Transaction Reordering and Grouping for Continuous Data Loading

Gang Luo[1], Jeffrey F. Naughton[2], Curt J. Ellmann[2], and Michael W. Watzke[3]

[1] IBM T.J. Watson Research Center
[2] University of Wisconsin-Madison
[3] NCR
luog@us.ibm.com, naughton@cs.wisc.edu,
ellmann@wisc.edu, michael.watzke@ncr.com

Abstract. With the increasing popularity of operational data warehousing, the ability to load data quickly and continuously into an RDBMS is becoming more and more important. However, in the presence of materialized join views, loading data concurrently into multiple base relations of the same materialized join view can cause a severe deadlock problem. To solve this problem, we propose reordering the data to be loaded so that at any time, for any materialized join view, data is only loaded into one of its base relations. Also, for load transactions on the relations that contain "aggregate" attributes, we propose using pre-aggregation to reduce the number of SQL statements in the load transactions. The advantages of our methods are demonstrated through experiments with a commercial parallel RDBMS.

1 Introduction

Today's business market is becoming more and more versatile and competitive. To become and remain successful, an enterprise has to make real-time decisions about its day-to-day operations in response to the fast changes happening all the time in the world [6]. As a result, enterprises are starting to use operational data warehouses to provide fresher data and faster queries [4]. In an operational data warehouse, the stored information is updated in real time or close to it. Also, materialized views are used to speed query processing. This poses some technical challenges. In this paper, we consider a challenge that arises in the context of continuous data loading in the presence of materialized views.

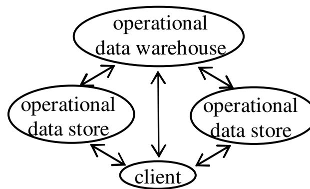

Fig. 1. Operational data warehouse architecture

C. Bussler et al. (Eds.): BIRTE 2006, LNCS 4365, pp. 34–49, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Figure 1 shows the architecture of a typical operational data warehouse [4] (Wal-Mart's data warehouse uses this architecture [22]). Clients store new data into operational data stores in real time, where an operational data store is an OLTP database, a message queue [3], or anything else that is suitable for an OLTP workload. The purpose of these operational data stores is to acknowledge the clients' input immediately while ensuring the durability of this data. As quickly as feasible, this new data is transferred by continuous load utilities from operational data stores into a centralized operational data warehouse, where it is typically managed by an RDBMS. Then clients can query this operational data warehouse, which is the only place that global information is available.

Note: The continuous load utilities are not used for arbitrary applications. Rather, they are used to synchronize the centralized operational data warehouse with the operational data stores. As a result, the state-of-the-art commercial continuous load utilities (e.g., Oracle [16], Teradata [20]) have certain characteristics that are not valid in some applications. We will describe these characteristics in Section 2.1 below.

For performance reasons, existing continuous load utilities often load data into the RDBMS through multiple concurrent sessions. In the presence of materialized join views, a deadlock problem can occur during immediate materialized join view maintenance. This is because a materialized join view *JV* links together multiple base relations. When a base relation used in the definition of *JV* is updated, in order to maintain *JV*, all other base relations in its definition must be read. Hence, transactions updating different base relations in the definition of *JV* can deadlock due to their lock requests on these base relations.

A simple solution to the above deadlock problem is to do materialized join view maintenance in a deferred manner rather than immediately. That is, an update is inserted into the base relation as soon as possible; but the materialized join views that refer to that base relation only see the update at some later time, when the materialized join views are updated in a batch operation. Unfortunately, this makes the materialized join views at least temporarily inconsistent with the base relations. The resulting semantic uncertainty may not be acceptable to all applications. This observation has been made elsewhere. For example, [11] emphasizes that consistency is important for materialized views that are used to make real-time decisions. As another example, in the TPC-R benchmark, maintaining materialized views immediately with transactional consistency is a mandatory requirement [18], presumably as a reflection of some real world application demands. As a third example, as argued in [11], materialized views are like indexes. Since indexes are always maintained immediately, immediate materialized view maintenance should also be desirable in many cases.

The reader might wonder whether using a multi-version concurrency control method can solve the above deadlock problem. In general, a multi-version concurrency control method can avoid conflicts between a pure read transaction and a write transaction (or a transaction that does both reads and writes) [2, 11]. However, in our case, the immediate materialized join view maintenance transactions do both reads and writes. As a result, a multi-version concurrency control method cannot avoid the conflicts between these transactions [2, 11]. In fact, [11] proposed a multi-version concurrency control method to avoid conflicts between pure read transactions on materialized join views and immediate materialized join view maintenance

transactions. For this reason, in this paper, we do not discuss pure read transactions on materialized join views.

To solve the deadlock problem without sacrificing consistency between the materialized join views and the base relations, we propose reordering the data to be loaded so that at any time, for any materialized join view *JV*, data is only loaded into one of its base relations. (As we describe in Section 2.3, in the context of continuous load operations, standard partitioning techniques can be used to guarantee that there are no deadlocks among transactions updating the same base relation. Also, as we describe in Sections 2.1 and 2.3, reordering is allowed in the state-of-the-art continuous load utilities.)

Reordering transactions may cause slight delays in the processing of load transactions that have been moved later in the load schedule. On balance, these delays will be offset by the corresponding transactions that were moved earlier in the schedule to take the place of these delayed transactions. For some applications, this reordering is preferable to the inconsistencies that result from deferred materialized view maintenance. These are the target applications for our reordering technique.

Reordering transactions is not a new idea. For example, [15] proposed reordering queries to improve the buffer pool hit ratio. Also, in practice, some data warehouse users reorder transactions themselves in their applications to avoid contention among the transactions [5]. However, to our knowledge the published literature has not considered an automatic, general purpose transaction reordering method that attempts to reduce deadlocks in continuous data loading applications.

In addition to reordering transactions, we propose a second method to improve the efficiency of continuous data loading. For relations with attributes representing aggregate information (e.g., quantity, amount), we use pre-aggregation to reduce the number of SQL statements in the load transactions. In our experiments, we observed that pre-aggregation can greatly increase the continuous data loading speed.

Of course, the techniques proposed in this paper do not solve all the problems encountered in continuous data loading in the presence of materialized views. Other problems exist, e.g., concurrency control conflicts on materialized views [11, 12], excessive resource usage during materialized view maintenance [13]. However, we believe our techniques form one part of the solution that is required for continuous data loading in the presence of materialized views.

The rest of this paper is organized as follows. In Section 2, we provide some background for continuous data loading. In Section 3, we explore the deadlock problem with existing continuous load utilities in the presence of materialized join views, and show how this problem can be avoided using the reordering method. In Section 4, we explore the use of pre-aggregation to reduce the number of SQL statements in the load transactions. Section 5 investigates the performance of our method through an evaluation in a commercial parallel RDBMS. We conclude in Section 6.

2 Continuous Data Loading

Since loading data into a database is a general requirement of database applications, most commercial RDBMS vendors provide load utilities, each of which have

(somewhat) different functionality. Some are continuous load utilities, while others only support batch bulk load. The functionality of certain load utilities can be implemented by applications. However, since a large number of applications need such functionality, RDBMS vendors typically provide this functionality as a package for application developers to use directly. In the rest of this paper, we do not differentiate between the load utilities provided by the RDBMS vendors and the applications written by the application developers that provide data loading functionality. We refer to both of them as load utilities, and our discussion holds for both.

In this section, we describe how existing continuous load utilities typically work (minor differences in implementation details will not influence our general discussion).

2.1 Workload Specification

Figure 2 shows a typical architecture for loading data continuously into an RDBMS [19, 20]. Data comes from multiple data sources (files, OLTP databases, message queues, pipes, etc.) in the form of modification operations (insert, delete, or update). Then a continuous load utility loads the data into the RDBMS using update transactions. Each update transaction contains one or more modification operations.

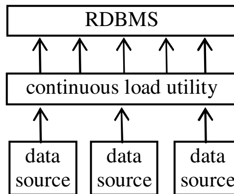

Fig. 2. Continuous data loading architecture

As is the case in data stream applications, the system has no control over the order in which modification operations arrive [1]. To decide which transformations are valid on the stream of load transactions, we discuss the semantics of continuous data loading. The state-of-the-art two popular commercial continuous load utilities (Oracle [16], Teradata [20]) make the following assumptions for continuous data loading:

(a) The RDBMS is running with standard ACID properties for transactions. The continuous load utility looks to the RDBMS like a series of transactions, each containing a single modification operation (insert, delete, or update) on a single relation. Hence, load transactions submitted by continuous load utilities will not cause inconsistency for transactions submitted by other applications.
(b) The RDBMS neither imposes nor assumes any particular order for these load transactions – indeed, their order is determined by the (potentially multiple) external systems "feeding" the load process. Hence, the load process is free to arbitrarily reorder these transactions.
(c) The RDBMS has no requirement on whether multiple modification operations can or cannot commit/abort together. Hence, for efficiency purposes, the load process is free to arbitrarily group these single-modification-operation transactions.

In this paper, we make the same assumptions. Hence, in our techniques, we can do reordering and grouping arbitrarily.

The alert reader may notice that arbitrary reordering can cause certain anomalies. For example, such an anomaly arises if the deletion of a tuple *t* is moved before the updating of tuple *t*. In practice, some applications tolerate such anomalies [5]. In other cases, the application ensures that the order in which modification operations arrive at the continuous load utility will not allow such anomalies [5]. For example, before the continuous load utility acknowledges the completion of updating tuple *t*, the operation of deleting tuple *t* is not submitted to the continuous load utility. In either case, the continuous load utility does not need to worry about these anomalies.

In this paper, we make the further assumption that some locking mechanism is used to provide concurrency control. More specifically, we assume that:

(a) The system uses strict two-phase locking.
(b) The system uses tuple-level locks. The extension to multiple-granularity locking [9] is straightforward.

To increase concurrency, a continuous load utility typically opens multiple sessions to the RDBMS (at any time, each session can have at most one running transaction [10, page 320]). These sessions are usually maintained for a long time so that they do not need to be re-established for each use. For efficiency, within a transaction, all the SQL statements corresponding to modification operations are usually pre-compiled into a stored procedure whose execution plan is stored in the RDBMS. This not only reduces the network overhead (transmitting a stored procedure requires a much smaller message than transmitting multiple SQL statements) but also eliminates the overhead of repeatedly parsing and optimizing SQL statements.

2.2 Grouping Modification Operations

Continuous load utilities usually combine multiple modification operations into a single transaction rather than applying each modification operation in a separate transaction [19, 20]. This is because of the per transaction overhead. Using a large transaction can amortize this overhead over multiple modification operations. In the rest of this paper, we refer to the number of modification operations that are combined into a single transaction as the *grouping factor*.

2.3 The Partitioning Method

As mentioned in Section 2.1, to increase concurrency, a continuous load utility typically opens multiple sessions to the RDBMS. In this section, we review the standard approach used to avoid deadlock in continuous load operations in the absence of materialized views.

Suppose the continuous load utility opens $k \geq 2$ sessions S_i ($1 \leq i \leq k$) to the RDBMS. If we randomly distribute the modification operations among the *k* sessions, transactions from different sessions can easily deadlock on X lock requests on the base relations. This is because these transactions may modify the same tuples concurrently [20]. A simple solution to this deadlock problem is to partition

(e.g., hash on some attribute) the tuples among different sessions so that modification operations on the same tuple are always sent through the same session [20]. In this way, the deadlock condition (transactions from different sessions modify the same tuple) no longer exists and deadlocks will not occur. (Note: the partitioning method may change the order that the tuples arrive at the RDBMS. However, as mentioned in Section 2.1, such reordering is allowed in existing continuous load utilities.)

3 The Reordering Method

In this section, we consider the general case in which materialized views are maintained in the RDBMS, and show that in this case the partitioning method of Section 2.3 is not sufficient to avoid deadlocks. We focus on an important class of materialized views called *join views*. In an extended relational algebra, by a join view *JV*, we mean either an ordinary join view $\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_h))$ or an aggregate join view $\gamma(\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_h)))$, where γ is an aggregate operator. SQL allows the aggregate operators *COUNT*, *SUM*, *AVG*, *MIN*, and *MAX*. However, because *MIN* and *MAX* cannot be maintained incrementally (the problem is deletes [8]), we restrict our attention to the three aggregate operators that make the most sense for materialized aggregates: *COUNT*, *SUM*, and *AVG*.

3.1 Impact of Immediate Materialized View Maintenance

In continuous data loading, we allow data to be loaded into multiple base relations concurrently. This is necessary if we want to keep the data in the RDBMS as up-to-date as possible. However, if a join view is defined on multiple base relations, deadlocks are likely to occur. This is because a join view *JV* links different base relations. When a base relation of *JV* is updated, to maintain *JV*, all the other base relations in the definition of *JV* are read. That is, the introduction of the join view changes the update transactions into update-read transactions. These reads can conflict with concurrent writes to the other base relations of *JV*. For example, consider the following two base relations: $A(a, c)$ and $B(d, e)$. Suppose a join view $JV = A \bowtie B$ is defined on *A* and *B*, where the join condition is $A.c=B.d$. Consider the following two modification operations:

(1) O_1: Modify a tuple t_1 in base relation *A* whose $c=v$.
(2) O_2: Modify a tuple t_2 in base relation *B* whose $d=v$.

These modification operations require the following tuple-level locks on base relations *A* and *B*:

O_1: (L_{11}) A tuple-level X lock on *A* for tuple t_1.
(L_{12}) Several tuple-level S locks on *B* for all the tuples in *B* whose $d=v$ (for join view maintenance purpose).

O_2: (L_{21}) A tuple-level X lock on *B* for tuple t_2.
(L_{22}) Several tuple-level S locks on *A* for all the tuples in *A* whose $c=v$.

Suppose operation O_1 is executed by transaction T_1 through session S_1, while operation O_2 is executed by transaction T_2 through session S_2. If transactions T_1 and T_2 request the locks in the order

Step 1: T_1 requests L_{11}. **Step 2:** T_2 requests L_{21}.
Step 3: T_1 requests L_{12}. **Step 4:** T_2 requests L_{22}.

a deadlock occurs. This is because L_{11} (L_{22}) contains a tuple-level X (S) lock on A for tuple t_1. Also, L_{21} (L_{12}) contains a tuple-level X (S) lock on B for tuple t_2.

Allowing dirty reads is a standard technique to improve the concurrency of read-only queries. Since materialized join view maintenance has at its heart a join query, it is natural to wonder if dirty reads can be used here. Unfortunately, in the context of materialized view maintenance, allowing dirty reads is problematic. This is because using dirty reads to maintain join views makes the results of these dirty reads permanent in the join views [21]. Thus, although dirty reads would avoid the deadlock problem, they cannot be used.

It is also natural to question whether some extension of the partitioning method described in Section 2.3 can be used to avoid deadlocks in the presence of materialized join views. In certain cases, the answer is yes. For example, suppose we use the same partitioning function to partition the tuples of A and B among different sessions according to the join attributes $A.c$ and $B.d$, respectively. Then for immediate materialized view maintenance, the deadlock problem will not occur. This is because in this case, “conflicting” transactions are always submitted through the same session. Also, at any time, one session can have at most one running transaction [10, page 320]. Unfortunately, in practice, such an appropriate partitioning method is not always possible:

(1) In continuous data loading, modification operations on a base relation R usually specify some (e.g., the primary key) but not all attribute values of R [20]. We can only partition the tuples of base relation R among different sessions according to (some of) those attributes whose values are specified by the modification operations on R. This is because we use the same attributes to partition the modification operations on base relation R among different sessions. Suppose that base relation R is a base relation of a join view. Also, suppose the join attribute of R is not one of those attributes whose values are specified by the modification operations on R. Then we cannot partition the tuples of base relation R among different sessions according to the join attribute of R.
(2) If multiple join views with different join attributes are defined on the same base relation R, then it is impossible to partition the tuples of base relation R among different sessions according to these join attributes simultaneously.
(3) If within the same join view (e.g., $JV=A \bowtie R \bowtie B$), a base relation R is joined with multiple other base relations (e.g., A and B) on different join attributes, then it is impossible to partition the tuples of base relation R among different sessions according to these join attributes simultaneously.

3.2 Solution with Reordering

The deadlock problem occurs because we allow data to be concurrently loaded into multiple base relations of the same join view. Hence, a natural question is if this were

not allowed, would the deadlock problem still occur? Luckily, the answer is "no" if we set the following rules:

(1) **Rule 1:** At any time, for any join view *JV*, data can only be loaded into one base relation of *JV*.
(2) **Rule 2:** Modification operations (insert, delete, update) on the same base relation use the partitioning method discussed in Section 2.3.
(3) **Rule 3:** The system uses a high concurrency locking protocol (e.g., the V locking protocol [12], or the locking protocol in [11]) on join views so that lock conflicts on the join views can be avoided.

The reason is as follows.

(1) Using rules 1 and 2, all deadlocks resulting from lock conflicts on the base relations are avoided.
(2) Using rule 3, all deadlocks resulting from lock conflicts on the join views can be avoided (e.g., in the V locking protocol [12], V locks are compatible with themselves; in the locking protocol in [11], E locks are compatible with themselves).

Since all possible deadlock conditions are eliminated, deadlocks no longer occur.

We now consider how to implement rules 1-3. It is easy to enforce rules 2 and 3. To enforce rule 1, we can use the following reordering method to reorder the modification operations. Recall in Section 2.1, the semantics of the workload allows us to reorder modification operations arbitrarily. Consider a database with d base relations R_1, R_2, …, and R_d and e join views JV_1, JV_2, …, and JV_e. We keep an array J that contains d elements J_i ($1 \le i \le d$). For each i ($1 \le i \le d$), J_i records the number of transactions that modify base relation R_i and are currently being executed. Each J_i ($1 \le i \le d$) is initialized to zero. For each m ($1 \le m \le k$), we maintain a queue Q_m recording transactions waiting to be run through session S_m. Each Q_m ($1 \le m \le k$) is initialized to empty. During grouping (see Section 2.2), we only combine modification operations on the same base relation into a single transaction.

If base relations R_i and R_j ($1 \le i, j \le d$, $i \ne j$) are base relations of the same join view, we say that R_i and R_j conflict with each other. Two transactions modifying conflicting base relations are said to conflict with each other. We call transaction T a "desirable transaction" if it does not conflict with any currently running transaction. Consider a particular base relation R_i ($1 \le i \le d$). Suppose R_{s_1}, R_{s_2}, …, and R_{s_w} ($w \ge 0$) are all the other base relations that conflict with base relation R_i. At any time, if either $w=0$ or all the $J_{s_u} = 0$ ($1 \le u \le w$), then a transaction T modifying base relation R_i ($1 \le i \le d$) is a desirable transaction.

We schedule transactions as follows:

(1) **Action 1:** For each session S_m ($1 \le m \le k$), as discussed in Section 2.2, whenever the continuous load utility has collected n modification operations on a base relation R_i ($1 \le i \le d$), we combine these operations into a single transaction T and insert transaction T to the end of Q_m. Here, n is the pre-defined grouping factor that is specified by the user who sets up the continuous load utility. If session S_m

is free, we try to schedule a transaction to the RDBMS for execution through session S_m.

(2) **Action 2:** When some transaction T modifying base relation R_i ($1 \le i \le d$) finishes execution and frees session S_m ($1 \le m \le k$), we do the following:
   (a) We decrement J_i by one.
   (b) If Q_m is not empty, we schedule a transaction to the RDBMS for execution through session S_m.
   (c) Suppose J_i is decremented to zero (so that some waiting transaction possibly becomes desirable). For each g ($1 \le g \le k$, $g \ne m$), if session S_g is free and Q_g is not empty, we try to schedule a transaction to the RDBMS for execution through session S_g.

(3) **Action 3:** Whenever we try to schedule a transaction to the RDBMS for execution through session S_m ($1 \le m \le k$), we do the following:
   (a) We search Q_m sequentially until either a desirable transaction T is found or all the transactions in Q_m have been scanned, whichever comes first.
   (b) In the case that a desirable transaction T modifying base relation R_i ($1 \le i \le d$) is found, we increment J_i by one and send transaction T to the RDBMS for execution.

The above discussion does not address starvation. There are several starvation prevention techniques that can be integrated into the transaction reordering method. We list one of them as follows. The idea is to use a special header transaction to prevent the first transaction in any Q_g from starvation ($1 \le g \le k$). We keep a pointer r whose value is always between 0 and k. r is initialized to 0. If every Q_m ($1 \le m \le k$) is empty, $r=0$. At any time, if $r=0$ and a transaction is inserted into some Q_m ($1 \le m \le k$), we set $r=m$. If $r=m$ ($1 \le m \le k$) and the first transaction of Q_m leaves Q_m for execution, r is incremented by one (if $m=k$, we set $r=1$). If Q_r is empty, we keep incrementing r until either Q_r is not empty or we discover that every Q_m ($1 \le m \le k$) is empty. In the later case, we set $r=0$. We make use of a pre-defined timestamp *TS* determined by application requirements. If pointer r has stayed at some v ($1 \le v \le k$) longer than *TS*, the first transaction of Q_v becomes the header transaction. Whenever we are searching for a desirable transaction in some Q_m ($1 \le m \le k$) and we find transaction T, if the header transaction exists, we ensure that either T is the header transaction or T does not conflict with the header transaction. Otherwise transaction T is still not desirable and we continue the search.

4 The Pre-aggregation Method

A large number of data warehouses have relations with certain attributes representing aggregate information (e.g., *quantity* or *amount*). In many cases, updates to these relations increment or decrement the aggregate attribute values [7]. As discussed in Section 2.2, when we load data continuously into these relations, we combine multiple modification operations into a single load transaction. This creates an opportunity for optimization: by pre-aggregation, we can reduce the number of SQL statements in the load transactions on these relations.

For example, consider a relation *R* in the database whose *R.b* attribute represents aggregate information. Suppose the following two modification operations O_1 and O_2 are combined into a single load transaction *T*:

(1) O_1: update R set R.b=R.b+b_1 where R.a=v;
(2) O_2: update R set R.b=R.b+b_2 where R.a=v;

If we let $b_3=b_1+b_2$, then transaction *T* can be transformed into an equivalent transaction T' that contains only a single SQL statement:

update R set R.b=R.b+b_3 where R.a=v;

Compared to transaction *T*, transaction T' saves one SQL statement. Hence, transaction T' is more efficient. The reason is that executing a SQL statement is much more expensive than aggregating the two values b_1 and b_2 into a single value b_3.

4.1 Algorithm Description

We call the above method the pre-aggregation method. The general pre-aggregation method works in the following way. Consider a base relation *R* with one or multiple "aggregate" attributes. Assume that in the grouping method discussed in Section 2.2, all modification operations combined in a single transaction are on the same base relation. For each load transaction on relation *R*, we do the following operations:

(1) Find all the modification operations that increment/decrement the "aggregate" attribute values. Move these modification operations to the beginning of the transaction (i.e., ahead of all the other modification operations). Suppose each such modification operation can be represented as a pair $<a, b>$, where *a* denotes the tuple (set of tuples) to be modified, and *b* denotes the amount that will be added to (or subtracted from) the "aggregate" attribute value(s) of the tuple(s).
(2) Sort these modification operations so that modification operations on the same tuple (set of tuples) are adjacent to each other.
(3) Among these modification operations, combine multiple adjacent modification operations $<a, b_1>$, $<a, b_2>$, …, and $<a, b_m>$ on the same tuple (set of tuples) into a single modification operation $<a, c>$, where $c=b_1+b_2+\ldots+b_m$. In the extreme case that $c=0$, the single modification operation $<a, c>$ can be omitted.

The above procedure can be easily extended to handle the UPSERT/MERGE SQL statement [17].

The pre-aggregation method has the following advantages:

(1) The processing load of the database engine is reduced.
(2) The transaction execution/response time is reduced. This may further improve database concurrency, as the period that transactions hold locks is reduced.

These advantages come from the fact that pre-aggregation outside of the database engine followed by executing fewer SQL statements inside the database engine is more efficient than executing all the SQL statements inside the database engine.

5 Performance Evaluation

In this section, we describe experiments that were performed on the commercial IBM DB2 parallel RDBMS. Our measurements were performed with the database client application and server running on an Intel x86 Family 6 Model 5 Stepping 3 workstation with four 400MHz processors, 1GB main memory, six 8GB disks, and running the Microsoft Windows 2000 operating system. We allocated a processor and a disk for each data server, so there were at most four data servers on each workstation.

5.1 Experiment Description

The relations used for the tests model a real world scenario. Customers interact with a retailer via phone/web to make a purchase. The purchase involves browsing available merchandise items and possibly selecting an item to purchase. The following events occur:

(1) Customer indicates desire for a specific item and event is recorded in the *demand* relation.
(2) The *inventory* relation is checked for item availability.
(3) If the desired item is on hand, a customer order is placed and the *inventory* relation is updated; otherwise a vendor order is placed.

The schemas of the *demand* and *inventory* relations are listed as follows:

demand (partkey, date, quantity, <u>custkey</u>, comment),
inventory (<u>partkey</u>, <u>date</u>, quantity, extended_cost, extended_price).

The underscore indicates the partitioning attributes. For each relation, we built an index on the partitioning attribute(s). In our tests, each *inventory* tuple matches 4 *demand* tuples on the attributes *partkey* and *date*. Also, different *demand* tuples have different *custkey* values. In practice, there can be a large number of different parts. However, for any given day, most transactions only focus on a small portion of them (the "active" parts). In our testing, we assume that *s* parts are active today. We only consider today's transactions that are related to these active parts. We believe that our conclusion would remain much the same if all transactions related to both active and inactive parts were considered. This is because in this case, the number of deadlocks caused by the transactions that are related to the active parts would remain much the same.

Table 1. Test data set

	number of tuples	total size
demand	8M	910MB
inventory	2M	77MB

Suppose that the *demand* and *inventory* relations are frequently queried for sales forecasting, lost sales analysis, and assortment planning applications, so a join view *onhand_demand* is built as the join result of *demand* and *inventory* on the join attributes *partkey* and *date*:

create join view onhand_demand as select d.partkey, d.date, d.quantity, d.custkey, i.quantity
from demand d, inventory i where d.partkey=i.partkey and d.date=i.date partitioned on d.custkey;

There are two kinds of modification operations that we used for testing, both of which are related to today's activities:

(1) $\boldsymbol{O_1}$**:** Insert one tuple (with today's *date*) into the *demand* relation. This new tuple matches 1 *inventory* tuple on the attributes *partkey* and *date*.
(2) $\boldsymbol{O_2}$**:** Update one tuple in the *inventory* relation with a specific *partkey* value and today's *date*.

We created an auxiliary relation for the *demand* relation that is partitioned on the (*partkey*, *date*) attributes to change expensive all-node join operations for join view maintenance to cheap single-node join operations [13].

We evaluated the performance of the reordering method and the naive method in the following way:

(1) We tested the largest available hardware configuration with four data server nodes.
(2) We executed a stream of modification operations. A fraction p of these modification operations are O_1. The other $1-p$ of the modification operations are O_2. Each O_1 inserts a tuple into the *demand* relation with a random *partkey* value. Each O_2 updates a tuple in the *inventory* relation with a random *partkey* value.
(3) In both the reordering method and the naive method, we only combine modification operations on the same base relation into a single transaction. Each transaction has the same grouping factor n.
(4) In the naive method, if a transaction deadlocked and aborted, we automatically re-executed it until it committed.
(5) We performed a concurrency test. We fixed $p=50\%$ and the number of active parts $s=10,000$. In both the reordering method and the naive method, we tested four cases: $k=2$, $k=4$, $k=8$, and $k=16$, where k is the number of sessions. In each case, we let the grouping factor n vary from 1 to 128.

5.2 Concurrency Test Results

The throughput (number of modification operations per second) is an important performance metric of the continuous load utility. For the naive method, to see how deadlocks influence its performance, we investigated the relationship between the throughput and the deadlock probability.

By definition, when the deadlock probability becomes close to 1, almost every transaction will deadlock. Deadlock has the following negative influences on throughout:

(1) Deadlock detection/resolution is a time-consuming process. During this period, the deadlocked transactions cannot make any progress.
(2) The deadlocked transactions will be aborted and re-executed. During re-execution, these transactions may deadlock again. This wastes system resources.

Hence, once the system starts to deadlock, the deadlock problem tends to become worse and worse. Eventually, the throughput of the naive method deteriorates significantly.

We show the throughput of the naive method in Figure 3. For a given number of sessions *k*, when the grouping factor *n* is small, the throughput of the naive method keeps increasing with *n*. This is because executing a large transaction is more efficient than executing a large number of small transactions, as discussed in Section 2.2. (In our testing, the performance advantages of having a large grouping factor *n* are not very large. This is mainly due to the fact that due to software restrictions, we could only run the database client application and server on the same computer. In this case, the overhead per transaction is fairly low. Amortizing such a small overhead with a large *n* cannot bring much benefit.) When *n* becomes large enough, if the naive method does not run into the deadlock problem, the throughput of the naive method approaches a constant, where the system resources become fully utilized. The larger *k*:

(1) the higher concurrency in the RDBMS and the larger the constant.
(2) the easier it becomes to achieve full utilization of system resources and the smaller *n* is needed for the throughput to achieve that constant.

When *n* becomes too large, the naive method runs into the deadlock problem. The larger *k*, the smaller *n* is needed for the naive method to run into the deadlock problem. Once the deadlock problem occurs, the throughput of the naive method deteriorates significantly. Actually, it decreases as *n* increases. This is because the larger *n*, the more transactions are aborted and re-executed due to deadlock.

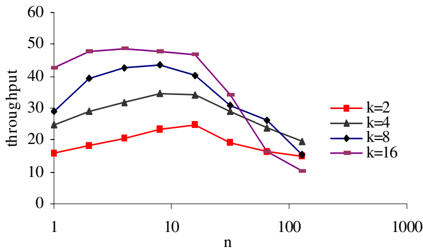

Fig. 3. Throughput of the naive method (concurrency test)

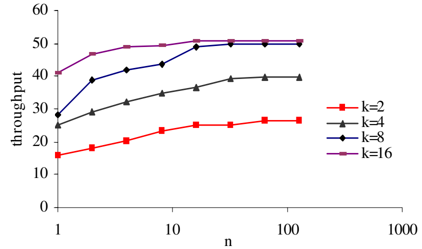

Fig. 4. Throughput of the reordering method (concurrency test)

For a given *n*, before the deadlock problem occurs, the throughput of the naive method increases with *k*. This is because the larger *k*, the higher concurrency in the RDBMS. However, when *n* is large enough (e.g., *n=128*) and the naive method runs into the deadlock problem, due to the extreme overhead of repeated transaction abortion and re-execution, the throughput of the naive method may decrease as *k* increases.

We show the throughput of the reordering method in Figure 4. The general trend of the throughput of the reordering method is similar to that of the naive method (before the deadlock problem occurs). That is, the throughput of the reordering method increases with both *n* and *k*. For a given *k*, as *n* becomes large, the throughput of the reordering method approaches a constant. However, the reordering method never deadlocks. For a given *k*, the throughput of the reordering method keeps approaching that constant no matter how large *n* is. Once the naive method runs into the deadlock

problem, the reordering method exhibits great performance advantages over the naive method, as the throughput of the naive method in this case deteriorates significantly.

In both the *k=8* case and the *k=16* case, when *n* becomes large enough, the throughput of the reordering method approaches (almost) the same constant. This is because in these two cases, all data server nodes (e.g., disk I/Os) become fully utilized. In our testing, if we had a larger hardware configuration with more data server nodes, the constant for the *k=16* case would be larger than that for the *k=8* case.

We show the ratio of the throughput of the reordering method to that of the naive method in Figure 5. Before the naive method runs into the deadlock problem, the throughput of the reordering method is smaller than that of the naive method. This is because the reordering method has some overhead in performing reordering and synchronization (i.e., switching from executing one type of transactions (say, transactions updateing the *inventory* relation) to executing another type of transactions (say, transactions updating the *demand* relation)). However, such overhead is not significant. In our tests, the throughput of the reordering method is never lower than 96% of that of the naive method.

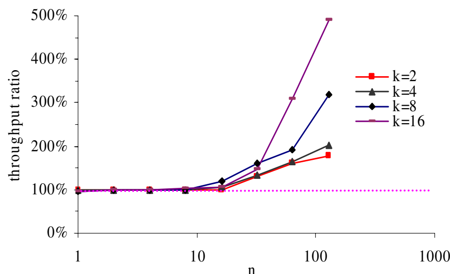

Fig. 5. Throughput improvement gained by the reordering method (concurrency test)

When the naive method runs into the deadlock problem, the throughput of the reordering method does not drop while the throughput of the naive method is significantly worse. In this case, the ratio of the throughput of the reordering method to that of the naive method is greater than 1. For example, when *n=32*, for any *k*, this ratio is at least 1.3. When *n=64*, for any *k*, this ratio is at least 1.6. In the extreme case when *k=16* and *n=128*, this ratio is 4.9. In general, when the naive method runs into the deadlock problem, this ratio increases with both *k* and *n*. This is because the larger *k* or *n*, the easier the transactions deadlock in the naive method. The extreme overhead of repeated transaction abortion and re-execution exceeds the benefit of the higher concurrency (efficiency) brought by a larger *k* (*n*). However, there are two exceptions. When *n=16* or *n=32*, the ratio curve for *k=16* is below the ratio curve for *k=8*. This is because in these two cases, for the reordering method, all data server nodes (e.g., disk I/Os) become fully utilized and the throughput is almost independent of both *k* and *n*. By comparison, in the naive method, as there are not enough transaction aborts, the throughput for the *k=16* case is higher than that for the *k=8* case.

Due to space constraints, we put the performance study of the pre-aggregation method into the full version of this paper [14].

6 Conclusion

This paper proposes two techniques to improve the efficiency of existing continuous load utilities:

(1) In the presence of join views in an RDBMS, we reorder the data load operations to avoid deadlocks.
(2) We use pre-aggregation to reduce the number of SQL statements in the load transactions.

Our experiments with a commercial system are promising, showing that these two techniques can significantly improve throughput for certain workloads.

Acknowledgements

We would like to thank Henry F. Korth for helpful discussions.

References

[1] Babcock, B., Babu, S., Datar, M., et al.: Models and Issues in Data Stream Systems. PODS 2002, pp. 1–16 (2002)
[2] Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
[3] Bernstein, P.A., Hsu, M., Mann, B.: Implementing Recoverable Requests Using Queues. SIGMOD Conf. 1990, pp. 112–122 (1990)
[4] Brobst, S., Rarey, J.: The Five Stages of an Active Data Warehouse Evolution (2001) http://www.ncr.com/online_periodicals/brobst.pdf
[5] Brobst, S.: Personal communication (2003)
[6] Dver, A.: Real-time Enterprise. Business week December 2, 2002 issue (2002)
[7] Gawlick, D.: Processing “Hot Spots” in High Performance Systems. In: Proc. IEEE Compcon Spring ’85 (1985)
[8] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams. SIGMOD Conf. 2001, pp. 13–24 (2001)
[9] Gray, J., Lorie, R.A., Putzolu, G.R., et al.: Granularity of Locks and Degrees of Consistency in a Shared Data Base. IFIP Working Conference on Modeling in Data Base Management Systems 1976, pp. 365–394 (1976)
[10] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, San Francisco (1993)
[11] Graefe, G., Zwilling, M.J.: Transaction Support for Indexed Views. SIGMOD Conf.2004 (2004)
[12] Luo, G., Naughton, J.F., Ellmann, C.J., et al.: Locking Protocols for Materialized Aggregate Join Views. VLDB 2003, pp. 596–607 (2003)
[13] Luo, G., Naughton, J.F., Ellmann, C.J., et al.: A Comparison of Three Methods for Join View Maintenance in Parallel RDBMS. ICDE 2003, pp. 177–188 (2003)
[14] Luo, G., Naughton, J.F., Ellmann, C.J., et al.: Transaction Reordering and Grouping for Continuous Data Loading. Full version (2006) available at http://www.cs.wisc.edu/~gangluo/tpump_full.pdf
[15] O’Gorman, K., Abbadi, A.E., Agrawal, D.: Multiple Query Optimization by Cache-Aware Middleware using Query Teamwork. ICDE 2002, p. 274 (2002)
[16] Oracle Streams (2002) http://otn.oracle.com/products/dataint/htdocs/streams_fo.html
[17] Oracle9i Database Daily Feature - MERGE Statement (2002) http://technet.oracle.com/products/oracle9i/daily/Aug24.html

[18] Poess, M., Floyd, C.: New TPC Benchmarks for Decision Support and Web Commerce. SIGMOD Record 29(4), 64–71 (2000)
[19] Pooloth, K.: High Performance Inserts on DB2 UDB EEE using Java (2002) http://www7b.boulder.ibm.com/dmdd/library/techarticle/0204pooloth/0204pooloth.html#overview
[20] Teradata Parallel Data Pump Reference (2002) http://www.info.ncr.com/eDownload.cfm?itemid=023390001
[21] Zhuge, Y., Garcia-Molina, H., Wiener, J.L.: The Strobe Algorithms for Multi-Source Warehouse Consistency. PDIS 1996, pp. 146–157 (1996)
[22] Zimmerman, E. Nelson. In Hour of Peril, Americans Moved to Stock up on Guns and TV Sets. Wall Street Journal Newsletter (September 18, 2001) http://www.swcollege.com/econ/ street/html/sept01/sept18_2001.html