Data Versioning Techniques for Internet Transaction Management

Ramkrishna Chatterjee and Gopalan Arun
Oracle Corporation, One Oracle Drive, Nashua, NH 03062, USA
1-603-897-3515
{Ramkrishna.Chatterjee, Gopalan.Arun}@oracle.com

ABSTRACT
An Internet transaction is a transaction that involves communication over the Internet using standard Internet protocols such as HTTPS. Such transactions are widely used in Internet-based applications such as e-commerce. With the growth of the Internet, the volume and complexity of Internet transactions are rapidly increasing. We present data versioning techniques that can reduce the complexity of managing Internet transactions and improve their scalability and reliability. These techniques have been implemented using standard database technology, without any change in database kernel. Our initial empirical results argue for the effectiveness of these techniques in practice.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems - *Transaction processing*

General Terms: Algorithms, Reliability

Keywords: Versioning, Scalability, Internet Transaction

1. INTRODUCTION
A transaction is usually defined as a unit of work that satisfies the ACID properties. However, in an Internet transaction (henceforth ITX), frequently, atomicity and isolation are relaxed, and on transaction failure, compensation is used for the completed steps. Typical ITXs include e-commerce transactions and transactions in Web-enabled e-business applications like CRM systems. In this work, we are interested in any ITX that has one or more of the following characteristics: (1) It involves human interaction, e.g., filling a form. (2) It spans multiple applications or companies. (3) It is implemented using a multi-tiered architecture (e.g., browser → firewall → web server → middle-tier → backend database). (4) It contains many points of possible failure. For example, in an ITX initiated from a PDA, the wireless connection is such a point of possible failure. (5) There are many concurrent transactions. These characteristics define a large set of ITXs, which includes the typical ITXs mentioned earlier.

Due to the above characteristics, many ITXs can last orders of magnitude longer (seconds, minutes, or hours vs. milliseconds) and have many more points of possible failure compared to a traditional SQL-transaction. For example, a user filling a long form may stop in the middle and then continue later. When the user comes back to complete the form, she expects to see the form as it was when she stopped. Hence, the overall ITX of which filling the form is a part may last for a long time, hours or even days. Moreover, until she is done, the user may not want her changes to be seen outside the ITX. As another example, consider an ITX for travel reservation. This ITX may include steps for airline, car and hotel reservations, each of which may be handled by a different company or application. If WS-BusinessActivity protocol [1] is used, for instance, there could be one coordinating activity for the trip, which invokes child activities for airline, car and hotel reservations at participating Internet sites. Since the coordinating activity converses with multiple child activities, it may last longer than a typical SQL-transaction (seconds vs. milliseconds). The child activity in itself may be short, but it may have to wait a while for the complete or cancel signal from the coordinating activity, making the child activity in effect last longer.

These characteristics, mainly longer duration, high volume and many points of possible failure, cause the following ITX implementation problems:

1. Before an ITX ends, if the data modifications done by the ITX are committed in a database for persistence, custom code has to be written for rollback and isolation, which increases the cost and complexity of implementation.

2. If the changes done by an ITX are done in an SQL-transaction that is kept open for the duration of the ITX, issues like rollback and isolation are automatically handled by the SQL-transaction; but the longer duration and the high volume of ITXs can reduce scalability because an open SQL-transaction consumes resources in both the middle-tier and the backend database.

3. If the changes done by a large number of ongoing ITXs are kept in the middle-tier, scalability can be affected as a middle-tier is usually not designed for storing a large amount of data. This also causes problems for session replication needed for load balancing and fail-over. Moreover, if a lot of data is kept in the middle-tier, it may not be possible to efficiently query it (e.g., find all incoming orders for a product). Ideally, the middle-tier should be used only as a cache for ITX data persistently stored in a database.

4. The multiple points of possible failure in an ITX and its longer duration can reduce reliability. For example, if changes done by an ITX are kept in an SQL-transaction or in the middle-tier, a database or middle-tier crash will terminate the ITX.

Although many protocols [1] for managing ITXs have been proposed, these critical ITX implementation problems have not been adequately addressed before.

2. DATA VERSIONING TECHNIQUES
We solve these ITX implementation problems by maintaining multiple virtual databases (henceforth *VDB)* in the same physical database by versioning database rows on demand. Only the changed data is versioned and only one copy of the data common

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-051-5/05/0005.

to multiple VDBs is stored. There is a system-defined *root VDB* into which a VDB is finally merged, and from which new VDBs are created. Data stored in a VDB is not visible outside it until the VDB is merged, and the data is persistent; it lasts across database, middle-tier, and browser crash. Data in a VDB is read and written through SQL-transactions. After an SQL-transaction commits, data modifications done in the SQL-transaction persist in the current VDB. Each ITX (or activity [1] in an ITX) is assigned a VDB in which data modifications done in the ITX are stored until the ITX completes. Intuitively, since a VDB automatically provides functionalities like rollback and isolation, no custom code is needed to achieve these, and, hence, the cost and complexity of implementation is reduced. Moreover, since (1) no SQL-transaction needs to be kept open for the duration of an ITX and (2) the middle-tier is not used as the repository for data modifications done by ongoing ITXs, this solves the scalability and reliability problems explained in Section 1. Frequently accessed data can still be cached in the middle-tier.

Each table that will be modified in a VDB is *version enabled* by augmenting its primary key with a version number. All rows stored in a VDB are tagged with a version number assigned to the VDB. A system-wide one-level deep tree of version numbers is maintained. The root VDB is assigned the root version number in this version tree and each of the other VDBs is assigned a leaf version number. All rows tagged with the root version number are initially seen from a leaf VDB. When a row is modified for the first time in a leaf VDB, the modification is done in a new copy of the row, i.e., a copy-on-demand approach is used. The new copy is tagged with the VDB version number. Subsequent modifications to the row in the VDB are done in place on the copy created above, without creating another copy. Similarly, new rows inserted in a VDB are also tagged with the VDB version number. A view called *VersionView(T)* is created on each version-enabled table *T*. This view shows only (1) the rows tagged with the current VDB version number and (2) the root VDB rows that have not been modified in the current VDB. It filters out the rows in sibling VDBs and the root VDB rows that have been modified in the current VDB.

The augmented table is renamed and the VersionView is given the original table name. As a result, user-issued SQL queries and data manipulation operations are now automatically done on the VersionView, instead of the table. Three triggers are created on each VersionView; one for each of insert, update and delete operations. These triggers implement the copy-on-demand approach described earlier. When the current VDB is set in a session, the current VDB ID is stored in a context variable that is later read inside VersionViews and view triggers to transparently determine the current VDB. These transformations ensure *SQL transparency*, i.e., ITX SQL code need not change to execute in a VDB. All the widely used database features like constraints and indexes are also supported in each VDB, without requiring any change in ITX SQL code.

When an ITX fails, the changes done by the ITX are easily rolled back by deleting the rows tagged with the version number of the VDB assigned to the ITX, without touching data in any sibling VDB. The rows to be deleted are efficiently identified using B-tree indexes on version identifier columns.

A VDB is merged by (1) deleting the root VDB rows that have been modified in the VDB and (2) updating the version number of all rows tagged with the VDB version number to the root version number. All the deletions and updates are done in a single SQL-transaction to ensure ACID properties for the merge process.

Persistent row-level locks that can last for the duration of a VDB are provided to ensure mutual exclusion between VDBs, and hence avoid conflicts. The lock information is stored in a metadata column of the augmented table. This column is checked inside view triggers to ensure mutual exclusion.

These versioning techniques use only widely available database features, i.e., column addition, views, triggers, stored procedures [2], table renaming and indexes. On most commercial databases, these features are available to application writers. Hence, these techniques can be easily implemented on most commercial databases, without making any changes in database kernel code.

3. EMPERICAL RESULTS

These versioning techniques have been implemented in Oracle Workspace Manager [2], the long transaction management component of Oracle Database. We used Workspace Manager to measure the total time taken for an ITX with and without VDB. We used a simple example ITX for this purpose. In this ITX, a three-part HTML form was filled to update information about a customer. Three tables were version enabled and these tables together contained half a million rows. The results are shown in Figure 1. The x-axis shows the number of rows modified by the ITX and the y-axis shows the corresponding ITX execution time. Since the time taken to enter data into the form varies with users, we avoided it by submitting the same data as initially displayed on the form, without modification. The columns *VDB*, *MidTier* and *SQL-TXN* respectively show the results when the changes done in the ITX were kept in a VDB, in RAM in the middle-tier, and in an open SQL-transaction. The results show that even excluding the time taken to enter data into the form, the overhead of using VDB is insignificant compared to the overall ITX execution time.

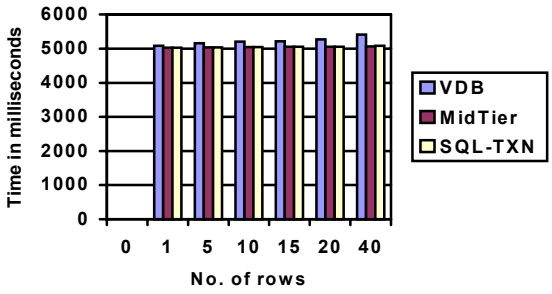

Figure 1. Overhead of using VDB for ITX management

4. CONCLUSIONS

We have identified some critical ITX implementation problems that have not been adequately addressed before. The data versioning techniques we presented can solve these problems, and thereby, reduce the cost and complexity of implementing ITXs and enhance their scalability and reliability. Our initial experiments indicate these techniques are effective in practice.

5. REFERENCES

[1] L. F. Cabrera et al., "Web Services Business Activity Framework (WS-BusinessActivity)", November 2004, http://www.ibm.com/developerworks/library/ws-busact.

[2] Oracle Corporation, "Oracle Database Application Developer's Guide – Workspace Manager", 10g Release 1, Part Number B10824-01, December 2003.