Modelling Database Lock-Contention in Architecture-level Performance Simulation

Philipp Merkle
Chair for Software Design and Quality
Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
merkle@kit.edu

Christian Stier
Chair for Software Design and Quality
Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
christian.stier2@student.kit.edu

ABSTRACT

Databases are the origin of many performance problems found in transactional information systems. Performance suffers especially when databases employ locking to isolate concurrent transactions. Software performance models therefore need to reflect lock contention in order to be a credible source for guiding design decisions. We propose a hybrid simulation approach that integrates a novel locking model into the Palladio software architecture performance simulator. Our model operates on a row level and is tailored to be used with architecture-level performance models. An experimental evaluation leads to promising results close to the measured performance.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modelling techniques

Keywords

Database; Lock Contention; Performance Prediction; Simulation; Palladio Component Model

1. INTRODUCTION

Relational database management systems (DBMS) are an integral part of many business information systems. They encapsulate established practices from decades of research, thereby hiding functional complexity from developers. Their complex performance behaviour, however, can neither be hidden from clients, nor be easily understood by developers. This dilemma gave rise to model-based approaches that seek to explain different aspects of database performance (cf. [10]). Database performance models can help software engineers to proactively evaluate the performance impact of design alternatives before they are translated into database schemas, queries, or source code in general. Ideally, the underlying methodology encapsulates knowledge of DBMS performance behaviour, thus relieving software developers from understanding performance-influencing subtleties of a database, its configuration or its workload.

In contrast to database performance models, architecture-centric performance models (e.g. [1]) provide decision support on a higher level of granularity, such as component composition and deployment. If database performance, however, dominates the overall system performance, architecture models must also capture database-related performance factors to achieve a sufficient accuracy. This is especially true for software systems that employ locking-based databases. Locking ensures proper isolation of concurrent transactions, so that inconsistencies due to conflicting data accesses (e.g. reading and writing at the same time) are prevented. Contention for locks on database items can severely impact system performance. This is why we address lock contention as a first step toward a performance modelling method for database-intensive software systems as motivated in our earlier work [5].

The contribution of this work-in-progress paper is an approach to modelling and simulation of database lock contention within architecture-level software performance simulation. First, we suggest a hybrid simulation model of row-level two-phase locking (2PL) tailored to architecture-level simulation. Second, we integrate our locking model into the Palladio software architecture simulator [1]. Third, we present an experimental evaluation that compares simulation results to measurements from a MySQL database.

2. SIMULATING LOCK CONTENTION

The transaction manager (TM) is responsible for lock management in a DBMS. It decides for arriving transactions if requested locks may be granted or must be denied to preserve the demanded isolation level. In the following, we propose a performance model of a TM. It utilises a model we establish for determining conflicts between transactions. This model is referred to as *conflict model* throughout this paper and will be introduced stepwise. Initially, we ignore shared locking to establish the fundamentals of the conflict model. On this basis, Sec. 2.3 outlines our extensions to shared locking.

2.1 Model Assumptions

In our model, a table is characterised solely by the number of contained rows. Rows neither have an identity nor any other distinguishing feature—all appear the same. Transactions are sequences of data access operations and end either with a commit or an abort. An access operation refers to a single table and is characterised by the number of accessed

ICPE'14, March 22–26, 2014, Dublin, Ireland.

ACM 978-1-4503-2733-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2568088.2576762.

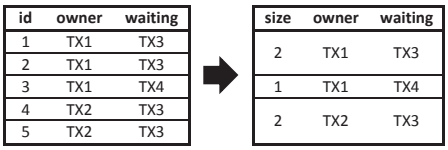


Figure 1: Row-level lock information (left) translated to conflict objects without row identities (right)

tuples and by the access type (read or write). A transaction submitting an access operation claims all rows at once. In case a lock request has to be denied due to conflicts, the transaction must wait for the lock to become available. Locks acquired by a transaction are only released once the transaction has been committed or aborted. This resembles rigorous 2PL. Table accesses are uniformly distributed, meaning every row is equally likely to be locked. Note that access hot-spots can still be modelled as is discussed in [12]. Finally, tables do not change in size over the course of an analysis.

2.2 Conflict Model (Exclusive Locks)

Our conflict model is inspired by the work of Morris and Wong [7], namely by their use of hypergeometric distributions to determine the probability of lock conflicts under 2PL. In general, a hypergeometric distribution yields the probability to select i balls of one type from an urn containing two types of balls. For simplicity, balls of one type are often referred to as *successes*, and one is then interested in the probability for selecting i successes out of the population. The hypergeometric distribution as applied in [7] takes as parameters the table size, the number of rows already locked, and the number of requested rows. These parameters provide a good level of abstraction for architectural modelling of database conflicts where knowledge of query-specific data distributions and access patterns should not be taken for granted.

Morris and Wong abstract from individual tables and assume that a database consists of a single table. Every transaction accesses the same number of tuples. The number of accessed tuples needs to be known at the time a transaction starts. Shared and exclusive accesses aren't distinguished. More importantly, however, the model of Morris and Wong does not take into account lock dependencies between conflicting transactions. This means once a transaction ends, every waiting transaction is equally likely to be continued, regardless of its arrival time or its actual conflicts.

Especially the last assumption hampers the applicability of their conflict model in software performance simulators such as Palladio. At simulation runtime, system requests have an identity that they inherit to their transactions. Throughout simulation, each individual request is tracked in order to collect corresponding performance measures. This can reveal, for example, a system design where two components mutually degrade their performance by frequently locking the same data items. By contrast, if conflict dependencies are neglected, requests may regularly continue their operation before their lock requests have actually been granted. We therefore extend Morris and Wong's stochastic conflict model by bookkeeping of conflicts.

The conflict bookkeeping keeps track of lock ownership and blocked transactions. This is similar to lock tables known from ordinary TMs. A lock table maps database elements (e.g. rows) to lock information about this element,

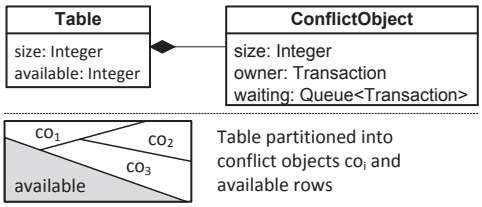


Figure 2: Bookkeeping of lock ownership and conflicts

including the current lock owner and waiting transactions organised as a queue [4]. While lock tables usually maintain an entry for each locked row, this fine-grained bookkeeping of locks does not fit well in our case, where rows do not have identities. We therefore abstract from individual rows using *conflict objects*. A conflict object subsumes a set of locked rows that share the same owner and the same waiting transactions (cf. Fig. 1). It is characterised by the number of rows it represents, by its owning transaction, and by the queue of transactions waiting for the conflict object—or more precisely, the represented rows—to be released. Note that conflict objects never overlap; they partition the locked rows of a table into disjoint subsets (cf. Fig. 2). Despite its name, a conflict object must not be involved in a lock conflict; the waiting queue remains empty until a conflict occurs.

Once a transaction requests access to one or more rows, the TM must determine conflicts with other active transactions. Only if the conflict size is zero, the transaction may proceed. Otherwise, it is blocked. The procedure we use for this is shown in Alg. 1. Depending on the access size, the table size, and the number of locked rows, we draw a series of samples from hypergeometric distributions. Each draw determines the overlap either with an existing conflict object or with the set of available (unlocked) rows. Step 1 calculates the overlap between requested and available rows. Access to these rows can be granted immediately. All remaining requested rows must be involved in a lock conflict. Step 2 provides the cause of each lock conflict by calculating the overlap of denied lock requests with existing conflict objects. If the overlap is larger than zero (i.e. there is a conflict), the requesting transaction needs to be enqueued as a waiting transaction—but only for a subset of the conflict object if the overlap does not involve the entire conflict object. In such a case, we *split* the conflict object into two conflict objects with the same owner and waiting transactions. Their size sums up to the size of the original conflict object. The requesting transaction can then be enqueued with the newly created conflict object representing the discovered conflict. Step 2 is repeated until all denied lock requests have been attributed to an existing conflict object. If the algorithm was able to grant all lock requests, the transaction may proceed. Otherwise, it is blocked and must wait.

Once a transaction commits or aborts, all held locks are released. If no transaction waits for the released conflict object, it is destroyed and the number of available rows is increased accordingly. Otherwise, the simulator selects the new owner from the queue of waiting transactions.

2.3 Conflict Model (Shared Locks)

So far we assumed exclusive access to data items. To support shared locks, we distinguish between shared and exclusive conflict objects. Shared conflict objects may have multiple owners. For each transaction, we record its demanded access type (shared vs. exclusive) in relation to a

Algorithm 1: `access` operation of transaction tx on a table t with k rows to be accessed

Input : t : *Table*, tx : *Transaction*, k : *Integer*
Output: blocked : *Boolean*

```
   // Step 1: Determine overlap with available rows
1  grantedLocks : Integer ← draw sample from
   hypergeometric distribution with population ← t.size,
   successes ← t.available, sample ← k
2  tx.ownedLocks.add(new ConflictObject of size
   grantedLocks))
3  t.available ← t.available − grantedLocks
   // Step 2: Determine overlap with conflict objects
4  remainingConflicts : Integer ← k − grantedLocks
5  conflictCandidates : Integer ← t.size − t.available
6  foreach co ∈ t.conflictObjects do
7      conflictSize ← draw sample from hypergeometric
       distribution with population ← conflictCandidates,
       successes ← co.size, sample ← remainingConflicts
8      if conflictSize = 0 then continue
9      if co.owner ≠ tx then
10         if conflictSize = co.size then
11             co.waiting.enqueue(tx)
12         else
13             split ← clone co
14             split.size ← conflictSize
15             co.size ← co.size − conflictSize
               split.waiting.enqueue(tx)
16     else
17         grantedLocks ← grantedLocks + conflictSize
18     conflictCandidates ← conflictCandidates − co.size
19     remainingConflicts ← remainingConflicts − conflictSize
20 return grantedLocks ≠ k
```

conflict object. This applies to transactions that own a conflict object, as well as to transactions waiting in the queue of a conflict object. A lock is only granted if the access type is compatible to the conflict object's type. This is referred to as lock compatibility (cf. [4]). A modified access algorithm that respects lock compatibility is discussed in detail in [11].

3. INTEGRATION INTO PALLADIO

In order to enable system-level QoS-analysis of transactional software systems, we integrate our work into the Palladio approach [1]. Palladio provides modelling and simulation capabilities for QoS-analysis of component-based software systems. Such a system can be modelled using the Palladio Component Model (PCM), a domain-specific language based on EMF[1]. A PCM instance includes the specification of components in terms of their performance-related behaviour, their assembly, and their deployment. Typical use cases of the modelled system are described in usage profiles. A PCM instance serves not only documentation purposes, but can especially be used for software quality simulation. Our integration encompasses metamodel extensions to the PCM, as well as extensions to EventSim [6], a discrete-event simulator for PCM instances.

Our metamodel can be seen in Fig. 3. All shown superclasses are imported from PCM. Component behaviour in the PCM is modelled by means of resource-demanding service-effect specifications (RDSEFFs). Similar to UML activity diagrams, an RDSEFF comprises linked actions. Unlike UML, however, the actions in an RDSEFF form a chain. Control flow constructs, such as branches, are modelled with nested RDSEFFs (ResourceDemandingBehaviour), one for each branch transition in our example. Similarly, we model a transaction (TransactionAction) as a nested RDSEFF (TransactionActionBehaviour). It encapsulates operations to be performed in the transaction's scope. Besides regular actions, transactions may contain DataAccessActions, which represent READ or WRITE access to one or more entities (e.g. tables). The accessSize attribute refers to the number of accessed data items (e.g. rows). It is characterised as a PCMRandomVariable, which can be a constant, a variable, a random variable, or a combination thereof.

DataEntities (e.g. customers or invoices) are declared in the repository model. Their characterisation is outsourced to the entitymapping model. This separation is mainly motivated by the separation of developer roles in PCM. Component developers that maintain the repository should not be forced to characterise entities in terms of their table size (cardinality) or their row size (bytesPerInstance). Furthermore, speaking of entities leaves open the decision in favour of or against a relational data persistence technology.

Instances of the extended PCM can be fed into EventSim, which maps actions of the type TransactionAction and DataAccessAction to calls to the coupled TM simulator. The coupling ensures that requests in EventSim are blocked until acquired locks are actually granted, thereby reflecting the effects of concurrency control.

4. EXPERIMENTAL EVALUATION

We evaluated our approach by comparing simulation results to measurements from a database. We chose MySQL 5.6.14 in combination with the InnoDB storage engine. The isolation level has been set to *serializable* so that InnoDB uses 2PL. All experiments and measurements were conducted through OLTP-Bench [3]—a benchmark suite for OLTP database workloads. We used a modified subset of the *Resource Stresser* benchmark that models a lock contention scenario. It comprises a single table updated by multiple transactions in parallel. Each transaction first updates a set of m rows within a single query. Affected rows are drawn from a discrete uniform distribution without replacement. Then, the transaction sleeps for a second before it commits. Multiple instances of this transaction are executed in a closed workload of size k.

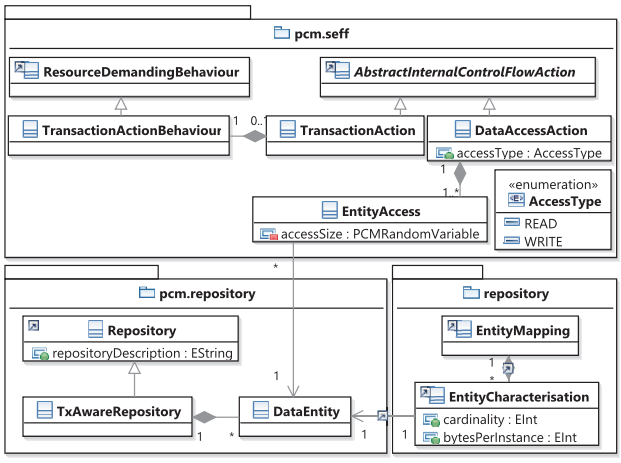
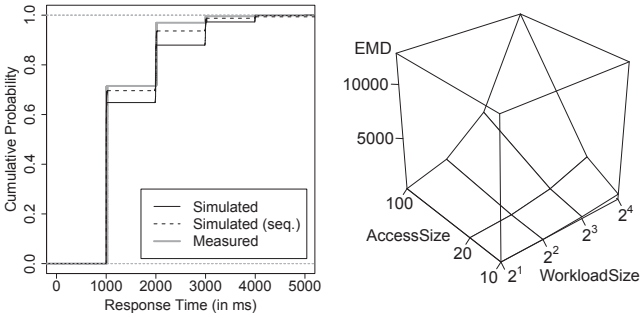


Figure 3: PCM metamodel extensions

[1]http://www.eclipse.org/modeling/emf/



(a) ECDF for workload size $k = 8$, access size $m = 10$ (b) Earth mover's distance predicted vs. measured results

Figure 4: Measurements compared to simulation results

In our setting, the table holds 1024 entries. The workload size k takes the values (2, 4, 8, 16); the access size m takes the values (10, 20, 100). For each combination of k and m, we measured the response time of approximately 1000 transactions. In addition, each experiment has been modelled and simulated with our Palladio extension. The earth mover's distance (Fig. 4b) between simulation results and measurements indicates that simulation accuracy suffers from highly contended tables. Results for low to moderate contention, however, appear promising. Results for a moderate contention scenario ($k = 8$, $m = 10$) are therefore compared in Fig. 4a (ignore for the moment the sequential simulation). The observed differences lead us to further analyses of InnoDB's locking behaviour. From the INNODB_LOCKS information schema, it becomes apparent that locks for a query are not requested all at once. Rather, a transaction in InnoDB blocks upon the first conflict and does not request remaining locks before this conflict is resolved. This sequential locking policy leads to lower lock contention compared with our conflict model. To emulate sequential locking in PCM, we split the DataAccessAction representing the update of m rows into a sequence of m sequential DataAccessActions, each with $m = 1$. The results in Fig. 4a for the sequential simulation are now close to the measurements. Remaining differences can mostly be explained by the increased deadlock risk caused by the model adjustments. Being prone to deadlocks and due to its increased modelling and computational complexity, the sequential modelling is impractical. Still, the measurements suggest that our simulated conflict model is a valid performance model of 2PL.

5. RELATED WORK

A multitude of approaches has been developed to evaluate database performance. A recent survey by Osman and Knottenbelt [10] provides an extensive comparison of queuing network (QN) based approaches. While QNs are well-suited for modelling database-induced hardware contention, they lack expressiveness to capture lock-contention sufficiently. Coulden et al. [2] therefore propose to use queueing Petri nets (QPN) for modelling table-level 2PL. QPNs combine the strengths of Petri nets and QNs, thus allowing for representing both software and hardware contention. Based on [2], Osman et al. [9] present their efforts toward row-level 2PL. The QPNs in [2] and [9] reflect the database schema but abstract from the system architecture. Recently, Mozafari et al. [8] introduced DBSeer, a holistic approach to database performance evaluation based on statistical models. DBSeer aims at supporting database administrators while our work targets software engineers from early development stages on.

6. CONCLUSION

This paper presented our work towards performance simulation of database-intensive software systems. We proposed a novel locking model that resembles the performance behaviour of 2PL sufficiently. Its integration with Palladio's software architecture simulation enables software engineers to evaluate transaction-related design decisions on a model basis. So far high contention scenarios cannot be sufficiently simulated: First, conflict objects might degenerate due to continuous split operations until their size reaches one, leading to a computationally expensive access algorithm. Second, simulation accuracy suffers from high contention. Reducing these limitations is subject to future work. Our next goal is to reflect different isolation levels in the simulation.

7. REFERENCES

[1] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *The Journal of Systems and Software*, 82:3–22, 2009.

[2] D. Coulden, R. Osman, and W. J. Knottenbelt. Performance modelling of database contention using queueing Petri nets. In *Proceedings of the International Conference on Performance Engineering*, 2013.

[3] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4), 2013.

[4] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database system implementation*. Prentice Hall, 2000.

[5] P. Merkle. Predicting transaction quality for balanced data consistency and performance. In *Proceedings of the International Doctoral Symposium on Components and Architecture*, 2013.

[6] P. Merkle and J. Henss. EventSim – an event-driven Palladio software architecture simulator. Karlsruhe Reports in Informatics 32, KIT, 2011.

[7] R. Morris and W. Wong. Performance analysis of locking and optimistic concurrency control algorithms. *Performance Evaluation*, 5:105–118, 1985.

[8] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the SIGMOD International Conference on Management of Data*, 2013.

[9] R. Osman, D. Coulden, and W. J. Knottenbelt. Performance modelling of concurrency control schemes for relational databases. In *Analytical and Stochastic Modeling Techniques and Applications*. Springer, 2013.

[10] R. Osman and W. J. Knottenbelt. Database system performance evaluation models: A survey. *Performance Evaluation*, 69(10):471 – 493, 2012.

[11] C. Stier. Transaction-aware software performance prediction. Master's thesis, KIT, 2014.

[12] Y. C. Tay, R. Suri, and N. Goodman. A mean value performance model for locking in databases: The no-waiting case. *Journal of the ACM*, 32:618–651, 1985.