Reducing Root Contention in Lazy Tree-structured Databases

Sophie Lathouwers
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
s.a.m.lathouwers@student.utwente.nl

ABSTRACT

Databases are expected to handle large amounts of transactions concurrently. Many databases block when a bulk transaction needs to be committed. A possible solution is to use a lazy tree-structured database to provide improved throughput while executing concurrent OLTP and bulk transactions. However in this solution all transactions are applied to the root, which causes root contention. This occurs mostly when many small transactions are added to the root after which they have to be pushed down to the leaves of the tree. In this paper a possible approach is discussed and implemented to reduce root contention. This approach consists of splitting the data structure in multiple smaller data structures, each with their own root, in order to divide the root contention over the multiple roots. Micro-benchmarks and the TPC-C benchmark are then used to compare the performance of the approach to the original solution where no special approach has been taken to reduce root contention. The results show that the solution provides an improved throughput and scalability compared to the original solution. Depending on the size of the database, it also provides an improved scalability compared to ScalaSTM.

Keywords

Root contention, transactions, lazy evaluation, concurrency, bulk update

1. INTRODUCTION

Contemporary companies must handle a great amount of transactions concurrently and their data needs to be available at all times. These concurrent transactions are needed to allow many users to use the same database at the same time, while ensuring that no conflicts will occur between these transactions.

Several solutions have been found to execute transactions concurrently in case the transactions do not overlap, such as two-phase locking (2PL) [2] and optimistic concurrency control [7]. However, the problem persists when transactions do overlap, in particular when data needs to be updated in bulk. In order to successfully perform bulk updates, most databases need to obtain a lock on the complete table which completely disables any other concurrent transaction until the complete bulk update has been performed.

25^{th} *Twente Student Conference on IT* July 1^{st}, 2016, Enschede, The Netherlands.

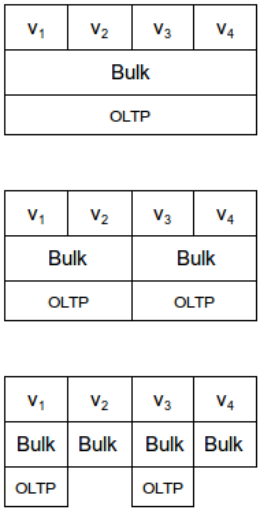


Figure 1: A bulk and OLTP transaction are added to the root of the tree and pushed down.

Several techniques to overcome this problem have been proposed such as the concurrency control mechanism described in [10]. In this concurrency control mechanism, the reader gets the responsibility of executing operations instead of the writer that usually is responsible for this. Transactions write suspended computations instead of computed values. These computations are evaluated lazily, enabling the reader to prioritize evaluation of the operations [4]. This mechanism uses a tree-structured index over the data to allow bulk operations to be committed instantaneously. Whenever a new transaction should be committed, it is queued at the root of the tree and then it is pushed down towards the leaves. This process is depicted in Figure 1 with a database consisting of four variables v_1, ..., v_4 where two transactions are added, called "Bulk", which updates all variables, and "OLTP", which updates v_1 and v_3. At the top you can observe that the transaction "Bulk" and "OLTP" have been added to the root of the tree. In the middle you can see the next step, in which the two transactions have been pushed down to the next level in the tree. In order to perform this action, the transactions are split into smaller transactions. In the bottom, the next level of the tree is depicted where the leaves are situated. The transactions are again split into smaller transactions. The "OLTP" transaction only updates v_1 and v_3 and thus it does not occur at the leaves of v_2 and v_4. Now both transactions have been pushed

down in the tree. When the value of one of the variables is requested, the suspended computations piled on the leaf belonging to this variable will be evaluated.

1.1 Problem Statement

Applying large amounts of mainly very small transactions, to the root of the tree-structured database, can cause a significantly reduced throughput. This problem is called *root contention*, which limits scalability and throughput.

1.2 Research Questions

To find a solution, the following questions have to be addressed:

1. How can root contention be reduced in lazy tree-structured databases?
2. To what extent does splitting the tree-structure into smaller partitions affect the scalability and throughput?

In this research partitioning is investigated as a method to reduce root contention. Partitioning can be described as splitting the data structure into smaller partitions, which results in multiple smaller trees and thus also multiple roots. A concurrency control mechanism is needed to ensure that all operations are performed atomically. Several concurrency control mechanisms can be used; this research uses 2PL. The expectation is that when the data structure is split, the root contention will be split over the multiple partitions as well.

Experiments have been carried out to compare the performance of this splitting method to the original solution proposed by Wevers et al. [10] Scalability and throughput have been measured to assess the performance. To measure the performance the TPC-C [9] benchmark has been altered such that the database will be split in multiple partitions and 2PL will be added to this structure to ensure atomicity.

Section 2 introduces several concepts and terminology that are used in this paper. Section 3 describes work that is related to this research. Then Section 4 explains the methods of the research. Several techniques which could be used to reduce root contention are discussed in Section 5. Details about the implementation of the micro-benchmarks and the results, including discussion, are presented in Section 6. The implementation and results, including discussion, of the TPC-C benchmarks are discussed in Section 7. Section 8 discusses the results compared to the expectations and introduces some points of discussion. Finally, in Section 9 the conclusion is presented and future work is suggested.

2. BACKGROUND

A *concurrency control mechanism* is needed to ensure atomicity when committing transactions. Atomicity is one of the four properties which are collectively called ACID (Atomicity, Consistency, Isolation, Durability) [5]. Atomicity states that either the whole transaction should succeed or fail; the transaction should not partially succeed. Consistency ensures that only valid states of the database will be committed. Isolation asserts that the concurrent execution of transactions results in the same state as when the transactions are performed sequentially. Durability ensures that the result of committed transactions will survive any malfunctions. To show the importance of a concurrency control mechanism, an example is provided. Take a database with two variables v_1 and v_2. There will be

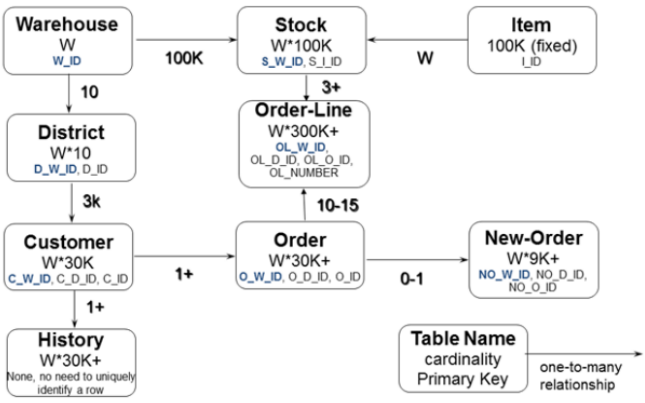


Figure 2: An outline of the TPC-C benchmark tables [6]

two transactions, T_1 and T_2, that want to commit. Both transactions will update v_1 and v_2. If no concurrency control is used it may occur that at v_1 the computations are added in the order T_1 followed by T_2, and at v_2 they are ordered as T_2 followed by T_1. If these would be committed, it might result in incorrect values if, for example, T_2 was dependent on the result of T_1.

Two-phase locking (2PL) is a concurrency control mechanism that can be used to ensure atomicity when committing transactions to the split data structure. 2PL operates according to three rules [2].

1. When a transaction a is received, the scheduler will check whether a conflicts with an already set transaction b. If so, a will be delayed until it can obtain the lock that it wants. If not, a may obtain the lock immediately.
2. Once a lock has been obtained by a transaction a, it may not release this lock at least until the transaction's operation has been performed.
3. Once a lock has been released for a transaction a, it may not subsequently obtain any other locks.

"An important and unfortunate property of 2PL schedulers is that they are subject to deadlocks." [2]

Optimistic Concurrency Control (OCC) [7] is a concurrency control mechanism where no locks are used. This is another concurrency control mechanism that could ensure atomicity when committing transactions to the split database. The main assumption that is made for optimistic concurrency control is that different transactions often complete without interfering one another. OCC can be described using four phases:

1. Create a timestamp that indicates the beginning of the transaction a. This may be needed in the last phase.
2. Transaction a is performed tentatively.
3. Before transaction a is committed, validation is performed to check whether another transaction b has modified any data that was used by transaction a.
4. If data has been modified, transaction a will be aborted; a rollback is executed until the timestamp created in the first phase. If not, the transaction is committed.

The *TPC-C benchmark* [9] is a benchmark for OLTP transactions. This benchmark tries to model a realistic database system which is centered around an order-entry environment. It represents this environment for one company. A company has a number of warehouses where the stock is stored. Each warehouse has 10 districts, each serving 3000 customers to whom the company can sell items. To serve their customers, five concurrent transactions are available for use. These transactions include monitoring stock at a warehouse, checking an order's status, recording payments, delivering orders and entering new orders. The database consists of nine tables which are depicted in Figure 2. In this figure one can observe how the tables in the benchmark depend on each other. Note that the rows of all tables, except for the table "Item", depend on the number of warehouses.

The TPC-C benchmark may be influenced by the system architecture as cache contention can lead to a significant degradation of performance for multi-core systems [11]. For example if multiple threads edit the same cache-line, this can result in lock contention which can lead to reduced scalability.

3. RELATED WORK

There exists prior work on lazy databases as well as tree-structured databases and possible concurrency control mechanisms.

Wevers et al. describe a concurrency control mechanism based on lazy evaluation [10]. This concurrency control mechanism has also been implemented and evaluated with micro-benchmarks. This paper also describes in its future work that root contention limits this approach. This research will address and test a possible solution for this.

Faleiro et al. [4] describe how lazy transactions can be used in databases and what the advantages and challenges are that come with it. They have also shown that some workloads are less suitable for lazy evaluation. Their approach differs from our approach as they delay evaluation whereas in our approach the transactions are split into smaller parts.

Argo et al. [1] have explored the possibility of a DBMS in a purely functional language. They assume a binary tree as a structure for the database. Some drawbacks are also pointed out, such as abortable transactions and the use of balanced trees. Their approach also uses a tree-structure and lazy evaluation to effectively execute transactions concurrently. Their approach also points out that the tree needs to be balanced whereas in this approach this is addressed by using tries.

Mu et al. [8] explain a new concurrency control protocol for distributed transactions which outperforms two-phase locking and optimistic concurrency control. This is a protocol that could help solve root contention.

4. METHODS OF RESEARCH

There are several ways in which root contention could be reduced such as using concurrency control [2, 7, 8] on the leaves and splitting the overall data structure in smaller partitions (See Section 5). The splitting of the overall data structure into smaller partitions will be investigated and taken as the option to be implemented and experimented with. This approach has been chosen as using concurrency control on the leaves could introduce large amounts of overhead for the concurrency control mechanism and thus may not be the best approach. Moreover partitioning the database can lead to using concurrency control on the leaves, if the database is split in partitions consisting of only one variable.

The implementation will build upon the current implementation made by Wevers discussed in [10], which has been made using Scala [3]. The approach of partitioning the database will be evaluated using micro-benchmarks and the TPC-C benchmark. The results will consist of measurements showing the performance of the implemented solution on throughput and scalability. For implementation details of the micro-benchmarks and TPC-C benchmarks, see Section 6.1 and 7.1 respectively.

The effect of the number of partitions, number of partitions per transaction and the number of operations per transaction are investigated using micro-benchmarks. The TPC-C benchmark is used to investigate the effect of the number of warehouses on scalability. The TPC-C benchmark will also be used to investigate the scalability compared to the original solution, as proposed by Wevers et al. [10], and ScalaSTM.

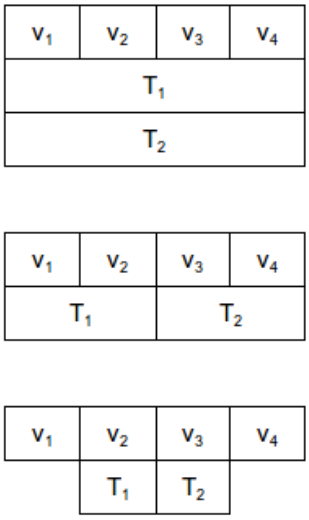
Our hypothesis is that the implemented solution will provide an improved throughput in workloads where many overlapping small transactions, which are distributed over different partitions, are committed compared to the performance of the original solution. A similar throughput to the original solution is expected when the transactions want to change variables in the same partition. In this case, all transactions are applied to the same root, thus root contention will occur resulting in a similar situation as the original. Note that some overhead of the concurrency control mechanism can result in slightly worse results than the original solution. Worse results may also occur when very large updates are executed as this can result in lock contention.

All experiments have been run on a computer with 4 AMD Opteron(tm) Processors 6376, where each one has 16 cores. Thus the computer can run 64 threads concurrently, and it has 516GB RAM. The OpenJDK Runtime Environment IcedTea 2.6.6 with Java version 1.7 is used on Linux kernel 3.13.0. Every experiment has been run with the flag -XX:+UseNUMA.

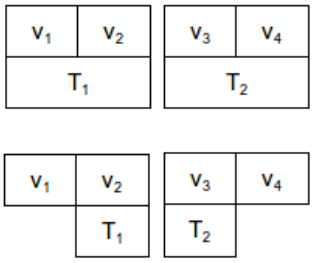
5. TECHNIQUES TO REDUCE ROOT CONTENTION

A possible approach to reduce root contention is to use a concurrency control mechanism such as 2PL or OCC on the leaves of the tree. To be able to use concurrency control on the leaves to solve root contention, the suspended computations should be added to the leaves directly instead of at the root of the tree. This evades the problem of root contention, although concurrency control is needed to ensure correct commits of transactions.

Another possible approach to reduce root contention is to split the tree-structure into multiple partitions. Splitting the data structure into smaller partitions will result in a collection of several smaller trees which each have their own root. For example the tree of Figure 1 could be split into two smaller trees. The first tree will contain the variables v_1 and v_2 and the second tree will contain the variables v_3 and v_4 (See Figure 3b). The first transaction which will be committed wants to update v_2. The second transaction wants to update v_3. To commit the first transaction, it will be applied to the root of the first tree. The second transaction has to be applied to the root of the second tree. Thus one transaction need to be pushed down in the first tree and one transaction in the second tree (See Figure 3b). If the tree was not



(a) Two transactions are added to a database that is not partitioned and pushed down.



(b) Two transactions are added to a partitioned database and pushed down.

Figure 3: Difference between adding and pushing down two transactions to a database that is not partitioned, and to a partitioned database.

split, two transactions would have to be pushed down in the original tree structure (See Figure 3a). Thus by splitting the data structure in smaller partitions, which results in having multiple roots, root contention was reduced in this case. A concurrency control mechanism is needed to correctly commit transactions. An example will be given to show the importance of using a concurrency control mechanism. Take a database consisting of four variables, v_1, ..., v_4, which is split into two smaller partitions. The first partition contains v_1 and v_2, the second partitions contains v_3 and v_4. Now take the transactions T_1, which will update v_1, ..., v_3, and T_2, which will read v_2. If no locks are used, it is possible that v_2 is read before it has been updated, thus resulting in a different result compared to when transactions are executed sequentially. However, transactions should uphold the properties ACID. The isolation property ensures that any concurrent execution of transactions results in the same state compared to when the transactions are executed sequentially.

Several concurrency control mechanisms can be used such as 2PL and OCC. Bulk transactions executed concurrently with OLTP transactions likely result in many conflicts. Therefore it is concluded that OCC may not be the best option as this concurrency control mechanism's main assumption is that the transactions often complete without conflicts. Thus 2PL has been chosen to be implemented.

6. MICRO-BENCHMARKS

6.1 Implementation

For the micro-benchmark the following variables need to be defined: number of threads, time to run and the workload. The workload depends on four other variables

1. *Number of partitions*: This number defines in how many partitions the database should be divided.

2. *Number of elements per partition*: This number defines how large the partitions are in terms of number of elements. An element represents a variable in the database.

3. *Number of partitions per transaction*: This number defines how many partitions a transaction will, at most, update.

4. *Number of operations per transaction*: This number defines how large a transaction is in number of operations.

The effect of the number of partitions, number of partitions per transaction and the number of operations per transaction are investigated using micro-benchmarks. To investigate this, these factors will be changed while the other factors stay constant. In order to determine the effect of the factors, the amount of operations performed is compared.

In the beginning of the micro-benchmark a new database is instantiated as well as a reentrant lock for each partition of the database. Then, based on the number of partitions per transactions, partitions which should be updated are randomly chosen. For each transaction per partition that should be updated, an update function is made. An update function stores the changes between the old and new data. This update function will store the update function that needs to be applied for each transaction. The elements which should be updated in each partition are chosen at random. Each specific variable will be updated such that the new value equals the old value plus one. When everything that should be updated has been chosen at random, all partitions that need to be updated will be locked. Note that it is important that the locking will always happen in the same order otherwise a deadlock may occur. To avoid this problem, the array containing partitions which need to be updated will always be ordered before any locks are used. Then for each transaction, the updated partitions will be locked and the update function will be applied. The database is then updated with this result. Now all locks which were acquired will be released. At last, for each update function the warehouse which was updated will be forced to evaluate. Immediate evaluation is preferred to avoid memory constraints and it does not reduce concurrency compared to delayed evaluation [10].

The constant factors have been chosen based on the description of the TPC-C benchmark. In Table 1 the constant factors are shown. The first variable is the number of partitions per transaction. Note that this does not mean that it will update only five variables in a partition, as a transaction may consist of multiple operations. The partitions to be updated will be chosen at random and therefore it could occur that one partition may be updated multiple times, therefore this number defines how many partitions a transaction will change at most. The second variable is the number of operations per transaction. The chosen value, 10, has been chosen as this is the average number

Table 1: Constant factors in micro-benchmarks

Name	Value
Number of partitions per transaction	5
Number of operations per transaction	10
Number of elements per partition	100.000
Number of partitions	150

of different items in a new order in the TPC-C benchmark. This seemed like a representative value as entering a new order is the most frequent type of transaction in the benchmark. The third variable is the number of elements per partition, 100.000, which corresponds to the number of items for which a warehouse maintains the stock in the TPC-C benchmark. The last value, the number of partitions, is based on what would seem a good split to maximize concurrency. Two options seemed reasonable: splitting at the warehouses or splitting at the districts of warehouses. Splitting at the warehouses results in one level less in which the variables have to be pushed down whereas splitting at districts results in two levels less in which the variables have to be pushed down. Moreover splitting at districts results in more roots to commit to which could lead to better division of the workload over the different roots. Therefore it has been chosen to split at the districts of warehouses. Each warehouse in the TPC-C benchmark must have ten sales districts. Assuming there would be 15 warehouses, this results in 150 partitions.

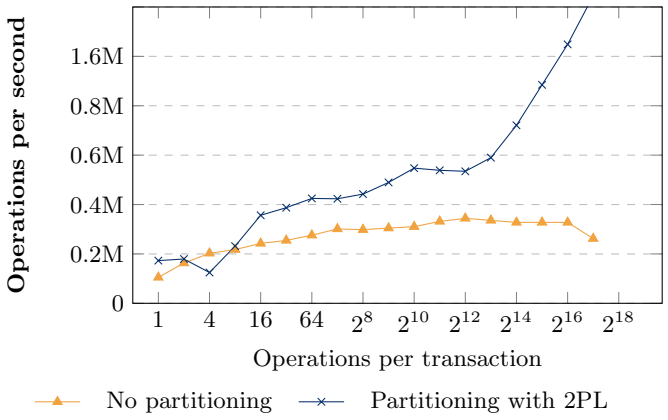
6.2 Results

The micro-benchmarks will be run in 8 concurrent threads and each run will be given 2 minutes. Outliers have not been discarded. The effect of the following three factors on throughput is investigated: number of operations per transaction, number of partitions per transaction and elements per partition. In each graph you can observe one line representing the situation when no partitioning is used and one line representing the situation with partitioning.

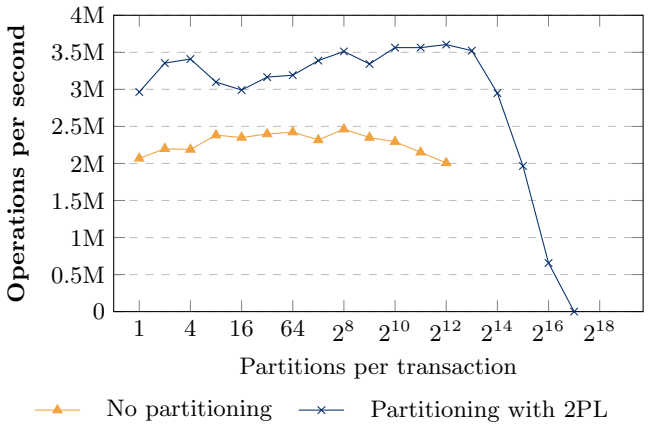
In Figure 4a the effect of the number of operations per transaction on throughput is shown. When transactions consist of a large amount of operations, the throughput increases drastically when using partitioning with 2PL compared to using no partitioning. Note that this is because many operations are performed on the same variable. Overall partitioning provides an improved throughput. Therefore we conclude that partitioning with 2PL provides improved throughput if a transaction consists of 8 or more operations.

Figure 4b shows the effect of the number of partitions per transaction. Partitioning with 2PL provides an improved throughput throughout all amount of partitions per transaction. When no partitioning was used, data could only be collected up until 2^{12} transactions as afterwards the stack was not large enough. The decrease in throughput when using 2^{14} or more partitions per transaction is caused by lock contention. When using 2^{14} partitions per transaction, and 10 operations per transaction, it results in 163.840 operations that need to be performed. Since the database has a total of 150.000 elements, lock contention is guaranteed to occur. When using more partitions per transaction, lock contention will only increase resulting in less throughput.

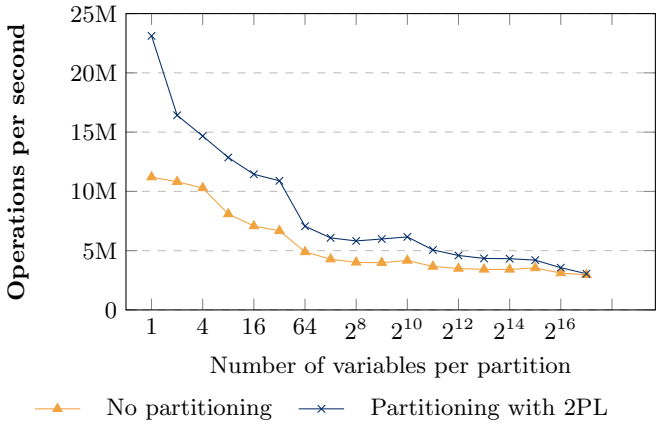
The effect of the partition size on the throughput is shown in Figure 4c. Partitioning with 2PL provides improved throughput when the partitions are smaller and it starts to act similar to no partitioning when the partitions be-



(a) The effect of the number of operations per transaction on throughput, where each operation is an update.



(b) The effect of the number of partitions per transaction on throughput, where each operation is an update.



(c) The effect of partition size on throughput, where each operation is an update.

Figure 4: Results of the micro-benchmarks

come larger. An important note with this experiment is that for partitioning the standard 150 partitions were used whereas the base case has only one partition. As a result the database used without partitioning is 150 times smaller than the database used with partitioning. This may influence the results as smaller databases are more prone to conflicts. Thus using no partitioning results in a higher probability of conflicts in the whole database. One can roughly guess the throughput, with no partitioning, based on the database size. This can be done by taking a number of variables in the partitioning with 2PL, for example 16. The database for this case consists of 150 partitions with each 16 elements thus the database has 2400 elements. You can then look up this number on the x-axis, this is roughly around 2^{11}, and read the corresponding value belonging to no partitioning. In this case no partitioning has a throughput of around 4.5M whereas partitioning has a throughput of around 11M. This indicates that partitioning a database in smaller partitions provides improved throughput as stated earlier.

6.3 Discussion

Transactions with 16 or more operations per transaction had an improved throughput when partitioning was used with 2PL. This is because, due to the partitioning, the transactions need to be pushed down less levels. When the transaction consists of more operations, the chance increases that more variables need to be updated. Therefore the transaction likely needs to be pushed down to more leaves. The throughput increases when more operations are done per transaction, because the advantage of having to push down less levels increases when this has to be done for more variables.

Partitioning with 2PL provided overall improved throughput for different number of partitions per transaction. This is due to the fact that when using partitions, the transactions need to be pushed down less levels. The amount of levels stays the same throughout the benchmark, thus the overall improved throughput is also constant.

The throughput was improved when using smaller partitions and more operations per transaction. This is as expected as the transactions need to be pushed down fewer levels in the tree-structure. Thus one can conclude that there is an overall improved throughput.

It is important to note that, in the micro-benchmarks, forcing warehouses per update function may result in forcing one warehouses multiple times if this warehouses has been updated multiple times. This could be addressed by storing all updated keys in an array and to force all the unique keys.

7. TPC-C

7.1 Implementation

Now the implementation for the TPC-C benchmark will be discussed which is based on the implementation of Wevers et al. [10]. In their implementation the database consists of two maps. The first contains all warehouses and the second contains all items. In this implementation the choice has been made to use an array containing all warehouses. Another array has been made which contains a reentrant lock, one for each warehouse. Padding has been added to the array containing the warehouses in order to avoid cache-line contention [11]. The padding has been added by making the array 16 times as big and every i^{th} element of the non-padded array was placed on the $16 * i^{th}$ entry of the padded array. This padding changes depending on which system architecture is used. Therefore the specific padding used in this case may need to change depending on the system architecture to avoid cache contention.

Note that the results have been corrected for the time spent on garbage collection. This has been done by measuring the time spent on garbage collection, which is used to calculate the ratio spent on garbage collection. The amount of transactions that could be performed is then calculated by dividing the amount of transactions performed by the ratio spent performing transactions. This ratio is calculated by $1 - r$, with r is the ratio spent on garbage collection.

The TPC-C benchmark is meant to be measured in tpm-C, the number of orders fully processed per minute. However multiple runs of one second were used instead to reduce the runtime of the experiments.

The TPC-C benchmark has been measured using 2^0, 2^1, ..., 2^8 threads. For each measurement nine runs have been made, where the two worst and two best results have been discarded, then the average has been calculated which is taken as the final result of a run. The two best and worst results are discarded to avoid influence of outliers. A run is influenced by the following five other factors: introduction time, running time, number of threads, number of warehouses and the defined workload.

The introduction time defines a warm-up time in which no measurements are collected and which is used to try and avoid any side-effects. The running time defines how long the test will run. The number of warehouses determines how many warehouses there are in the database and thus how large the database is. The workload defines how each transaction of the TPC-C benchmark will be handled. While a workload is running, it will choose which method to execute taking into account that some methods are executed more frequently. Monitoring stock level, checking an order's status and delivering an order all have a chance of 4%, recording payments has a chance of 43% and entering a new order has a chance of 45% to be executed.

The methods that have been made are similar to those of Wevers et al. [10]. Therefore a rough outline is given and the changes that have been made are discussed in more detail.

When entering a new order, first all order lines and the total amount of the order are computed optimistically. Then an update function for stocks per warehouse has been made. This update function needs to be applied to the stocks of the warehouses. Then another update function is made in order to insert the order into the warehouse and update the stock. It is important that the update functions have been merged, otherwise the warehouses need to be locked longer in order to apply all updates. The update functions are then merged so that only one update function needs to be applied to the database. The last update function, which inserts an order, is merged with the update function which updates the stock mentioned earlier. Locks will be needed when committing the transactions to the database, this is the same as applying the update function to the database. For each variable that is updated by the update function, a lock on the corresponding warehouse is acquired and the update function is then applied to the warehouse. When all warehouses have been updated, all locks are released. At last all newly added suspended computations are evaluated. The suspended computations include the new order, the related customer and the stock of the ordered items.

Recording a payment starts by retrieving the customer's id using a snapshot of the database. Update functions are then made which contain information such as an update of the year-to-date of the warehouse and district as well as financial information of the customer. For each variable that is updated by the update function, a lock is acquired on the corresponding warehouse. The update function is then applied to the warehouse and the lock is released. The suspended computations that have been added to the district and customer will then be forced to evaluate.

Checking an order's status is a read-only operation. Reads can be quite expensive as they can trigger a large number of suspended computations. Therefore reads should be performed on a snapshot when no locks are active. Thus first a snapshot of the corresponding warehouse is made after which all other information is derived from this snapshot. Only one warehouse is needed to check an order's status, therefore no locks are needed to obtain this snapshot. However if multiple warehouses are required in a read-only operation, it is important to note that locks are needed to obtain this information. Otherwise it is possible that you take a snapshot of the first warehouse, then a transaction updates the first and second warehouse, after which you take a snapshot of the second warehouse. In this case the combined snapshots represent a situation that has never occurred in the database.

The method used for delivering an order starts by taking a snapshot of the database. For each warehouse the districts need to be updated with information such as determining the new order lines and updating the customer's order with the delivery information. When these new changes have resulted in a list of updated districts, the commit phase is started. First a lock is acquired on the warehouse of the districts. Then the districts of the warehouse are updated with the list of districts. At last the lock is released. The suspended computations, that have been added to the database, will then be forced to evaluate.

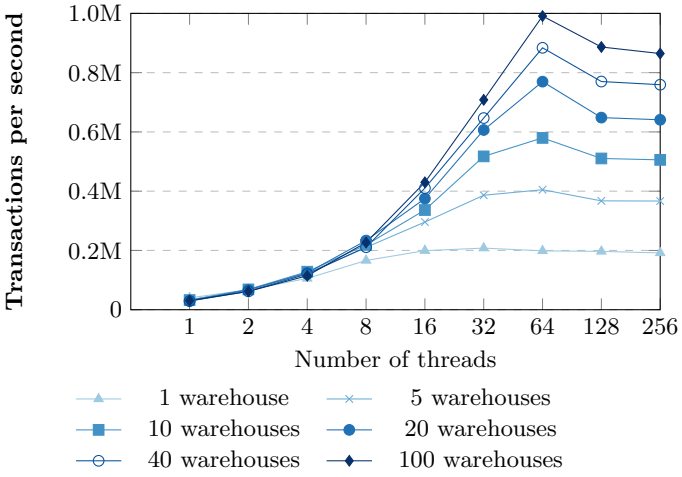
Monitoring the stock level is, just as checking an order's status, a read-only operation. This read operation also has only one corresponding warehouse, thus no locks are needed. Therefore first a snapshot is taken of the warehouse of which the stock level needs to be overseen. The read operations are then performed on this snapshot.

Note that all operations, which are not read-only, use locks in their commit phase. Although another operation may commit between the commit phase and the forcing of the evaluation of the current operation, it is certain that the suspended computations have been committed and therefore will be forced to evaluate.

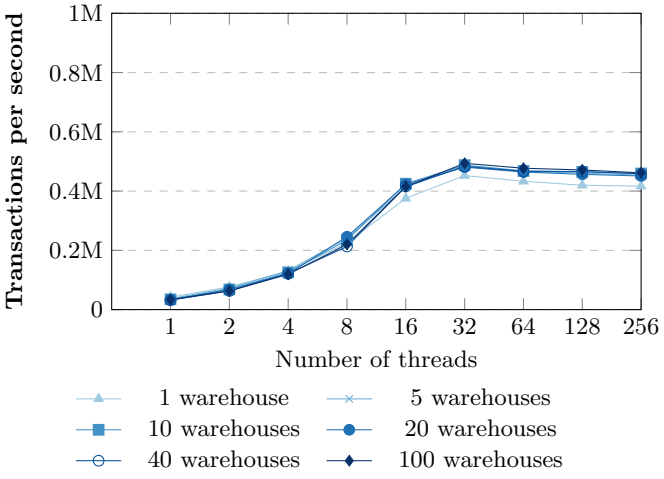
7.2 Results

The TPC-C benchmark will be run in various numbers of concurrent threads and each run will be given 1 second. The effect of the following factors on scalability is investigated: number of warehouses and number of items in a warehouse.

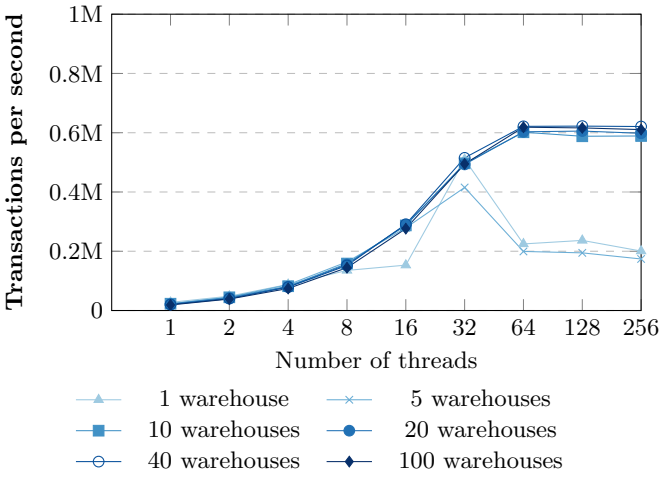
The effect of the number of warehouses on the scalability has been measured using the TPC-C benchmark. The results are depicted in Figure 5a. One can observe that the scalability improves when the database consists of more warehouses. Comparing the scalability of 2PL partitioning (see Figure 5a), with the original solution (see Figure 5b) shows that the scalability significantly increases when using 32 or more threads with 2PL partitioning. The original solution seems to scale only up to 16 threads. Figure 5c shows the same measurements using ScalaSTM.



(a) The effect of the number of warehouses on scalability using the lazy TPC-C benchmark with partitioning.



(b) The effect of the number of warehouses on scalability using the lazy TPC-C benchmark without partitioning.



(c) The effect of the number of warehouses on scalability using the TPC-C benchmark implemented with ScalaSTM.

Figure 5: Result of the TPC-C benchmarks

Partitioning with 2PL provides an improved scalability if the database has 20 or more warehouses compared to ScalaSTM.

7.3 Discussion

Note that when using one warehouse, the scalability is worse compared to the original solution. This is because when forcing evaluation, the most recent version is taken from the database which can lead to forcing the evaluation of other computations that have been added as well. When there is one warehouse, this warehouse will always be forced to evaluate. As a result, more communication between processors is needed thus it leads to a reduced throughput.

Note the drop in performance after 64 threads in Figure 5a. This is due to the fact that the experiments have been run on a computer with 64 cores and it will have to start context-switching when more than 64 threads are used.

Figure 5a shows that when using more warehouses, the scalability improves. This is due to the fact that the number of committed transactions stays the same, which are then divided over more partitions. This results in the root contention being divided over more partitions as well. Moreover, lock contention will reduce when the same amount of transactions are divided over more warehouses. Thus, this shows that using more partitions results in higher scalability.

8. DISCUSSION

Our hypothesis was that reducing root contention would improve scalability. Our experiments show that scalability improves when using more warehouses. This is as expected, since using more warehouses result in more partitions over which the transactions are divided thus root contention is reduced. The other expected result was that the solution would provide improved throughput. Our experiments show that throughput was improved when using more partitions and more operations per transaction. This is as expected as the transactions need to be pushed down fewer levels in the tree-structure.

One should however take into account that random data was used for the micro-benchmarks and TPC-C benchmarks. It has not been tested how this solution performs when using skewed random data, a situation that is not very rare in databases.

In this research, testing was limited to 2PL, and splitting the data structure at warehouse level. Using smaller trees, splitting below warehouse level, can result in a better performance as using smaller trees likely results in less contention. Using smaller trees would result in more partitions over which the root contention is divided thus root contention would be reduced, assuming that not all transactions need to commit at the same partition. Therefore one would expect better throughput when using more partitions. This is supported by the experiments which show that the throughput improves when partitions are smaller. However one should take into account that when updating more partitions, the overhead of 2PL becomes greater.

9. CONCLUSIONS AND FUTURE WORK

A mechanism to solve root contention for lazy tree-structured databases has been presented as well as how an implementation can be made. This mechanism splits the data structure in multiple partitions in order to divide the root contention over the multiple partitions. The experiments show that splitting the data structure, using 2PL as a concurrency control mechanism to ensure atomicity, provides improved scalability compared to the original solution as well as ScalaSTM. This scalability improves when using a more partitions.

9.1 Future Work

Micro-benchmarks have been run using 2PL as a concurrency control mechanism. However another mechanism, which has not been considered, may provide improved scalability and throughput.

Also, in the micro-benchmarks only updates have been used. It would be interesting to also investigate the effect of read transactions.

Another option that could be investigated, depends on the division of the workload. In these micro-benchmarks all data was chosen at random, however databases may have certain values which are often used. It would therefore be interesting to investigate the consequences committing a large number of the transactions of a specific value, district or warehouse. This may represent a situation that is more realistic for some databases.

Another solution for solving root contention could be to combine several small transactions into one larger transaction. When combining multiple transactions into one transaction, only one transaction needs to be pushed down from the root. This option did not seem favourable as a moment needs to be found when transactions are combined. One option is to wait until x transactions have been committed and then combining them. However a possible scenario would be that $x - 1$ transactions have been committed and it could take a long time until the last transaction has been committed. Another possibility is to use a timer and to combine all transactions that fall within this time zone. It would be interesting to see if combining transactions is favourable and how this can best be achieved.

10. REFERENCES

[1] G. Argo, J. Hughes, P. Trinder, J. Fairbairn, and J. Launchbury. Implementing functional databases. In *Proceedings of the Workshop on Database Programming Languages*, pages 87–103, Roscoff, France, Sept. 1987.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[3] École Polytechnique Fédérale de Lausanne. The Scala Programming Language. Retrieved from `http://www.scala-lang.org/`, n.d. Accessed: 20-06-2016.

[4] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 15–26, New York, NY, USA, 2014. ACM.

[5] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[6] HammerDB. Introduction to Transactional (TPC-C) Testing for all Databases. Retrieved from `http://www.hammerdb.com/hammerdb_transactionintro.pdf`, n.d. Accessed: 14-06-2016.

[7] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[8] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, Broomfield, CO, Oct. 2014. USENIX Association.

[9] F. Raab, W. Kohler, and A. Shah. Overview of the TPC-C Benchmark The Order-Entry Benchmark. Retrieved from `http://www.tpc.org/tpcc/detail.asp`, n.d. Accessed: 15-06-2016.

[10] L. Wevers, M. Huisman, and M. van Keulen. Lazy Evaluation for Concurrent OLTP and Bulk Transactions. 2016. To appear in *proceedings of the 18th International Database Engineering & Applications Symposium*.

[11] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 27–38, New York, NY, USA, 2011. ACM.