databases. The costs of the changes are still relevant because we measure *relative* performance, i.e. performance before the change vs. performance during the change.

Four client nodes, one server node and one admin node, all running Linux kernel 2.6, have been used. Each node had 2 AMD Athlon 1600+ CPUs (the prototype has only used one on each node) and 1GB RAM. The nodes were connected with a 100 Mb/s ethernet.

Hundreds of tests have been executed to find the cost of the described schema changes. The cost is measured in reduction in throughput and increased response time of normal transactions run both alone and concurrently with the schema changes.

Each transaction updated 10 records using record locks. 100% workload was defined as the number of concurrent transactions that produced the highest possible throughput. Lower workloads were achieved by reducing the number of concurrent transactions.

The tests for the FOJ transformation were done with 50000 records in $R$ and 20000 records in $S$. For the split transformation, 50000 records were inserted into $T$. These were split into approximately 50000 records in $R$ and 20000 records in $S$.

Some important test results for split transformations are shown in Figure 4. As can be seen in Figures 4(a) and 4(b), the interference incured on user transactions heavily depends on the workload on the server, ranging from approximately 2% to 6% for throughput and 5% to 30% for response time. In these plots, 20% of all updates are on records in the source table. Little variation is observed for throughput tests while the response time tends to vary more with increasing server workload. Tests on concistency checking during split transformations and initial population of FOJ transformations show very similar results to those presented in Figures 4(a) and 4(b).

For log propagation to finish, more log records have to be propagated than generated. Because of this, the propagator needs a higher priority if many log records are generated than it needs if few are generated. Figure 4(c) illustrates this point. Two plots are shown: the lower plot is for tests where 20% of all generated updates are on records in $T$. The upper plot is for 80% updates on $T$, thus 4 times more relevant log records are generated during the same time interval. The operations that are not on $T$ update records in a dummy table to keep the workload constant. The priority of the transformation could be kept lower in the 20% case, resulting in less interference. Again, the same effect is observed on log propagation for FOJ on both throughput and response time.

As discussed, a reduction in the priority of the transformation process reduces interference. Unfortunately, this also increases the completion time of the transformation. Figure 4(d) shows how both the time needed to propagate log and the interference to throughput responds to the same changes in priority. The plot is for log propagation of split transformations with 75% workload on the server. FOJ tests show similar results. The transformation will never finish if the priority is set too low, in this case at about 0.5%. Clearly, the priority of the transformation must be chosen with care.

Transformations should for obvious reasons be executed when the workload on the server is as low as possible. If executed during off-hours, say at 50% workload, the observed interference should be acceptable on both throughput ($< 2\%$) and response time ($< 9\%$). During normal usage, say at 70% workload, the interference on throughput is still acceptable at approximately 2.5%. The interference to response time may, however, be too high. The cost should be carefully considered before the transformation is started. If too much interference is observed, the transformation should be aborted immediately. Aborting the transformation simply means that log propagation is stopped, and that the transformed tables are deleted.

Transactions that operate on the source tables could potentially be long lived. The completion time of the synchronization step is therefore much more predictable if the non-blocking abort strategy is used than if non-blocking commit is used. Synchronization takes less than 1 ms in the prototype tests with non-blocking abort. Interference plots for this step is therefore of little interest.

## 7 Conclusion and Further Work

A method to perform non-blocking full outer join (FOJ) and split schema transformations has been developed for relational databases. A prototype able to perform the transformations with approximately 2% interference on throughput and 5% on response time has also been developed. The results also show that interference increases with increasing workload. Because of this, database schemas should be transformed during periods with as low workload as possible.

FOJ and split are considered the most important nontrivial operators in a transformation framework because the normalization degree can be changed using these. Methods for other relational operators should, however, also be developed.

Even though it is not discussed in this paper, the split framework is able to split one source table into a many-to-many relashionship by repeating splits.

Non-blocking population of tables may have other important usages than schema changes. Using the technique to create other types of derrived tables like Materialized Views is an obvious example.

## References

1. R. Baylis and K. Rich. *Oracle9i Database Administrator's Guide Release 2 (9.2).* 2002.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Weslay Publishing Company, 1st edition, 1987.
3. J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc of the 1986 ACM SIGMOD Intl. Conference on Management of Data*, pages 61–71, 1986.
4. S. E. Bratsberg, S.-O. Hvasshovd, and Ø. Torbjørnsen. Parallel solutions in ClustRa. *IEEE Data Eng. Bull.*, 20(2):13–20, 1997.
5. L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc of the 1996 ACM SIGMOD Intl. Conference on Management of Data*, pages 469–480. ACM Press, 1996.
6. R. A. Crus. Data Recovery in IBM Database 2. *IBM Systems Journal*, 23(2):178, 1984.
7. H. Garcia-Molina and K. Salem. Sagas. In *Proc of the 1987 ACM SIGMOD Intl. Conference on Management of Data*, pages 249–259. ACM Press, 1987.
8. T. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Rec.*, 27(3):22–27, 1998.
9. A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Workshop on Deductive Databases, JICSLP*, pages 185–194, 1992.
10. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc of the 1993 ACM SIGMOD Intl. conference on Management of data*, pages 157–166. ACM Press, 1993.
11. H. Gupta and I. S. Mumick. Incremental maintenance of aggregate and outerjoin expressions. 2005.
12. S.-O. Hvasshovd. *Recovery in Parallel Database Systems.* Verlag Vieweg, 2nd edition, 1999.
13. S.-O. Hvasshovd, T. Sæter, Ø. Torbjørnsen, P. Moe, and O. Risnes. A continously available and highly scalable transaction server: Design experience from the HypRa project. In *Proc of the 4th International Workshop on High Performance Transaction Systems*, 1991.
14. IBM. *IBM DB2 Universial Database Administration Guide: Implementation, version 8.* IBM.
15. IBM. *IBM DB2 Universial Database SQL Reference, Volume 2.* IBM, 8 edition.
16. A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross. Concurrency control theory for deferred materialized views. In *Proc of the International Conference on Database Theory*, pages 306–320, 1997.
17. H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, 1983.

18. S. Marche. Measuring the stability of data. *European Journal of Information Systems*, 2(1):37–47, 1993.
19. Microsoft Corporation. Microsoft sql server 2000 books online, version 8.00.002 (sp3), published 17.01.2003.
20. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine- granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.
21. X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *Knowledge and Data Engineering*, 3(3):337–341, 1991.
22. D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc of the 4th International Conference on Parallel and Distributed Information Systems, 1996, USA*, pages 158–169. IEEE Computer Society, 1996.
23. M. Ronström. On-line schema update for a telecom database. *Proc of the 16th International Conference on Data Engineering*, 2000.
24. O. Shmueli and A. Itai. Maintenance of views. In *Proc of the 1984 ACM SIGMOD Intl. Conference on Management of Data*, pages 240–255. ACM Press, 1984.
25. D. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, 1993.
26. M. Widenius and D. Axmark. *MySQL Reference Manual*. O'Reilly & Associates Inc, 1 edition, 2002.
27. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc of the 1995 ACM SIGMOD Intl. conference on Management of data*, pages 316–327. ACM Press, 1995.