Eliminating ECperf Persistence Bottlenecks when using RDBMS with Pessimistic Concurrency Control

Samuel D. Kounev
Databases & Distributed Systems Group, Darmstadt University of Technology
skounev@informatik.tu−darmstadt.de

I. Introduction

After carrying out a number of experiments with Ecperf1.0, we noticed that the benchmark exhibits quite a different behavior depending on the type of RDBMS that is used for persistence. When an Optimistic Concurrency Control RDBMS such as Oracle8i is employed, the benchmark behaves as intended. However, when deployed with a Pessimistic Concurrency Control (CC) RDBMS, such as Informix Universal Server, the persistence layer seems to turn into a bottleneck preventing one to stress the application server and benchmark its performance.

In the following we take a closer look at the sources of this problem and then proceed to offer a concrete solution which eliminates the persistence bottleneck.

II. Deployment Environment

The ECperf EJBs and the Emulator Servlet are deployed on WebLogic Server 6.1, running under Solaris 2.7 on a SUN Ultra Sparc II machine with 1GB of main memory. The database − Informix Universal Server 21 is housed on a separate similar Sun Server with 2GB of main memory. The Driver is run on a client PC, running Red Hat Linux 7.1 with 192MB of RAM.

III. Problem Statement

Pessimistic Concurrency Control schedulers usually employ a locking−based protocol such as the popular 2PL in its many variants. Under 2PL data items are locked before being accessed. Concurrent transactions trying to access locked data items in conflicting mode are either aborted or blocked waiting for the locks to be released.

When we run ECperf out−of−the−box in such an environment we monitored the database and observed very high data contention levels. Large amounts of data access operations were resulting in lock conflicts which were blocking the respective transactions. High proportion of the latter were eventually being aborted either because of timing out or because of causing a deadlock. Different throughput levels were achievable depending on how the database was configured to behave on lock conflicts, i.e. whether to abort resp. transaction immediately or to block it with a configurable timeout period.

In any case, as soon as the transaction injection rate Ix was raised beyond 1 data contention soared, leading to what is often called data thrashing. As a result, most of the update transactions were being aborted and very poor throughput levels were achievable. The application server was ’relaxing’ while the DMBS was occupied with rolling back transactions. One could hardly stress and benchmark the application server in the presence of such a bottleneck.

The first thing that comes to mind when trying to reduce data contention is to decrease the locking granularity. After setting up Informix to use row−level locks (instead of page−level locks) we observed significant increase in throughput. To further optimize the data layer we tried decreasing the isolation level. The read−only entity beans (i.e. DiscountEnt, RuleEnt, SupplierEnt, BomEnt, PartEnt) can safely be configured to operate under the SQL TX_COMMITTED_READ isolation level. This could also be done for the rest of the entity beans at the risk of compromising data consistency and integrity at high transaction injection rates.

Other Application Server−specific tuning and optimization methods could be applied to further improve performance and reduce locking conflicts. For example, the above mentioned read−only entity EJBs could be explicitly declared as such, allowing the container to cache them across transactions and minimize database access calls.

While the above optimizations could help to reduce the identified bottleneck, they could not completely eliminate it and make ECperf behave as originally intended.

IV. Getting to the bottom of the problem

Here we take a closer look at the actual reasons for the problem. After carrying out a number of tests and carefully monitoring the database we noticed the following: the two crucial transactions WorkOrderSes.scheduleWorkOrder() for Planned Lines and WorkOrderSes.scheduleWorkOrder() for the LargeOrder Line are taking way too long to complete, while holding exclusive locks on some highly demanded database tables.

Lets follow the execution of the WorkOrderSes.scheduleWorkOrder() transaction (we consider its variant for Planned Lines):

1. WorkOrderSes.scheduleWorkOrder() Tx begins.

2. WorkOrderEnt.ejbCreate() is called.

Here an exclusive lock on the newly added row in the M_WORKORDER table is obtained, as well as an exclusive lock on the leaf node of the resp. table index (M_WO_IDX) !!!

3. WorkOrderEnt.process() is called:

3.1 AssemblyEnt.getBoms() (the M_PARTS and M_BOM tables are read)

3.2 For each BOM:

3.2.1 The respective ComponentEnt is loaded (M_PARTS table read).

3.2.2 The respective InventoryEnt is loaded (M_INVENTORY table read)

3.2.3 If new quantity needs to be ordered:

- ComponentEnt.addOrderedInventory() → InventoryEnt.addOrdered() → M_INVENTORY table to be updated on commit
- BuyerSes session is started if not already done, adding respective component quantity to be purchased (SComponentEnt.qtyDemanded possibly updated → S_COMPONENT table to be updated on commit)

3.2.4 ComponentEnt.takeInventory(qtyOff) → InventoryEnt.take() → M_INVENTORY table to be updated on commit

3.3 BuyerSes.purchase() is called (lets assume that a new purchase order is needed)

3.3.1 Select supplier(s) (load SupplierEnt entities → read S_SUPPLIER table)

3.3.2 POEnt.ejbCreate() → insert row in the S_PURCHASE_ORDER table and 1 or more rows in the S_PURCHASE_ORDERLINE table → exclusive locks on the resp. table rows and index nodes are set !!!

3.3.3 sendPO(POEnt) → Establish an HTTP connection to the Emulator servlet and send the order as an XML document

3.4 WorkOrderState is updated → WorkOrderEnt updated → M_WORKORDER table to be updated on commit

4. WorkOrderSes.scheduleWorkOrder() Tx is committed.

As we can see, even the above simplified sketch of the scheduleWorkOrder transaction, could well give us a feel of its size and complexity. As already mentioned this transaction was observed to take quite a while to complete. This is where we believe the root of the problem is.

More specifically, the following entity beans are created during the transaction − 1 WorkOrderEnt, 1 or more POEnt and 1 or more POLineEnt, causing locks to be set on the newly inserted table rows. Also possibly multiple instances of the InventoryEnt and SComponentEnt are updated and thus would require locks to be obtained on the respective table rows at transaction commit. In step 2 exclusive lock on the new row in the M_WORKORDER table is set as well as exclusive lock on the respective leaf node of the M_WO_IDX index. These locks are held until transaction commit. While the lock on the M_WORKORDER table might not be too big of a concern, the lock on the index may well block concurrent access to a large part of the table, if not the whole. This occurs typically when there are only a few table rows, which would, in the worst case, imply that all data entries of the index (typically a B+Tree) fit into a single leaf page. The latter is write−locked, preventing access to the table until the transaction commits. One might think, why don’t we then remove the index to avoid this. However, if the index is removed all read/write access to the table would require a table scan to locate the respective row, which would still encounter the exclusive lock on the newly added table record.

The same reasoning applies to the S_PURCHASE_ORDER and S_PURCHASE_ ORDERLINE tables which are locked in step 3.3.2. All these locks are held for the duration of the communication with the Emulator servlet in step 3.3.3. The latter may take considerable time especially if the server is under heavy load.

While the above concerns are to do with performance we also identified an even more subtle problem to do with the ScheduleWorkOrder transaction’s atomicity and semantics. All the actions in the transaction are reversible (undoable), except the communication with the Emulator Servlet in step 3.3.3. Once a new purchase order is sent to the servlet, its processing begins and this processing cannot be rolled back even if the scheduleWorkOrder transaction eventually aborts. Indeed, if that occurs, for example because of not being able to obtain the locks on the M_INVENTORY table at transaction commit, all actions of the transaction will be rolled back except 3.3.3. As a result, the respective POEnt and POLineEnt entities will be removed, but the Emulator won’t be aware that the order has been canceled and will continue processing it. Then later when the order is delivered the DeliveryServlet will try to find the corresponding POEnt/POLineEnt entities and will be ’surprised’ that they do not exist.

When running our experiments we monitored the database lock tables and our observations confirmed that indeed most of the lock conflicts were occurring when trying to access the M_WORKORDER, S_PURCHASE_ORDER and S_PURCHASE_ ORDERLINE or their indices. This further supported our initial feeling that it’s the access to these tables that makes the bottleneck.

V. Our proposal

Here we suggest a minor modification of the benchmark that aims to circumvent the identified bottleneck. We keep adhering to the EJB1.1 specification and try to minimize the changes needed as far as that is possible.

Our approach is to break down the WorkOrderSes.scheduleWorkOrder() transaction into smaller separate transactions. The main goal is to commit update operations as early as possible, so that respective locks are released. We strive to isolate time−consuming operations in separate transactions which do not require exclusive locks. At the same time we ensure that transaction semantics remain unaltered.

Here is how we proceed:

1. Eliminate the bottleneck on M_WORKORDER

We start by moving Step 3 (the WorkOrderEnt.process() call) in a separate transaction, making WorkOrderSes.scheduleWorkOrder() commit immediately after the creation of the WorkOrderEnt bean. In this way, locks on M_WORKORDER and M_WO_IDX are released much earlier. Stage 1 processing of the WorkOrder is encapsulated into a separate transaction of the WorkOrderSes bean, just like the next stages (updateWorkOrder and completeWorkOrder). The new transaction is called WorkOrderSes.processWorkOrder(). It takes as a parameter the ID of the work order, locates the resp. WorkOrderEnt bean and calls its process() method. The Driver code is modified to include a call to WorkOrderSes.processWorkOrder() after the scheduleWorkOrder() call. The web interface beans are also modified appropriately.

2. Eliminate the bottleneck on S_PURCHASE_ORDER and S_PURCHASE_ORDERLINE

While eliminating the bottleneck on the M_WORKORDER table was not that difficult, the bottlenecks on the S_PURCHASE_ORDER and S_PURCHASE_ORDERLINE tables proved to be a bit tougher to deal with. This is because Step 3 needs to execute atomically and therefore it cannot be chopped into separate transactions. However, we can still achieve a significant performance improvement if we take Step 3.3.3 out of the processWorkOrder transaction. Indeed, no one has said that the order needs to be send immediately after it is created. As a matter of fact, it is even more likely that in real businesses orders are accumulated and dispatched in batch mode by a separate process. As already alluded to, under heavy load the communication with the Emulator Servlet was observed to take much longer than usual. Therefore, it would be reasonable to ensure that no locks are held during this communication.

However, in order to make sure that orders are eventually sent we need to create an additional independent process, which periodically checks for new purchase orders and if such exist sends them to the Emulator servlet. The new process is implemented as a separate Agent (POAgent) similar to the LargeOLAgent. The only difference is that instead of checking for large orders and then scheduling them, the new POAgent checks for purchase orders and then dispatches them.

In order to distinguish between orders that have been sent and those that haven’t we add an additional attribute poStatus to the POEnt bean and a corresponding column PO_STATUS in the S_PURCHASE_ORDER table. A poStatus of 0 indicates that the order has not been sent and a poStatus of 1 indicates that sending has taken place. This attribute could also be used to further distinguish between delivered and undelivered orders.

Not only does this modification significantly reduce lock conflicts by making the processWorkOrder transaction shorter, but as a side effect we also eliminate the atomicity problem mentioned earlier. Namely, with this new design it is impossible for the Emulator Servlet to receive a purchase order that is later aborted.

VI. Preliminary Results

After carrying out a number of tests with the alternative design of the benchmark, we observed an astounding performance improvement in terms of throughput, which soared by at least a factor of 5. Almost no lock conflicts were observed on the problematic tables and the persistence bottleneck appeared to be eliminated. There are still some limited conflicts on the M_INVENTORY table, but there is little that can be done to eliminate them without significantly affecting the overall application design.