Concurrency in the Data Warehouse

Dr. Richard Taylor

Informix Software Inc.
485 Alberto Way
Los Gatos
USA
richard.taylor@informix.com

Abstract

When a data warehouse is loaded at night and queried during the day, there is no requirement for concurrent update and querying. However there are a number of situations where concurrency is needed: trickle feed applications, correcting exception data from the nightly load, the narrowing load window. The end point of the narrowing load window is a data warehouse that is available 7x24. Query Priority Concurrency is the concurrency mechanism implemented by the Informix Red Brick Decision Server. It is called Query Priority Concurrency because it uses versioning to achieve the goal that query performance is unaffected by concurrent loads. The paper discusses the differing requirements for concurrency in a data warehouse, explains why versioning is appropriate, gives a sketch of the implementation and discusses the 6 lock modes that are needed to achieve concurrency and serialised execution. Finally, the frozen query feature is described. This allows users to query the current published version of the data warehouse while the administrators go through all the steps of loading and verifying new data to create the next issue of the warehouse for publication.

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

1. Introduction

The common data warehouse cycle is to load the data warehouse at night with a day's worth of transactions, and to query the data warehouse during the day. In this regime, there is no need for concurrent querying and update. However, the increasing demands on data warehousing creates situations where concurrency is needed.

One situation is a trickle feed application that has a small amount of critical data that needs to be continuously loaded during the day. Trickle feed is commonly found in financial applications where stock prices or currency exchange rates that change during the day are loaded as they change. Another situation where concurrency is convenient is to allow corrections and exception data from the nightly load to be reloaded during the day. Also, it may be convenient to update dimensional data as soon as a new version of the customer or product master file becomes available.

Companies spread across many time zones, stores and offices that stay open late, increased business volume and other similar causes are causing the nightly load window to narrow. The narrowing load window means that the data warehouse may not be completely loaded by the time it is needed for querying. At the end point of the narrowing load window are the global companies and e-commerce enterprises that never sleep. These companies want their data warehouse to be available 7x24.

2. Concurrency Requirements

The requirements for concurrency in a data warehouse are very different from those of an OLTP system. As most database systems have been designed to support OLTP, they do not match the requirements of a data warehouse.

Consider the transaction. An OLTP transaction typically uses indexes to directly select a small number rows and perhaps update some of them. In a data warehouse a typical query may access multiple tables through several indexes, join the results, hopefully with the aid of a multi-table join index and then perform some aggregation to produce a result. The other type of transaction in a date warehouse is a bulk load, which needs to build several indexes including join indexes and may need to check referential integrity and if necessary automatically generate rows to maintain referential integrity.

In an OLTP system, having a transaction lock individual rows works most of the time, and where there are hot spots that prevents locking from working, special techniques can be used. One special technique is versioning where the unit of data being versioned can be a row or even a data item. In a data warehouse, locking individual rows is not useful for either queries that roam over large amounts of data or for the bulk loader. Versioning is a useful technique for allowing bulk loads to proceed in parallel with queries, but the versioning system has to be able to handle versioned data on a far larger scale than is ever envisioned by the implementers of an OLTP system.

2. Query Priority Concurrency

The Informix Red Brick Decision Server implements concurrency using versioning. This concurrency mechanism is called **Query Priority Concurrency** because it is specifically designed for data warehousing with the goal that query performance is unaffected by concurrent modifications of the database. A query sees a consistent snapshot of the database. This snapshot is called a **revision**. A transaction that modifies the database makes a new revision of the database. Each new revision is assigned a monotonically incrementing number.

The implementation of Query Priority Concurrency is straightforward. When a data block is modified, the new version of the block is written to a special segment called the **version log**. Whenever a block is fetched from disk, the transaction checks an in-memory index called the **version log index** to see whether the block should be fetched from the database or from the version log. There may be multiple versions of a block in the version log associated with different revisions. The version log index ensures that the correct block is selected. Finally, when it is safe to do so, blocks from the version log are merged back into the database by a **vacuum cleaner daemon**.

The transaction manager keeps a table of active revisions that is used to determine when the vacuum cleaner daemon should clean a revision. When a query starts, it is assigned a revision R to read. R is always the latest committed revision, except in the case of the frozen query revision feature, which is described later. When the transaction finishes and the transaction manager determines that no other transaction is accessing R or any previous revision, the vacuum cleaner is alerted that it can start cleaning. The vacuum cleaner cleans all revisions in the version log up to and including the latest active revision. This protocol means that when modifications to the database have completed and the queries that access older revisions finish, the vacuum cleaner daemon can clean out the entire version log.

A transaction that updates the database reads the current revision of the database when it starts. Modified blocks are written to the version log, but the blocks are not assigned a revision number until the transaction commits. This means that many transactions can be modifying different tables the database concurrently and the order in which they can commit is not predetermined.

Another important optimisation is that new blocks are written to the database directly. The only blocks that are written to the version log are blocks that are modifications of existing blocks. Thus in a load of new data, we expect the data blocks to be written to directly to the database while most of the changes to indexes will be modifications of existing blocks, and therefore go to the version log. The flip side of this optimisation is that bulk deletes create new versions of blocks that are completely empty. Users are suggested to use non-versioning deletes when they roll-off data to create space.

3. Locking

As has been discussed, fine granularity locking is not appropriate for data warehousing. The Red Brick server only implements table locks, and these locks are used to ensure that there is only one transaction that is modifying a table at one time. While a transaction is creating a new revision of a table, other transactions can read the previously created revisions of the table. However, there is a potential for problems if these other transactions are also modifying other tables in the database system.

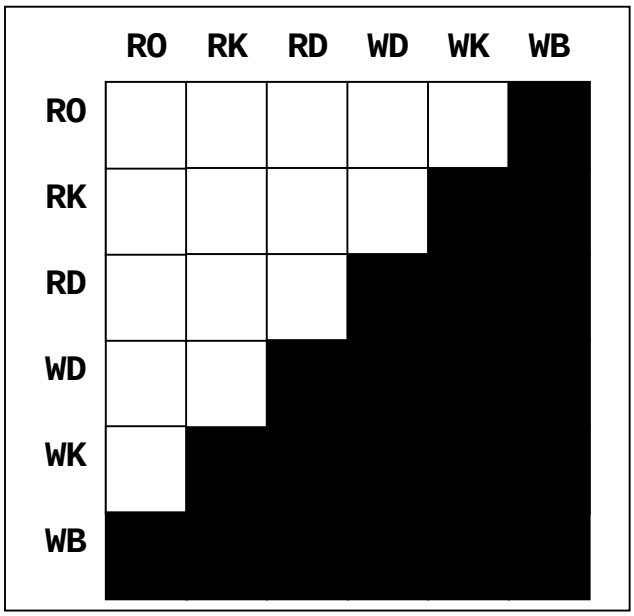
For example, a transaction could be reading a fact table to create an aggregate table, while at the same time, another transaction could be loading new data into the fact table. If this were allowed, when both of these transactions commit, the aggregate table will not reflect the contents of the fact table. Another example is that a transaction could be loading a fact table and checking referential integrity by reading a dimension table while at the same time another transaction could be deleting rows from the dimension table. If this were allowed, when both transactions committed, referential integrity of the fact table would be broken.

To overcome these problems, new lock modes are required. The Red Brick server implements 6 lock modes as shown in Table 1.

Table 1. Lock Modes

RO	Read Only	Normal read lock, used by queries. Compatible with versioned writes.
RK	Read Key	Indicates that the key should not change. Used for referential integrity checking
RD	Read Data	Not compatible with versioned writes, used by a transaction that read existing tables to modify another table.
WD	Write Data	Used by versioning operations that do not change the existing key column in a table: versioned inserts and non-key column updates.
WK	Write Key	Used by versioning operations that change the key column: versioned deletes and updates to key columns.
WB	Write Only	Non-versioned modifications to the table. Not compatible with any other lock.

The lock compatibility matrix is shown below.



The compatibility matrix shows that a RK lock is compatible with a WD lock but not a WK lock. Thus, a transaction can insert rows into a table that is being used for referential integrity checking, but the transaction cannot delete rows from the table.

By default, a transaction that does an INSERT … SELECT … gets a RD lock because it is both reading tables and writing a table. The RD lock is not compatible with any versioning write locks. This behaviour can be overridden by setting the transaction isolation level to repeatable read, in which case an RO lock is used.

The RK lock and its associated WK lock can be thought of as a special case of the RD lock that allows greater freedom while checking referential integrity. In practice this is important in the Red Brick server, which relies heavily on referential integrity for maintaining the coherence of its join indexes and algorithms.

4. Periodic Commit

Query Priority Concurrency provides a simple transaction atomicity mechanism. When a transaction aborts, all that is needed to rollback the transaction is to throw away the blocks in the version log that have been created by the transaction. Similarly, crash recovery is just a matter of restoring the version log metadata and then restarting the vacuum cleaner, which will proceed to clean all the committed blocks in the version log.

Statement atomicity is an essential characteristic of a transaction. However it is sometimes convenient to have sub-statement atomicity. **Periodic Commit** is a feature in the Red Brick loader that implements sub-statement atomicity. The concept is that a load statement can be specified to commit after a number of rows have been loaded, or that the commit occurs after a specified time interval or either depending on which comes first.

The time interval is used to implement trickle feed applications. In a trickle feed application, a low volume stream of data is loaded continuously while the data warehouse is being queried. To implement trickle feed, the periodic commit timer is set to a suitable interval, for example 15 minutes. Then, every 15 minutes, the date loaded since during the interval is committed and immediately becomes visible to any new queries that access the database.

Another use of periodic commit is to save having to restart a bulk load from the beginning when the load aborts. In this case the loader is set to commit after loading a specific number of rows, say for example, every 10 million rows in a 100 million row bulk load. If the load does abort, at most 10 million rows needs to be reloaded, rather than an average of 50 million rows that would have to be reloaded without using periodic commit.

5. Frozen Query Revision

The process for creating a data warehouse involves a number of steps. Data is extracted from operational systems transformed into a suitable form and loaded into the data warehouse. First the dimension tables are loaded and then the fact tables. If there are load problems such as referential integrity failures, these need to be fixed up. Next aggregate tables and their indexes need to be loaded. Finally a few test queries may be run to verify that the data warehouse is complete and consistent. When the data warehouse is ready, it is published to the user community

for querying. Creating the next version of the data warehouse repeats the cycle.

Frozen Query Revision allows the administrator to create the next version of the data warehouse while users are querying the published version. A frozen query revision is created with an alter database statement that specifies the current revision as the frozen revision. Normally with versioning, when a query starts it is assigned to access the current revision of the database. When a frozen revision exists, a query is accesses the frozen revision by default.

After publishing the data warehouse by creating a frozen query revision, the administrator uses versioning operations to create the next version of the database for publication. While the frozen query revision exists, all modifications of the database must be versioned, because non-versioned operation would modify the underlying database and change the frozen revision. The frozen query revision is read only, it cannot be modified.

Modifications to the database must be made in the context of the latest revision. For example, a query that creates an aggregate table must see the latest version of the base table. A flag is set in the administrator session so that transactions in that session access the latest revision. Each versioned operation such as a load creates a new revision when it commits. If the load fails, only that statement is rolled back. When the next version of the data warehouse is ready to be published, the administrator issues an alter database statement to remove the frozen revision, and all new queries access the latest revision which is the newly published revision.

6. Conclusions

Query Priority Concurrency is a versioning mechanism that is designed specifically for data warehousing. For that reason it is different from versioning that has been implemented in other database systems. In the Red Brick server, the unit of versioning is the block. Other versioning schemes version pages, rows or even the individual data item. This aspect of the design matches the data warehouse regime where database modifications are almost always bulk operations.

In the Red Brick server, the new data is written to the log, from where it may be accessed, and the database retains the old version of the data. In other database systems, the new version of the data typically is written to the database and older versions of the data may be accessed from the database, or from an exception file or from the pre-image log. This aspect of the design comes from the requirement that query performance is unaffected by modifications to the database.

Finally, most versioning systems have a tight limit on the number of versions that are kept around. In Red Brick, the default is to allow for 5000 active revisions, and much larger numbers have been used.