Query Processing for SQL Updates

César A. Galindo-Legaria
cesarg@microsoft.com

Stefano Stefani
stefanis@microsoft.com

Florian Waas
florianw@microsoft.com

Microsoft Corp.,
One Microsoft Way,
Redmond, WA 98052

ABSTRACT

A rich set of concepts and techniques has been developed in the context of query processing for the efficient and robust execution of queries. So far, this work has mostly focused on issues related to data-retrieval queries, with a strong backing on relational algebra. However, update operations can also exhibit a number of query processing issues, depending on the complexity of the operations and the volume of data to process. Such issues include lookup and matching of values, navigational vs. set-oriented algorithms and trade-offs between plans that do serial or random I/Os.

In this paper we present an overview of the basic techniques used to support SQL DML (Data Manipulation Language) in Microsoft SQL Server. Our focus is on the integration of update operations into the query processor, the query execution primitives required to support updates, and the update-specific considerations to analyze and execute update plans. Full integration of update processing in the query processor provides a robust and flexible framework and leverages existing query processing techniques.

1. INTRODUCTION

Data update operations are a necessary part of customer applications, and their efficient processing is as important as that of data-retrieval operations. When data is inserted, deleted or modified in a database, the DBMS (Database Management System) needs to validate the changes against declared constraints and maintain the underlying storage structures to preserve a consistent and correct representation of the data.

Some aspects of update processing have been thoroughly studied and documented in the past. Transactions and concurrency control issues [1] are examples where formal models, algorithms, and performance analyses have been developed over time and contribute to our understanding of the problem space. It has also led to successful and robust commercial implementations.

Depending on the complexity of the operations and the volume of data to process, standard query processing issues also surface during the execution of updates. These issues include lookup and matching of values, navigational vs. set-oriented algorithms and tradeoffs between plans that do serial or random I/Os. This motivates the integration of update processing in the general framework of query processing. To be successful, this integration needs to model update processing in a suitable way and consider the special requirements of updates. One of the few papers in this area is [2], which deals with delete operations, but there is little documented on the integration of updates with query processing, to the best of our knowledge.

In this paper, we present an overview of the basic concepts used to support SQL Data Manipulation Language (DML) by the query processor in Microsoft SQL Server. We focus on the query processing aspects of the problem, how data is modeled, primitive operators and different execution plans. There are a number of other aspects required in a complete implementation, notably concurrency control and locking, and algorithms for the manipulation of B-trees. Those subjects are outside of the scope of the present paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2004 June 13-18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 . . . $5.00.

2. SQL UPDATES

The concepts presented in this paper can be applied to a number of data modification scenarios, but we focus on the basic operations and functionality provided by SQL. This functionality is briefly reviewed in this section. Examples throughout the paper use the well-known TPC-H schema [4].

The basic data-modification statements provided by SQL are *INSERT*, *DELETE*, and *UPDATE*. Their target is a single table, although multiple tables may be modified as a result of the changes, as we shall see later. INSERT takes a collection of rows computed using an arbitrary SQL query and adds those rows to the target table. DELETE and UPDATE use a WHERE condition to qualify the rows of interest, either to be removed from the table or to assign values to their columns. It is valid to use subqueries in either the qualifying predicate or the assignment values, so multiple tables and arbitrarily complex queries are often part of data modification statements.

The following simple queries show deletion and update of a row in the ORDERS table:

```
DELETE ORDERS
WHERE O_ORDERKEY = 3271

UPDATE ORDERS
SET O_PRIORITY = URGENT
WHERE O_ORDERKEY = 3271
```

Update operations need to be validated against declared constraints. Several forms of constraints are supported by SQL.

Check constraints are predicates that enforce the domain of a column, or relationships between columns of a single table. For example, a constraint can be O_ORDERSTATUS IN (“O”, “F”, “P”).

When inserting a row or updating column values, the DBMS rejects the operation if the column values fail to satisfy the check constraint. A common special case of this form of constraint is to declare a column *NOT NULL*.

Referential integrity (RI) constraints establish a relationship between tables, indicating that the values appearing in one of the tables must be found in the key of another table. For example, the values of column O_CUSTKEY in ORDERS must be found in C_CUSTKEY of CUSTOMER. The column C_CUSTKEY is called the referenced key, and the column O_CUSTKEY of ORDERS is called the foreign key.

Finally, *uniqueness constraints* indicate that the value of one of more columns cannot have duplicates within the table, such as C_CUSTKEY in CUSTOMER.

In addition to validation, update operations need to maintain the underlying storage structures. From a query processing point of view, maintenance consists of finding all the underlying structures that represent the logical data and issuing the necessary modifications to keep them consistent and accurate for the data represented. In practice, this means updating all necessary indices and materialized views that depend on the changed data. Due to space limitations, we will focus on index maintenance issues in this paper.

Cascading RI constraints combine validation with maintenance of structures. They specify, for example, that if the value of C_CUSTKEY is changed then the system needs to automatically change the value of the referencing O_CUSTKEY to preserve the constraint. This is one example where multiple tables are modified as a result of an update statement, even though its target is a single base table.

3. MODELING UPDATES IN QUERY PROCESSING

The query processor of Microsoft SQL Server is based on an algebraic, extensible model that supports the addition of new operators as necessary. Both in query optimization and in query execution, the framework provides a contract for operators and manipulates trees of those operators. This general philosophy is formulated in [3] and it serves as the background for the work described here.

The following concepts make up the backbone of our update processing approach:

Delta stream. A delta stream is a collection of rows encoding changes to a particular base table. This is simply a relation with a well-defined schema and as such it can be processed by any relational operator. Two different relational encodings of delta streams will be detailed later.

Application of a delta stream. A side-effecting operator called StreamUpdate issues a data-modification command to the storage engine for each of its input rows. Typically, an update plan will contain several instances of StreamUpdate to maintain the various physical structures.

Mapping updates to a relational processing setting in this way allows leveraging all the existing relational technology and execution infrastructure. This is accomplished using a relational expression made up of standard operations, optimized as such and executed using all available execution algorithms.

For the optimizer, a concept that needs to be incorporated is that it handles a side-effecting operator. Unlike regular relational operators, which are purely functional and side-effect free, StreamUpdate needs to be executed to completion and exactly once.

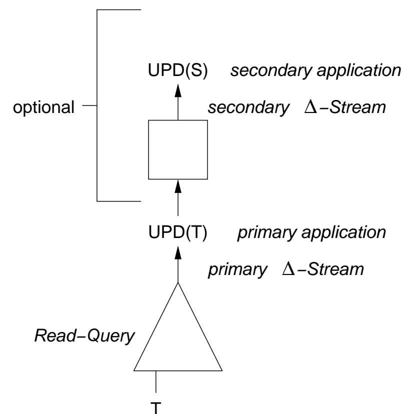

Figure 1: General form for update plan.

Figure 1 shows a general template for the execution plan of update statements. It comprises two components. The first is read-only and responsible for delivering a delta stream that specifies the changes to be applied to the target table. The second component consumes the delta stream, applying the changes to the base table, and then performs all the actions that the DML statement implicitly fires. The list of actions to perform is determined by enumerating all the active dependencies against the target table and deciding which of these are required by the current statement.

4. INDEX MAINTENANCE

4.1 Indices in Microsoft SQL Server

In Microsoft SQL Server, each base table has a "primary access path" storing the table contents. Such primary path can be either a *clustered index*, if organized as a B-tree, or a *heap* otherwise. In either case, there is always a *locator* consisting of one or more columns whose value uniquely identifies a row within the table and can be used to locate it efficiently. A heap is not ordered; however, a row is uniquely identified by a physical locator called RID. In a clustered index, a row can be uniquely identified with the values for the index keys, if the index is unique. If the clustered index is non-unique, an extra, hidden column is added to the index keys, called *uniquifier*. The uniquifier column is used to distinguish between two rows with identical clustered index keys, and is automatically assigned by the storage engine when populating or maintaining the B-tree.

Secondary indices, also called *non-clustered indices* in Microsoft SQL Server, can also be created for the base table and they also correspond to B-trees. These indices need to carry, together with their index keys, also the locator of the primary access path, to make base table lookups possible. The physical keys of any B-tree need to be unique, which can be achieved by adding the locator to the declared key of non-unique non-clustered indices.

Non-clustered indices can optionally have *included columns*. An included non-key column is stored at the leaf level of the B-tree, without being added to the columns the index is sorted on. At the level of data structures, this is actually the behavior of the clustered index for all the table columns not belonging to the clustering key, and it is an option available for non-clustered indices as well. The index cannot be used to implement seeks against the included column. It makes however possible to construct query plans based on the index that do not require extra base table lookups in order to

retrieve the column value. In terms of update processing, included columns can be updated "in place," since there is no need to move the row if key values are not changed.

Both clustered and non-clustered indices can contain *computed columns*. A computed column is one whose value is defined as the result of a scalar expression over other columns of the same table. An example could be the year of a date field, or the difference between CommitDate and ShipDate. A computed column is not directly assignable, as its value is implicitly modified by setting or changing the columns it is defined against.

Another option for indices is *uniqueness*. When the keys of an index are declared as unique, the storage engine will reject through an exception the insertion of new entries identical to existing ones in the B-tree. This causes the statement to be aborted. Since an update to the keys of an index is internally implemented as a delete followed by an insert, the technique also works for updates.

4.2 Primary Application of a Delta Stream

As shown in Figure 1, the read query component of the update plan provides a description of the changes to apply. In the case of delete, the delta stream contains the locators of rows to remove; for insert it contains column values for rows to insert; and for update it contains the locator of the rows to modify and the new values to assign. The StreamUpdate operator is set up at compilation time to know the format of its input delta stream. It issues the appropriate commands to the storage engine layer to locate, modify, insert or delete rows of the table.

Some column values required for non-clustered index maintenance are obtained only after the primary access path has been updated. For example, if the table is organized as a heap, the locator (i.e., RID) of a new row will only be available after the row is inserted. After the primary update application we can get a delta stream whose rows have a complete set of columns for all the old and new values.

The way to expose old values for columns modified by an update is through a *pre-copy* of such values, occurring after locating the base table row, but before applying the update. The pre-copy operation can be expensive, especially for columns of large size, and it is performed only if the values are required either for maintenance of non-clustered indices or materialized views.

Once we come to maintenance of a non-clustered index, the delta table provided by the primary update application contains all the necessary information: keys to locate a row to delete or modify, and values to set or insert.

4.3 Multiple Index Maintenance

Dependency analysis is a first step to generate update plans. This consists of identifying the indices affected by the DML statement being processed. Insert and delete statements need to be propagated to all the indices, while updates are optimized to only maintain the non-clustered indices that carry columns being modified. Since non-clustered indices store the base table locators, updates affecting any locator columns will require the maintenance of all the indices. If an index stores a computed column, the query processor gathers the list of regular columns it depends on, and if any of them is being updated, the computed column will be assumed to be changing.

Microsoft SQL Server has two different ways of maintaining non-clustered indices, *per-row* and *per-index*. With a per-row maintenance plan, a single execution operator applies the changes for each row of the delta stream to both the base table and all the non-clustered indices. Changes are propagated from the base table to the non-clustered indices on a row by row basis.

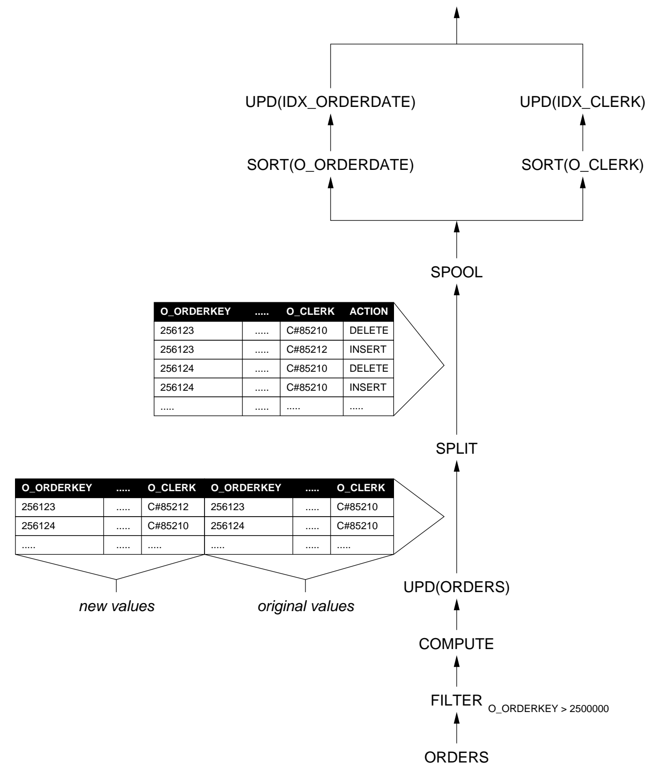

Figure 2: Update plan with per-index maintenance.

In a per-index plan, a non-clustered index is maintained by a different update operator. This allows the index to be maintained separately from the others, and *after* all the changes have been applied to the base table or clustered index. In insert and delete plans, the per-index sub-plans are fed by the delta stream delivered by the base table maintenance operator. When more than one non-clustered index is maintained in a per-index basis, the delta stream will be stored in a temporary data spool, and read by each of the index maintenance sub-plans. This allows maintaining one index at a time, rather than one row per each index at a time. Figure 2 shows an update plan that does per-index maintenance of two indices. The *compute* operator defines the values to be assigned in each row of the update.

The main motivation behind per-index plans is that maintaining one index at a time can yield better data locality because there is no need to switch back and forth between pages belonging to different indices. Data locality can be dramatically improved by sorting the stream that feeds an index maintenance plan on the index keys, before the changes are applied. It is possible to sort the stream to apply differently for each index and this way achieve much better I/O behavior, which is unfeasible in a per-row plan. The decision whether to sort the stream is made in a cost-based fashion.

Sorting on index keys guarantees index pages to be touched at most once—as long as the index key itself is not being modified. Otherwise, if the index key is changing, this will result in a row being moved within the B-tree and therefore re-introduces some random I/O behavior. We transform the delta stream into a different representation to overcome this problem.

The standard update delta stream contains both the old and new values of the table columns. It is transformed into a *split delta stream* by splitting each row into two, one containing the old values, the other the new values. This is done by a *split operator*. An

additional *action column* is introduced in the stream as part of splitting. The purpose of the action column is to distinguish between rows carrying old and new values. Per-index update operators are fed a stream that has gone through the split operation. They apply changes to the non-clustered index as series of deletes and inserts, rather than with direct updates. The value of the action column indicates, for each row, whether the current operation should be a delete or an insert. The format of the delta streams and the position of the split operator is shown in Figure 2.

Sorting the split delta stream on the index keys first and then the action column guarantees optimal data locality for updates. Locating a row and moving it to the new location are performed in optimal sequence. Also, this strategy resolves a potential problem with spurious unique key violations. For example, if there is a unique index on column A and an update is made of the form A = A + 1, then updating some rows may cause a temporary violation of uniqueness. However the update describes a change that is consistent with the uniqueness constraint. Since we sort on keys plus action column, no spurious unique key violations would ever occur at the storage engine level, because an index key will always be deleted before being inserted back.

The StreamUpdate operator is set up at compilation time to know the format of its input delta stream, either a standard delta table or a split format.

4.4 Choosing an Update Plan

The query processor decides whether to perform per-row or per-index maintenance on an index by index basis. It is possible to have different indices maintained in different ways inside the same plan. There are several factors to consider when taking the decision.

- Per-row plans have some internal limitations. For example, in the current implementation they cannot update a non-clustered index of a computed column, as the required scalar computation is performed by a separate operator. This will sometimes force the query processor to opt for a per-index solution for a particular index. In most of the common cases, however, both solutions are available.

- A per-index maintenance plan requires more operators in the query plan, causing overhead during query execution. The compilation time is longer, and the plan requires more memory.

- When more than one nonclustered index is maintained in a per-index basis, spooling an intermediate result is required in the plan to store the delta stream. Spools have a population cost that depends on the number and size of the columns, and the number of rows. There is also a cost related to reading from a worktable, but it is usually smaller than that of populating it.

- There are scenarios where a spool has to be introduced above the base table update for reasons that are unrelated to non-clustered index maintenance. Examples that will be covered later are checking self-referential integrity and maintenance of materialized views. In these cases, the populating of the spool can be usually considered as a given, even if adding per-index maintenance to the query plan might require storing extra columns as part of the spool operation. Such columns are index keys and typically of small size.

- Per-index plans can be combined with referential integrity validations, as we show later.

We have described "wide" index maintenance plans, where the delta stream is spooled and then consumed by multiple branches that update each index. We also considered "stacked" per-index maintenance, where the delta stream is piped through a series of sorts followed by index updates. We do not use this because the lower sort operations are required to copy all the columns needed by all later index operations. Instead, to keep the size of the sort buffers as small as possible, per-index plans are organized such that each index maintenance sub-tree reads from the common spool and processes only the columns required by the particular index.

Choosing between per-index and per-row depends on the structure of the table, its current size, the type of DML operation being performed, the number of rows it will affect, and the other actions that need to take place together with index maintenance, e.g., foreign key validations. Per-row plans are clearly the preferred choice for DML statements affecting a limited amount of rows, as they introduce less overhead, both in compilation and execution. As the amount of data to be maintained grows, per-index plans become more appealing, and scale far better as the execution of the query takes a big toll on the system. Insertions into empty or almost empty tables should always be performed per-index. With only very few exceptions where choosing the per-index strategy is mandatory, both are equally functional, and the query processor is allowed to take a choice purely based on performance, on an index by index basis.

4.5 Cache Effects

A straightforward way of optimizing DML queries could be to separately process the read-only part of the plan, as if it was a standalone select query. Not acknowledging the context in which the read-only portion is being processed could however lead to poor query plans. One of the reasons is that the pages that are loaded into memory when reading a base table or a non-clustered index may or may not be re-used later by the operators that apply the changes in the plan. An update plan always has to maintain the base table, but will only propagate the changes to the indices affected by the statement. Reading from an index that is later on modified could make its maintenance more efficient, as the operation will have better chances of finding the data pages it has to work with already cached. This is especially true with the base table, as it is always maintained first. Obviously reading from a particular non-clustered index could allow very efficient seeks in presence of a selective predicate in the update statement, and this could easily overshadow the benefits of populating the cache. If, however, there are different alternatives to consider from, and none of them allows considerably better seeks, taking into account the cache warm-up factor could lead to a different plan choice, especially if one of the alternatives is the base table.

5. CONSTRAINT VALIDATION

5.1 Single-table constraints

We have stated earlier that unique and primary key constraints are enforced by the storage engine upon insertion into B-trees. The one issue the query processor needs to address is to avoid spurious uniqueness violations. We described the problem and its solution earlier in the discussion about index maintenance.

Check constraints are defined as a Boolean scalar expression over a set of one or more columns in the table. If the expression does not evaluate to TRUE, an exception is raised, the statement aborted and the transaction is rolled back. They apply to insert and update statements. An example is (ShipDate <= ReceiptDate).

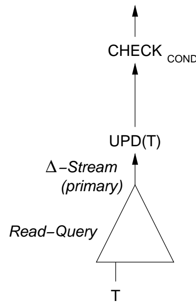

Figure 3: Update plan with constraint checking.

Check constraints are enforced by the query processor with the following two step process:

1. Check constraints affected by the statement are identified, analyzing dependencies with an approach similar to index maintenance. For insert statements, all check constraints defined on the table are enforced. For update statements, a list of the columns referenced inside each constraint is collected. A constraint will be enforced only if at least one of the columns is being modified.

2. For each constraint to be enforced the query processor adds to the query plan a generic operator which evaluates a scalar expression for each input row it receives. It throws an exception if the value computes to FALSE. This exception rolls back the transaction undoing any modifications executed up to that point. The operator is added to the query tree above the base table update, i.e., check constraints are enforced after the row has been inserted or updated in the base table.

A generic plan for checking constraints is shown in Figure 3.

5.2 Referential Integrity Constraints

A foreign key constraint establishes a one-to-many relationship between two tables, respectively called referenced and foreign (or referencing) side. The relationship involves a unique key on the referenced side, and a matching set of columns on the foreign side. The system enforces and guarantees that for each row in the referencing table, one and only one row exists in the referenced table with the same values for the set of columns involved in the constraint. An example is the relationship between the CUSTOMERS and ORDERS tables, through the C_CUSTKEY/O_CUSTKEY columns. Unlike the previous constraints, foreign key validations can also take place when processing delete statements. Foreign key constraints are handled by the query processor in a fashion similar to the one applied to check constraints:

1. First, the list of constraints to be enforced is identified. The cases are:

    - Insert to the foreign table. It must be verified that a matching key exists in the referenced table.

    - Update to a column involved in the constraint in the foreign table. The new value must exist in the referenced table.

    - Delete to the referenced table. No matching foreign key must exist in the referencing table.

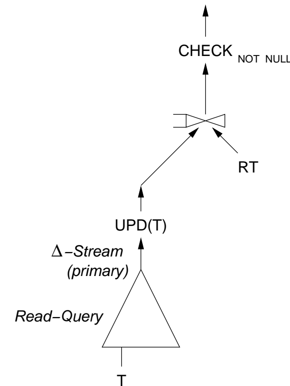

Figure 4: Checking referential integrity constraint.

    - Update to a column involved in the constraint in the referenced table. The old value must not exist in the referencing table.

2. For each constraint to be enforced, the query processor adds a sub-tree to the query plan, which outer-joins the columns read from the base table row with the other table involved. When the DML statement affects the referencing side, the set of new values of these columns is used, and it is enforced that one row exists on the other side of the join. Conversely, when the update target is the referenced side, the old values from the base table row are used, and it is verified that no row on the foreign side satisfies the join predicate. Errors are thrown through the same operator used for check constraints. Referential integrity constraints verification sub-trees are put above the base table update in the query plan.

Figure 4 shows a plan to verify referential integrity constraints. An outer-join is used to lookup the matching row in the other table so that verifying the reference becomes checking for nulls.

Running the verification joins after having updated, and consequently exclusively locked, the base table row of the target of the DML statement allows better handling of concurrency issues. Verifying the constraint after updating the referenced side has, however, also a performance drawback. The join has to be performed using the pre-update values of the columns, to ensure that there are no outstanding rows in the referencing side pointing to the old key. This forces the query processor to save off a copy of the old value of the key columns being modified before the update takes place. The columns to be pre-copied are, however, part of a unique key, and this guarantees that their size never exceeds a certain threshold.

The join that gets added for referential integrity verification is optimized just like any other by the query processor. If the join is against the referenced table, the unique index that must exist over the keys involved in the constraint is perfectly suited to implement the plan. There is, however, no requirement of any kind on the referencing side. This means that a delete or update to the keys to the referenced table might result in a join requiring a full table scan of the foreign table. It is a good database design practice to build a non-unique index on the foreign columns of the referencing table if such DML operations are planned on the referenced side.

Referential integrity constraints for which the referenced and foreign tables coincide are called self-referential. An example is an EMPLOYEES table, where the ManagerId column points to the EmployeeId. In order to properly validate self-referential con-

straints, it is necessary to first complete the entire set of changes required by the DML statement, and then start the verification process, as this should not be based on transient, temporary data. The separation between the phases is implemented by saving the delta stream delivered by the base table update to a temporary spool, and reading from it in order to perform the validation. This solution is perfectly functional for all cases of referential integrity, but is only used by Microsoft SQL Server in presence of self-referential constraints, as populating and reading from the worktable is an expensive operation.

When a non-clustered index over the columns involved in a referential constraint is maintained with a per-index plan, the query processor will tie constraint validation and index maintenance together. The constraint validation sub-tree will not be placed on top of the base table maintenance operator as usual, but rather above the branch propagating the changes to the non-clustered index. In most cases, the presence of an index over the same columns is expected on the other side of the constraint, and it is a requirement to have a unique index on the referenced table. If such index on the other side of the constraint validation join exists, it represents an excellent candidate for an index-lookup plan. Sorting the delta stream to improve the performance of index maintenance will then also yield optimal data locality when seeking into the index. For example, when optimizing a delete statement against the CUSTOMERS table, a per-index strategy could be chosen by the query optimizer for the maintenance of the unique index over C_CUSTKEY. If the stream is sorted on C_CUSTKEY before deleting from the index, such sort will also guarantee optimal data locality when seeking into a non-unique index on the same column of the ORDERS table, to verify that no row exists for that particular key value. A further improvement is then introduced when the target of the DML statement is the foreign table. The index to be maintained will typically not be unique, being a foreign key usually a representation of a one-to-many relationship, and it is possible that the same value of the key will be processed multiple times. If the stream is ordered on such keys, it is efficient to discard the duplicates with a streaming distinct operation, so the join is only performed over distinct values. For example, if bulk inserting a million orders for a certain customer, it is not necessary to validate the existence of the customer more than once. In general, distincting the stream before the join with the referenced table could be a non-effective choice, if this requires introducing a sort in the plan, and there are not many frequent values for the foreign key. However, if the stream is already sorted on those keys for index maintenance purposes, then it is possible to eliminate the redundant values with little overhead.

5.3 Cascading Actions

Cascading referential integrity constraints allow defining actions other than just verifying correctness when processing a delete or a key update against a referenced table. It is possible to instruct the system to either propagate the change to the foreign table, or set the foreign keys to NULL or their default value. An example is automatically deleting all the orders pertaining to a customer if this customer gets deleted.

The plan to implement cascading actions is similar to one built for referential integrity validation purposes but where the operator that performs the verification is replaced with an update or delete to the foreign table. The plan for this fits the general pattern of deriving a delta table from another, then applying the changes. The update scheme is reused recursively so that the cascaded operation over the foreign table can in turn drive index and indexed view maintenance, constraint validations and further cascaded actions. In general, an update to foreign key columns of a referencing table requires the verification of the constraint, but this is not necessary in case the update is due to a cascading action through this particular foreign key constraint. For example, if propagating a change to the C_CUSTKEY column from the CUSTOMERS to the ORDERS table, it is not necessary to verify that the new key exists in the referenced table, because this is implicitly guaranteed by the operation. It is however necessary to perform the verification when setting the foreign keys to their default, because there is no guarantee that this value exists on the referenced table.

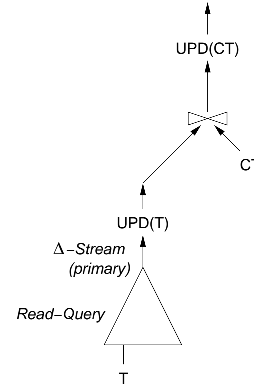

Figure 5: Cascading updates through foreign keys.

Figure 5 shows a plan to cascade a primary key update into the referencing table.

6. CONCLUSION

Efficient execution of update operations is important for customer applications, and depending on their complexity and the volume of data to process they exhibit standard query processing issues. Mapping updates to a relational processing setting allows leveraging existing relational technology and infrastructure in both query optimization and execution. This leads to re-use of components, and provides robustness of implementation and better performance.

Motivated by this, Microsoft SQL Server integrates processing of updates into its query processor framework. In this paper we presented a brief overview of the basic concepts for this integration. We discussed implementation issues and showed query plans that perform table updates, constraint verification, and maintenance of derived physical structures.

7. REFERENCES

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[2] A. Gärtner, A. Kemper, D. Kossmann, and B. Zeller. Efficient Bulk Deletes in Relational Databases. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 183–192, Heidelberg, Germany, April 2001.

[3] G. Graefe. The Microsoft Relational Engine. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 160–161, New Orleans, February 1996.

[4] Transaction Processing Performance Council, San Jose, CA, USA. *TPC Benchmark H (Ad-hoc, Decision Support)*, Revision 1.2.1 1999.