Profiling the Resource Usage of OLTP Database Queries

Note to reviewer: this paper benefits heavily from being viewed in color.

Bianca Schroeder
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA
bianca@cs.cmu.edu

1. INTRODUCTION

The goal of this project is to obtain a detailed profiling of the resource usage of database requests for OLTP workloads. Our motivation in doing so is very different from that of traditional database profiling: previous work on profiling has been done mainly to get insights into how to design better database systems; our aim is to evaluate the potential of query scheduling to improve database response times. More precisely, we want to collect measurements that will help us in answering the following question:

**Question:** Suppose you want to implement differentiated services in a database system, i.e. you want to be able to give different priorities to different queries. What are the mechanisms needed and how do you implement them?

We are interested in this question not only to be able to provide different classes of service, but we also hope to be able to use such prioritization functionality to schedule transactions in order to improve overall mean response times across the entire workload.

The answer to the above question depends on the resource usage patterns of the database queries. We first need to determine the bottleneck resource is, i.e. the resource at which resource a typical database query spends most of its time during processing, since this is where prioritization should be applied. We then need to understand the usage patterns for this bottleneck resource in order to design good mechanisms for giving high-priority queries preferred access to this resource. Furthermore, knowledge of the distributions of the resource requirements at the bottleneck resource is also helpful in evaluating the potential of query scheduling and to design effective scheduling policies.

The purpose of this project is to analyze the resource usage patterns of OLTP database queries in an experimental (in contrast to simulation based) environment with respect to the above question.

In our experiments we use two different database systems: IBM DB2, a popular commercial database system, and an open source system implemented on top of the Shore storage manager [2]. Our workload is based on the TPC-C benchmark [1], a standard benchmark for OLTP systems.

2. PREVIOUS WORK

To the best of my knowledge nobody so far has attempted to analyze database workloads with respect to the applicability of prioritization and query scheduling.

3. EXPERIMENTAL SETUP

As mentioned above, our experiments are based on the TPC-C benchmark [1]. This benchmark is centered around the principal activities (transactions) of an order-entry environment. More precisely, in the TPC-C business model, a wholesale parts supplier operates out of a number of warehouses and their associated sales districts. TPC-C simulates a complete environment where a population of terminal operators executes transactions against a database. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses.

We run this workload on two very different database systems: the commercial IBM DB2 system and a system based on Shore [2]. We break down resource usage into three different components: the time spent by a transaction on IO, CPU time and time waited to acquire locks. In DB2 we can obtain this breakdown through the "get snapshot" functionality of DB2 and through the use of DB2 event monitors. Obtaining the same breakdown for Shore is more complicated. First, since it's not a commercial system, there's little built-in functionality for monitoring system statistics. Secondly, the system is based on user level threads (unlike DB2 which uses processes to handle transactions) which makes it difficult to obtain CPU and IO usage on a per transaction basis. We therefore limit the analysis of the Shore system to the analysis of its lock usage.

The hardware platform used in the experiments is an Intel Pentium 4, 2.20GHz with 1 GB of main memory running Linux 2.4.17.
4. RESULTS
As explained in the introduction an important step towards answering question on is determine the bottleneck resource. We do this in Section 4.1. Then in Section 4.2 we analyze the resource usage at the bottleneck resource in more detail.

4.1 Determining the bottleneck resource
Since the resource usage of a DBMS depends on many system parameters, such as the workload or the size of the database, there is not one single answer to the question of which resource is the bottleneck resource. Hence, in this section we study the resource breakdown as a function of different system parameters. The parameters we take into account are level of concurrency, the size of the database and the type of the transactions.

In the TPC-C model the level of concurrency is determined by the number of terminal operators we simulated. In the following we refer to these terminal operators also as clients. The size of the database is given by the number of warehouses in the database, where one warehouse roughly corresponds to a size of 100 MB. The types of transactions are entering orders, delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses.

Section 4.1.1 studies the resource usage under different levels of concurrency and Section 4.1.2 analyzes the effect of the database size. In Section 4.1.3 we vary both the level of concurrency and the size of database, as suggested in the TPC-C guidelines. Finally, Section 4.1.4 is concerned with the different resource usage of different types of transactions.

4.1.1 Effect of level of concurrency
Figure 1 shows the execution time breakdown for experiments with 2 warehouses and number of clients ranging from 1 to 50. Figure 1(top) shows the results for the IBM DB2 system and Figure 1(bottom) shows the results for Shore. Note that 2 warehouses correspond to a database size of roughly 200 MB. Since the main memory of the server is 1 GB, this database is completely main memory resident.

The left plots in Figure 1 shows the breakdown of the execution time in absolute numbers (sec). The right plots show the contribution of the different resources as percentages of the total execution time.

We observe, that as expected, the total execution time grows with higher degrees of concurrency, since more clients contend for the same resources. More interestingly, we see that for both DBMS, the execution time is dominated by the lock wait times (except for very small number of clients). The contributions of CPU and IO are negligible. ¹

Figure 2 shows the execution time breakdown for experiments with 10 warehouses, which corresponds to a database of size a little more than 1 GB. The buffer pool size we chose is 800 MB, so the database doesn’t entirely fit into the bufferpool any more. As expected we see that the IO component is now larger than compared to the 2 warehouse case, but still lock wait times make up a significant fraction of the total execution time. In fact, even for a database of 30 WH, i.e. 3 times the size of main memory, the lock times still make up more than 50 percent of the total execution time for a realist number of clients (figure omitted for lack of space).

¹As a side note, observe that for DB2 there is always some portion of the time that is left unaccounted for. We haven’t completely resolved this issue at this point, but have some indication that DB2 is not always correctly accounting for IO.)
4.1.2 Effect of database size

In the next experiments we keep the number of clients constant at 10 and vary the size of the database from 1 warehouse to 20 warehouses. The results are shown in Figure 3.

It might seem surprising at first that the total execution time either hardly increases with growing database size (Shore) or even decreases (DB2). The reason lies in the significant decrease of lock wait times for larger databases: since we keep the number of clients constant, but increase the amount of data in the system the clients' accesses are distributed over a larger amount of data and hence there is less contention for the same data. As a result, for both DBMS the percentage of lock wait times of the total execution times shrinks from 90 percent for 1 warehouse to only 10 percent for 20 warehouses. At the same time the contribution of IO increases for larger databases. E.g., for 20 warehouses IO time makes up more than 20 percent of the total time.

4.1.3 Results for TPC-C standard parameters

While we have previously always kept either the database size or the number of clients constant, in this section we scale the number of clients with the size of the database. The TPC-C benchmark suggests to use 10 clients per warehouse. Figure 4 shows the results for this choice of parameters for database sizes ranging from 1 warehouse to 20 warehouses.

Both database systems show a clear increase in absolute lock times with increasing database size and number of clients. For the shore system also the percentage of the lock wait time of the total execution time slightly increases, while for the DB2 system the percentage decreases. One explanation might be that the IO overhead for DB2 is bigger than that for Shore (which as a storage manager specializes in smart IO) causing the relative contribution of the locking being higher under Shore than under DB2 for big databases.

4.1.4 Effect of transaction type

Recall that the TPC-C workload consists of five different transaction types:

- entering orders (Neworder)
- delivering orders (Delivery)
- recording payments (Payment)
- checking the status of orders (Ordstat)
- monitoring the level of stock at the warehouses (Stocklev)

It turns out that the 5 transaction types can be classified into three categories with respect to their resource usage. Figure 5 shows one representative for each category. For all experiments in Figure 5 we scale the number of clients with the number of warehouses as suggested by TPC-C (using ten clients per warehouses)

The ordstat transactions form the first category which is completely IO bound. The reason for ordstat transactions being IO bound rather than lock bound is most likely that they are read-only operations and hence they never need to acquire a lock in exclusive mode.

The transaction types in the second category are completely lock bound. Payment, stocklevel and neworder fall into this category. Payment and neworder are write-heavy operations and therefore have higher chances of having to wait for locks than for example ordstat, since they need to obtain exclusive locks. Stocklev on the other hand is lock bound although it is a read-only operation like ordstat. The reason might be that stocklev reads data that is frequently updated forcing it to wait for transactions holding exclusive locks on data.
The third category is comprised of the delivery transactions, which can be either IO bound or lock bound depending on the size of the database. For in memory databases (less than 10 warehouses) the majority of the execution time is spent waiting for locks, for out of memory databases (more than 10 warehouses) IO time dominates execution times.

To summarize the results so far, we find that independently of the DBMS used (DB2 or shore) lock wait times dominate a transaction’s execution times for reasonable relation between the size of the database and the level of concurrency. This partially answers Question 1 from the introduction: based on our results so far locks are a bottleneck resource in many situations and hence it seems promising to concentrate on managing locks to implement priorities. To get more insights into the mechanisms that might be effective in implementing priorities based on locks, the next section analyzes the lock usage patterns in more detail.

4.2 Analyzing usage patterns at the bottleneck resource

We have seen that in most realistic scenarios locks are the bottleneck resource. To implement a successful prioritization and scheduling scheme, we need to understand the characteristics of the bottleneck resource usage in more detail.

As an example consider the question of whether the scheduling should be limited to letting high priority transactions jump ahead in the lock queue, or whether high priority transactions in the queue should also be allowed to abort a low priority transaction holding the lock. The first approach is in general more desirable since it is work-conserving: aborting the transaction in progress would not only waste the work that has been done on it so far, it also creates extra work since this transaction needs to be rolled back. On the other hand, just reordering the queue can only have an effect if in general the queue is long, i.e. a transaction waits most of their time for other transactions in the queue and not the transaction that’s holding the lock.

In the following we give some more detailed information on the characteristics of lock wait times. All the information was obtained by instrumenting Shore.

Table 1 shows for various warehouse and client numbers the average total execution time per transaction, the average wait time per transaction, the average number of times a transaction has to wait and the average wait time for each lock that a transaction has to wait for.

<table>
<thead>
<tr>
<th>WH</th>
<th>CL</th>
<th>Total/X (sec)</th>
<th>Wait/X (sec)</th>
<th># Waits /X</th>
<th>Wait/Lock (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0.57</td>
<td>0.39</td>
<td>1.72</td>
<td>0.23</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>1.12</td>
<td>0.12</td>
<td>1.81</td>
<td>0.45</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>3.04</td>
<td>2.31</td>
<td>1.78</td>
<td>1.29</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>7.68</td>
<td>6.13</td>
<td>1.85</td>
<td>3.30</td>
</tr>
</tbody>
</table>

Table 1: Lock wait time results obtained from Shore.

We see that on average each transaction waits less than 2 times for a lock (column 2), independently of number of warehouses and clients. However, each of these waits is costly: the average time for one lock wait ranges from 0.23 to 3.3 sec, depending on the setup, and is usually on the order of 40 percent of the average total execution time.

Next we try to answer the question whether the longs waits are due to waiting for one transaction holding the desired lock for a long time, or due to a long queue in front of the lock. Table 2 shows the average length of the lock queue for various warehouse and client numbers. To better understand the relation between wait time and queue length we compare...
the queue length for as seen by long wait events to the queue length seen by short wait events. More precisely, we order the wait events by the length of their wait time and compute the average queue length seen by the top and the bottom 10 percent of these events, respectively. These are listed as LongWait-Qlength and ShortWait-Qlength in Table 2.

<table>
<thead>
<tr>
<th>WH</th>
<th>CL</th>
<th>Avg. Qlength</th>
<th>LongWait Qlength</th>
<th>ShortWait Qlength</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5.78</td>
<td>5.81</td>
<td>2.58</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>6.04</td>
<td>6.60</td>
<td>2.41</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>6.23</td>
<td>7.05</td>
<td>2.84</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>6.50</td>
<td>7.11</td>
<td>2.93</td>
</tr>
</tbody>
</table>

Table 2: Relation between length of lock queue and lock wait time for Shore.

We observe that the average length of the lock queues is around 6, independent of the setup. We also observe that in the case of long waits the average queue length is much larger than in the case of short waits. This indicates that long waits are due to long queue lengths, rather than a single other transaction blocking the lock for very long.

Yet another way to look at the characteristics of the lock queue lengths and lock wait times is to consider the corresponding cumulative distribution functions (Figure 6). It is interesting to observe that both the cdf for the lock queue lengths and the cdf for the lock wait times follow a very similar shape. The curves are mostly linear and indicate that most of the values are distributed uniformly over a certain interval. Also observe that in less than 15 percent of the cases the transaction which enters the wait queue is first in line (the only transaction waiting). In more than 50 percent of the cases more than 7 other transactions are ahead of a transaction entering the wait queue.

Based on the results in this section we believe that a prioritization scheme that simply reorders the lock wait queues might be an effective way of prioritizing transactions.

5. CONCLUSION

This paper studies the resource usage patterns of OLTP transactions in order to better understand the mechanisms needed to implement differentiated service classes in DBMSs. We find that for OLTP workloads with a reasonable relation between database size and level of concurrency, lock wait times dominate execution times. This is the case for both the commercial IBM DB2 system and an open source system based on Shore. A closer analysis of the lock usage patterns reveals that most of the lock wait times are spent in very few waits which last very long due to a long lock wait queue. These results suggest that a prioritization scheme that simply reorders the lock wait queues might be an effective way of prioritizing transactions.

6. ACKNOWLEDGMENTS

Much of the work on IBM DB2 is joint work with David McWiler. Many thanks to Mengzhi Wang for providing the TPC-C kit for Shore.

7. REFERENCES
Figure 5: Transaction types Results for ordstat, payment and delivery transactions on IBM DB2.

Figure 6: Cumulative distribution functions for length of lock queue (left) and time waited per lock (right).