Non-blocking Creation of Derived Tables

Jørgen Løland
jorgen.loland@idi.ntnu.no

Svein-Olaf Hvasshovd
svein-olaf.hvasshovd@idi.ntnu.no

Department of Computer and Information Science
Norwegian University of Science and Technology

Abstract

Database systems are used in thousands of applications every day, including online shopping, mobile phone systems and banking. Many of these systems have high availability requirements, allowing the systems to be offline for only a few minutes each year.

In existing database systems, user transactions are blocked during materialized view creation and non-trivial database schema transformations. Blocking user transactions is not an option in systems with high availability requirements. A non-blocking method to perform such tasks is therefore needed.

In this paper, we present a method for non-blocking creation of derived tables, suitable for highly available databases. These can be used to create materialized views and to transform the database schema. The derived table creation may run as a low priority background process. Thus, the process has little impact on concurrent user transactions.

1 Introduction

Database (DB) systems are used in thousands of applications, spanning from simple personal DBs (CD or book collections) to highly available, complex systems with huge amounts of data, e.g. Home Location Registries in the telecom sector.

Applications change over time, thus requiring the schema of the DBs to change as well. Multiple studies have confirmed this: Marche [19] reports of significant changes to relational database schemas in a study of seven applications. Six of the systems had more than 50% of their attributes changed. The evolution of the schemas continued after the systems had been put into use. A similar study of a health management system [25] came to the same conclusion. Two ways to let a database evolve is to add materialized views and to transform the database schema.

Materialized views (MVs) may be added to a database to speed up processing of frequently used queries. During the last couple of decades, many authors have suggested methods to maintain consistency between MVs and the base tables they collect data from. The goal of their research (e.g. [1, 3, 9, 24]) has mainly been to interfere as little as possible with other transactions after the MVs have been created. The community has not

This paper was presented at the NIK-2006 conference. For more information, see //www.nik.no/.

shown the same interest in the initial creation of MVs, however. Thus, in today's systems, the MVs are created by read and insert operations that effectively block concurrent transactions from updating the involved tables.

A *database schema transformation* changes the table structure of a database. An example is to split records from one table into two new tables, e.g. because a department has been split into two physical locations. By using schema transformations, the database structure is able to evolve when system requirements change. In current DBMSs, non-trivial schema transformations are executed by an *insert into select* statement. The effect of this is the same as MV creation: read and insert operations block other transactions from updating the source tables.

The read and insert method described above may take minutes or more for tables with large amounts of data. Databases with high availability requirements should not be unavailable for long periods of time. Such databases, often found in e.g. the telecom and process regulation industries, would clearly benefit from mechanisms to perform these tasks without blocking. Although databases used in e.g. webshops, airline reservation systems and banking may be less critical, downtime should always be avoided to maintain good customer relations.

Both MV creation and schema transformations can be performed by creating a derived table (DT). In this paper, we describe a non-blocking method to create DTs using relational operators. These include vertical merge (full outer join) and split, horizontal merge (union) and split, difference and intersection.

The DT materialization method is based on log redo. This means that log records from the source tables are redone to the DTs in a similar way as normal crash recovery, but with redo rules adapted to each DT operation. We assume that the DBMS produces redo log records, and that undo operations produce Compensating Log Records (CLR) [4] as described for the ARIES method [21]. With CLR, all operations (including undo), are found in the same sequential log and with all the information necessary to apply the operations to the DTs. Furthermore, it is assumed that Log Sequence Numbers (LSNs) are used as state identifiers, and that the LSNs are associated with records in the database [10]. LSNs are commonly used in commercial systems, e.g. in SQL Server 2005 [20].

The paper is organized as follows: Section 2 describes other methods and research areas related to non-blocking DT creation. An overview of the framework and details for how to apply it to the six relational operators are presented in Sections 3 and 4, respectively. Finally, in Section 5, we conclude and suggest further work.

2 Related Work

Non-blocking creation of derived tables that involve any of the six operators has to the authors' knowledge only been researched in a schema transformation setting. Ronström [23] presents a non-blocking schema transformation method that uses both a reorganizer and triggers within user transactions. The method is able to perform horizontal and vertical merge and split, but methods for difference and intersection are not presented. Sagas [5] are used to organize the transformations. The reorganizer is used to scan 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, but triggers are used in a similar way to keep immediate MVs up to date. The extra workload

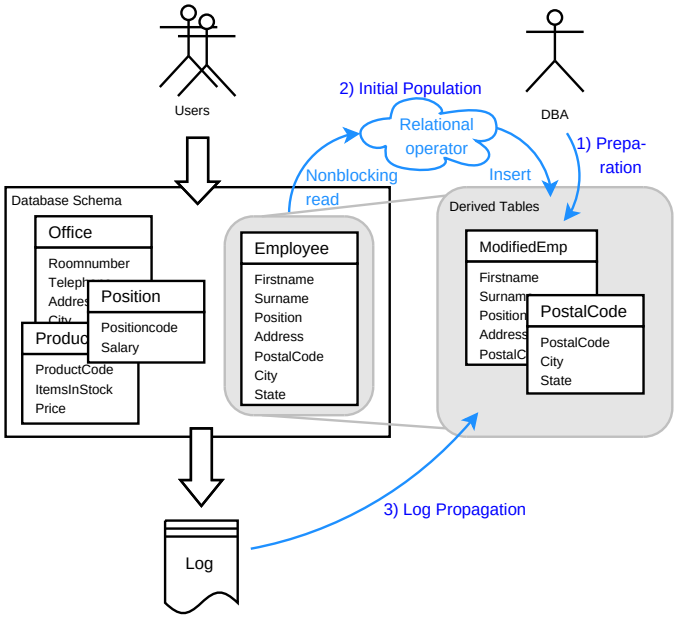

Figure 1: The first three steps of derived table materialization.

incurred by using triggers to update MVs is significant, and other update methods are therefore preferred whenever possible (see e.g. [3, 15]).

Fuzzy copy is a technique to make copies of a table without blocking update operations [2, 11]. 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 in ARIES [21] 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 CLRs to be used for undo processing.

Materialized views store the result of a query. They are used to speed up query processing and must therefore be consistent with the source tables. To the authors' knowledge, no method for non-blocking MV creation has been published. At first glance, maintenance of MVs has much in common with non-blocking DT materialization. However, all MV maintenance methods require the MVs to be consistent with a previous state of the source tables [1, 3, 6, 7, 8, 9, 15, 22, 24, 27]. Thus, since a fuzzy copy is not consistent with the source table, the MV update methods are not applicable.

Existing database systems, including IBM DB2 v8.2 [13, 14], Microsoft SQL Server 2005 [20], MySQL 5.1 [26] and Oracle 10*g* [16], offer some simple schema transformation functionality. These include removal of and adding one or more attributes to a table, renaming attributes and the like. Complex tranformations like the ones presented in this paper are not supported.

3 General Framework

The goal of the DT materialization framework is to provide a way to create a derived table without blocking other transactions. At the same time, the efficiency of concurrent transactions should be degraded as little as possible. After materialization, the DTs can be

used as MVs or to transform the schema. Using relational operators enables us to make use of existing, optimized code (like join algorithms) for parts of the process.

All DT materialization methods presented have four steps. As illustrated in Figure 1, the first three steps are to create the new tables, populate them with fuzzy copies of the source tables and then use modified recovery techniques to make the derived tables up to date with the source tables. The fourth step, *synchronization*, ensures that the DT has the same state as the source tables, thus making the DT ready for its intended use. The four steps are explained in general below.

Preparation Step

Before materialization starts, the derived tables must be added to the database. The tables must as a minimum include an LSN for all contributing source records and the attributes used to identify them in the log. In this paper, we use a Record ID (RID) as identifying attribute, but any unique identifier will work. Record IDs are commonly used internally by commercial DBMSs, e.g IBM 8.2 [12]. Depending on the operator, other attributes like join attributes in the case of vertical merge, may also be required. If any of the required attributes are not wanted in the DT, they must be removed *after* the materialization has completed.

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 DT materialization to abort. Any indices that are needed on the new tables should also be created at this point. In particular, since RIDs from the source tables are used to identify which records a logged operation should be applied to in the DTs, all six operators should have indices on source RID to speed up log propagation. Indices created during this step will be up to date when the materialization is complete.

Some of the relational operators require information that is not stored in the DTs. In these cases, an additional table may be required during the DT materialization process. This is commented on when needed.

Initial Population Step

The newly created DTs have to be populated with records from the source tables. This is done by a modified fuzzy copy technique, so the first step of populating DTs is to write a *fuzzy mark* in the log. This log record must include the transaction identifiers of all transactions that are currently 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 [11]. Once the source tables have been read, the relational operator is applied and the result, called the *initial image*, are inserted into the DTs.

Log Propagation

Log propagation is the process of redoing operations originally executed on source table records to records in the DTs. All these operations are reflected in the log, and are applied sequentially. By redoing the logged operations, the DTs will eventually have records with the same state as the source table records.

Log propagation starts when the initial images have been inserted into the DTs. Another fuzzy mark is first 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 DTs are then inspected and applied if necessary. 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.

When the log propagator reads a new log record, affected records in the DTs are identified and changed if the LSN indicate an older state than the log record represents. The effects of the propagation depend on the operator being used and are therefore described individually in Section 4.

If a schema transformation is the goal of the DT materialization, locks are maintained on records in the DTs during the entire log propagation process. By doing this, the locks are in place when the next step, synchronization, is started. Since locks are only needed when user transactions access both source and derived tables at the same time, 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 the source 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.

The log propagator will never finish executing if more log records are produced than the propagator can process in the same time interval. If this is the case, the DT materialization should either be aborted or get a higher priority.

Synchronization

When synchronization is initiated, the state of the DTs 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 DTs consistent with the source tables.

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

Blocking commit synchronization blocks all new transactions that try to access any of the source tables involved. Transactions that already have locks on the source tables are then allowed to complete before a final log propagation iteration is performed, making the DTs consistent with the sources. Depending on the purpose of the DT creation, either the source tables or the DTs are now available for transactions.

The non-blocking abort strategy latches the source tables for the duration of one final log propagation. Latching these tables effectively pauses ongoing transactions that work on them, but the pause should be very brief (less than 1 ms in our prototype [18]). Once the log propagation is complete, the DTs are in the same state as the source tables.

If the newly created DTs are to be used as materialized views, the preferred MV update strategy (e.g. [3, 9]) is used from this point. If the DTs are materialized to perform a schema transformation, the locks that have been maintained on the DTs since the first fuzzy log mark are made active. Records that are locked in the source tables are now also locked in the DTs. Note that locks forwarded from source tables conflict with locks set directly on a DT but not with each other [18]. Once the DT locks have been activated, transactions are allowed to access the unlocked parts of the DTs whereas transactions that operate on the source tables are forced to abort. Source table locks held in the DTs are released as soon as the propagator has processed the abort log record of the lock owner

Operator	Additional Information Needed
Vertical Merge	Rec. and state id problem
Vertical Split	Rec. and state id problem in T_r
Horizontal Merge (dup. rem.)	Rec. and state id problem
Horizontal Merge (dup. incl.)	
Horizontal Split	Missing record pre-state problem
Difference	Missing record pre-state: T_{int} and T_{comp}
Intersection	Missing record pre-state: T_{diff} and T_{comp}

Table 1: Summary of problems for DT creation operators.

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. In contrast to the previous strategy, however, transactions on the source tables are allowed to continue processing once the tables have been synchronized. If used for MV creation, there is no reason to choose the non-blocking strategy over this one. For schema transformations, however, this strategy enables transactions on the source tables to aquire new locks. These locks must be set on all involved records in the DTs as well as in the source tables, resulting in overhead and more locking conflicts. The non-blocking abort strategy may therefore be a better choice, especially in transformations where one DT record is composed of multiple source records, like join.

4 Descriptions of Derived Table Creation Operators

Before describing each operator, a few general problems that must be solved for all operator methods are discussed. First, there must be a way to uniquely identify which derived records a log entry should be applied to. It has been assumed that record IDs (RIDs) are used as a means to uniquely identify records. If only one source record contributes to each derived record, identification can be done by letting derived records store which source records contribute to them. This is not straightforward if one derived record may consist of multiple source records, however. This is called the *record identification problem.*

Second, there must be a way to identify which state a derived record represents. Recovery methods based on log, e.g. ARIES, use a Log Sequence Number (LSN) for this. For DT operators where one derived record may be derived from multiple source records, the LSNs of all contributors must be stored. This is called the *state identification problem*, and is encountered the same cases as the record identification problem.

Finally, the *missing record pre-state problem* is encountered if a record at some point in time during the materialization process does not qualify to be represented in any of the DTs. If the record is later updated, the log propagator has no knowledge of the previous state of its derived counterpart. This problem is found in the vertical split, difference and intersection operators, and is solved by letting these operators store excluded records in temporary tables.

Table 1 summarizes the problems that must be solved by the DT creation operators. The following sections provides a brief description of how the DT creation methods work. More details are provided in previous work by the authors [18, 17].

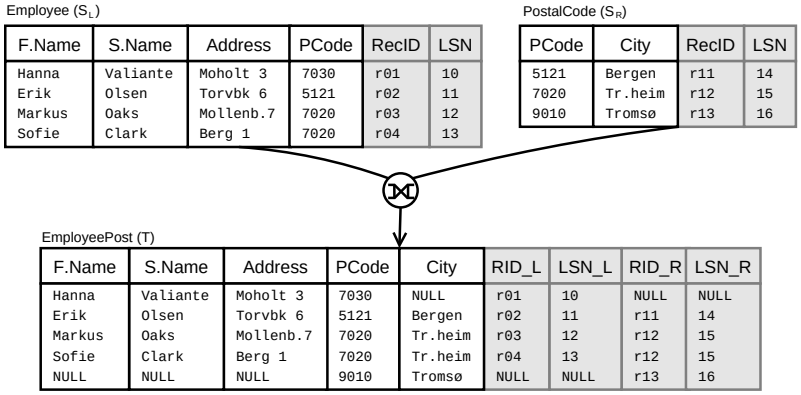

Figure 2: Example of vertical merge DT creation. Grey attributes are used internally by the DT creation process to solve the record and state identification problems. The reverse operation is an example of vertical split DT creation. The grey attributes are incorrect in a split context.

Vertical Merge (Full Outer Join)

Vertical merge takes two source tables, S_l (left source) and S_r (right source), and creates one derived table T by applying the full outer join operator. An example merge of "Employee" (S_l) and "PostalCode" (S_r) is shown in Figure 2. For readability it is assumed that the join attribute of table S_r (attribute $PCode$ in Figure 2) is unique, i.e. there is a one-to-many relation between the source tables. A merge of many-to-many relations is possible with minor changes to the described method.

In contrast to inner join and left and right outer join operators, FOJ is lossless in the sence that records with no join match are included in the result. In addition to the lossless property, there are multiple reasons for focusing on full outer join. First, the full outer join result can later be reduced to any of the inner/left/right joins by simply deleting all records that do not have appropriate join matches, whereas going the opposite way is not possible. Second, full outer join is the only of these operators that does not suffer from the missing record pre-state problem since all source records are represented at least once in the DT.

Vertical merge suffers from the missing record and state identification problems since two source records contribute to the same derived record. Since each record will be derived from at most two source records, the problems can be solved by adding a RID and LSN from each source table to the DT. These are denoted RID- and LSN- left and right in Figure 2. The method also requires the join attributes to be included in T for identification of join matches during materialization. All other source attributes are optional in T.

In addition to the indices on source RIDs (see Section 3), an index should be added to the join attributes of T. These indices provide fast lookup on the $T-$records that are affected by any logged operation.

After creating the table and indices, the full outer join operator is applied to the fuzzy read source tables, and the result is inserted into T. Special S_l- and S_r- NULL records are joined with records that otherwise would not have a join match, as illustrated in Figure 2.

Once the initial image has been inserted, the log propagator is started. The propagator has to make sure that insert, update and delete log records are applied to all relevant records in T. E.g., an insert of an S_r record must be propagated as an insert to all join matches in T. Updates of a join attribute is treated as a delete followed by an insert [17].

The log propagation rules are described in more detail in [17, 18].

Synchronization is performed as described in Section 3. If the DT is to be used for a schema transformation, the source table may then be removed.

Vertical Split

Vertical split is the reverse of the full outer join method described in the previous section. The method takes one source table S as input, and creates two DTs, T_l (left result) and T_r (right result), each containing a subset of the source table attributes. An example vertical split operation is that of splitting EmployeePost in Figure 2 into Employee and PostalCode. Note, however, that the grey attribute values are incorrect in a vertical split context.

In most cases, the source table S contains multiple records with equal T_r-parts, e.g. multiple people with the same postal code. These records should be represented by only one record in T_r. Furthermore, a record in T_r should only be deleted if there are no more records in S with that T_r-part. To be able to decide if this is the case, a *counter*, similar to that of Gupta et al. [8], is associated with each T_r-record. When a T_r- record is first inserted, it has a counter of 1. After that, the counter is increased every time an equal record is inserted, and decreased every time one is deleted. If the counter of a record reaches zero, the record is removed from T_r.

Intuitively, T_r suffers from the record and state identification problems due to the fact that multiple source records may contribute to the same T_r-record. Since all source records are represented in both derived tables, however, these problems are easily solved by adding a source RID and LSN to the T_l table. Thus, when a source record is updated, the T_l-record with the correct source RID is looked up. If the record exists in T_l and has a lower LSN than the log record, the update is applied to both derived records. The T_r- record is identified by using the join attribute found in T_l. Note that the source LSN of T_l should be updated to that of the log even in cases where only T_r attributes are updated.

It should be clear by now that T_l must include both source RID and LSN, in addition to the join attributes. Only the join attributes are required in T_r. All other source attributes are optional. Indices should be added to the source RID and join attributes of T_l, and to the join attributes of T_r. No other attributes are used to identify records.

Log propagation always starts by finding the record with the correct source RID in T_l. The existence and state of that record is used to determine if the logged operation should be applied to the derived records [17].

The described method works under the assumption that the source records are consistent. However, a problem arises if multiple source records that contribute to the same T_r-record, i.e. have the same join attribute, do not have equal T_r attribute values. If the two records with postal code 7020 in Figure 2 had different city names, e.g. “Trnodheim” and “Trondheim”, the method would fail. Such cases require special treatment [18].

Horizontal Merge (Union)

Horizontal merge adds records from two source tables, S_1 and S_2, into T_{union}. If duplicate records are included in T_{union}, the method is straightforward. All source records are represented unmodified in T_{union}, and the source RIDs and LSNs therefore identify the records and record states in T_{union}.

Horizontal merge with duplicate removal is more complex. Duplicate records from the source tables are represented by only one record in T_{union}, and logged operations

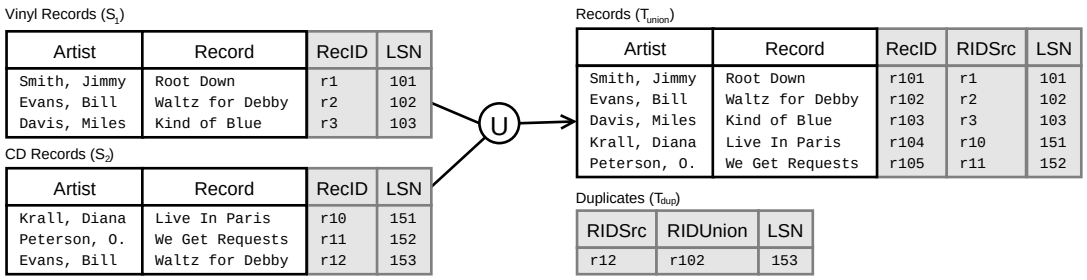

Figure 3: Union Derived Table with Duplicate Removal. Grey attributes are used internally by the DT materialization process, and are not visible to other transactions.

may merge new duplicates and split former duplicates into multiple records. The record and state identification problems are incurred by these duplicates. Figure 3 illustrates a horizontal merge with a duplicate pair with RID $r2$ and $r12$. RecID denotes the record ID in the DT, and RIDSrc denotes the ID the record from the source table. The Figure also shows a table "duplicates" (T_{dup}) used to store RID from the source table, RID from the union table and the LSN of duplicate records. This solves the two identification problems.

In addition to the index on source RID in T_{union}, indices should be added to both source and union RID in T_{dup}.

After the DTs are created, the source tables are read without setting locks. One record of each unique source record is then inserted into T_{union}; duplicates of these are inserted into T_{dup}. Both tables are used during log propagation. An insert may, e.g., result in an insert into either T_{union} or T_{dup}, depending on the existence of an equal $T_{union}-$record. Furthermore, if a record with duplicates is deleted from T_{union}, one of the records in T_{dup} must be moved to T_{union}. Update operations may both create and remove duplicates. If either the pre- or post-update version of the record has duplicates, the update is treated as a delete followed by an insert [17].

Horizontal Split

Horizontal split is the reverse of union. The operator takes records from one source table, S, and distributes them into two or more tables $T_{1..n}$ by using selection criterions. Example criterions include that of splitting an employee-table into "New York employee" and "Paris employee" based on location, or into "high salary employee" and "low salary employee" based on a condition like "salary $>$ \$40.000". The selection criterions may result in non-disjunct sets, and may even not include all records from the source table.

Records in $T_{1..n}$ are always derived from one and only one source record. Thus, record and state identification is acheived simply by including source RID and LSN as attributes in the derived records. Since a record may not satisfy the selection criterion of any DT, however, the operator suffers from the missing record pre-state problem. This is solved by using a temporary table T_{tmp} that stores all records that do not satisfy any of the DT selection criterions. As an example, consider the employee table that was split into New York and Paris offices. An employee in London would not match any of these. If a log record describes an update reflecting that the employee has moved to the Paris office, however, the log propagator will need the pre-update state found in T_{tmp}. The temporary table is removed once the synchronization step has completed.

With source RIDs and LSNs included in the DTs, log propagation for horizontal split is similar to normal crash recovery. The difference is that source records may be derived into multiple DTs. Hence, all instances of the record must be identified. For inserts, the

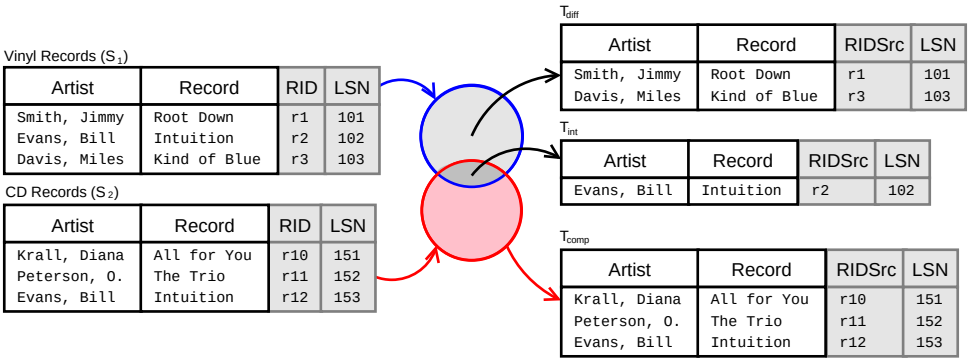

Figure 4: Difference and intersection DT creation. Grey attributes are used internally by the DT materialization process, and are not visible to other transactions.

attribute values found in the log are analyzed, and the record is inserted directly into the correct DTs. Propagation of delete and update operations, on the other hand, must start by scanning all DTs to find all records with the correct source RID before applying the logged operation. Update log operations may even have to move records between DTs, i.e. delete a record from one DT and insert it into another.

Difference and Intersection

The difference and intersection operators are so closely related that the same method is applied to create both kinds of DTs. The method compares records from one source table S_1 with the records in another source table, S_2. In the difference case, records that are represented only in S_1 are added to the DT. In the intersection case, records that are represented in both source tables are added to the DT.

Difference and intersection suffers from the missing record pre-state problem for two reasons. First, a record that at one point in time belongs to the difference set may later be updated so that it should belong to the intersection set and vice versa. Thus, the state of records that do not qualify for the DT being created is needed. Second, operations applied to records in S_2 may make a record derived from S_1 to move between the difference and intersection sets, depending on the old and new attribute values of the S_2 record. Thus, the derived states of S_2 records are also needed. The first problem is solved by always using a table for both the difference and the intersection sets. The second problem is solved by storing the derived state of S_2 records in a temporary table called T_{comp} (compared-to).

Figure 4 shows the involved tables: S_1 is compared to S_2, and difference and intersection sets are inserted into T_{diff} and T_{int}, respectively. In addition, the records from S_2 are stored into T_{comp}. Records in the DTs are always derived from only one source record, and records and states are therefore easily identified by adding a source RID and LSN attribute. Indices should be added to the source RID attributes of all derived tables.

When the source tables have been fuzzily read, records from S_2 are inserted into T_{comp}, whereas records from S_1 are inserted either into T_{diff} or T_{int}. Log propagation is then started. Insert of S_1-records are ignored if the record already exists in either T_{diff} or T_{int}. Otherwise, T_{comp} is scanned for an equal record to determine which of the tables it should be inserted into. Delete operations for S_1 removes the record from whichever DT it resides in. Updates may require a record to be moved between T_{diff} and T_{int}.

The deletion of a record from S_2 is propagated as a delete from T_{comp}, and may require a record in T_{int} to move to T_{diff}. The opposite may be the result of an insert into S_2.

Finally, update operations affecting a record in T_{comp} may require records to be moved between T_{diff} and T_{int}.

5 Conclusion and Further Work

A method to perform non-blocking derived table materialization for six common relational operators has been developed for relational databases. Once created, these derived tables can be used as materialized views or for non-trivial schema transformations. In contrast to the method described in this paper, current commercial DBMSs block the involved tables during these task which may take minutes or more when large source tables are involved.

The method has been shown to incur little response time and throughput interference to normal transactions executing concurrently with the vertical merge and split methods [18]. All described operators will be tested with the same prototype to verify these results. Since all operators use the same technique, however, the results are expected to be very similar in all cases.

References

[1] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD Conference on Management of Data*, pages 61–71, 1986.

[2] S. E. Bratsberg, S.-O. Hvasshovd, and Ø. Torbjørnsen. Parallel solutions in ClustRa. *IEEE Data Eng. Bull.*, 20(2):13–20, 1997.

[3] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 469–480. ACM Press, 1996.

[4] R. A. Crus. Data Recovery in IBM Database 2. *IBM Systems Journal*, 23(2):178, 1984.

[5] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259. ACM Press, 1987.

[6] T. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. *ACM SIGMOD Record*, 27(3):22–27, 1998.

[7] 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.

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 157–166. ACM Press, 1993.

[9] H. Gupta and I. S. Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Information Systems*, In Press, 2005.

[10] S.-O. Hvasshovd. *Recovery in Parallel Database Systems*. Verlag Vieweg, 2nd edition, 1999.

[11] S.-O. Hvasshovd, T. Sæter, Ø. Torbjørnsen, P. Moe, and O. Risnes. A continuously available and highly scalable transaction server: Design experience from the

HypRa project. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems*, 1991.

[12] IBM. *IBM DB2 Universal Database Glossary, Version 8.2*. IBM.

[13] IBM. *IBM DB2 Universial Database Administration Guide: Implementation, version 8*. IBM.

[14] IBM. *IBM DB2 Universial Database SQL Reference, Volume 2*. IBM, 8 edition.

[15] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross. Concurrency control theory for deferred materialized views. In *Database Theory - ICDT '97, Proc of the 6th International Conference*, volume 1186 of *Lecture Notes in Computer Science*, pages 306–320. Springer-Verlag, 1997.

[16] D. Lorentz and J. Gregoire. *Oracle Database SQL Reference 10g Release 1 (10.1)*. 2003.

[17] J. Løland and S.-O. Hvasshovd. Non-blocking materialized view creation and transformation of schemas. In *Advances in Databases and Information Systems - Proceedings of ADBIS 2006*, volume 4152 of *Lecture Notes in Computer Science*, pages 96–107. Springer-Verlag, 2006.

[18] J. Løland and S.-O. Hvasshovd. Online, non-blocking relational schema changes. In *Advances in Database Technology – EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 405–422. Springer-Verlag, 2006.

[19] S. Marche. Measuring the stability of data. *European Journal of Information Systems*, 2(1):37–47, 1993.

[20] Microsoft Corporation. Microsoft sql server 2005 books online, http://www.microsoft.com/technet/prodtechnol/sql/2005/downloads/books.mspx (december 6, 2005).

[21] 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.

[22] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *Knowledge and Data Engineering*, 3(3):337–341, 1991.

[23] M. Ronström. On-line schema update for a telecom database. In *Proc. of the 16th International Conference on Data Engineering*, pages 329–338. IEEE Computer Society, 2000.

[24] O. Shmueli and A. Itai. Maintenance of views. In *Proceedings of the 1984 ACM SIGMOD International 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 5.1 Reference Manual*. 2006.

[27] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 316–327. ACM Press, 1995.