A Performance Model of Multi-Version Concurrency Control

Pierangelo Di Sanzo, Bruno Ciciani and Francesco Quaglia
Sapienza, Università di Roma

Paolo Romano
INESC-ID

Abstract

In this article we present a performance model for Multi-Version Concurrency Control (MVCC). This type of concurrency control is currently very popular among mainstream commercial and open source database systems thanks to its ability to well cope with read intensive workloads, as in the case of transaction profiles proper of Web applications. To build the model we had to tackle the intrinsic higher complexity of MVCC when compared to traditional concurrency control mechanisms (i.e. 2-Phase-Locking and optimistic ones), such as the joint use of locks and aborts to resolve direct conflicts among write accesses to the same data item, and the management of multiple data versions. We validate our analytical model via an extensive simulation study, considering both uniform and skewed data accesses, as well as differentiated transaction profiles. To the best of our knowledge, the present study provides the first analytical model of MVCC.

1 Introduction

Database systems, and more in general transactional systems, are recognized as core software components for a wide spectrum of applications. These span from traditional client/server applications, to more modern multi-tier ones (e.g. Web-based e-business applications). As a consequence, adequate tuning and configuration of the database system is still a fundamental issue to address.

One of the most critical aspects to be tackled when evaluating the performance of database systems lies in capturing the effects of the employed concurrency control mechanism, which regulates concurrent data access in order to ensure desired isolation levels. Concerning this point, several maintstream proprietary (such as Oracle Database) and open source (such as PostgreSQL) database systems currently rely on Multi-Version Concurrency Control (MVCC) algorithms [9]. These algorithms exploit previous versions of data items in order to improve the level of concurrency among transactions. This approach revels particularly attractive in read intensive environments, where a read access (that would otherwise be delayed or aborted in case of conflict) can be immediately served via a previous version of that same data item. Read intensive workloads have been shown to be representative of several Web-based applications, which is one reason for the increasing diffusion of such a type of concurrency control algorithms.

In this article we provide a complete analytical model capturing the performance of the most diffused MVCC algorithm, namely the one specifically oriented to guarantee the so called *snapshot-isolation* consistency criterium [8]. Although not providing serializability guarantees (while ensuring repeatable reads), this isolation level is considered acceptable for a wide set of applicative contexts. Also, several recent works have provided formal frameworks for the identification/generation of classes of applications where this type of isolation level suffices to ensure serializability [12], or for detecting (and correcting) applications potentially exposed to non-serializable execution histories [15].

Some literature works have addressed the evaluation of the performance of MVCC algorithms. However, these works are mostly based on simulative approaches [11]. Hence, to the best of our knowledge, our proposal is the first analytical model describing the performance of snapshot-isolation MVCC algorithms. Actually, some analytical results for MVCC have been provided in [1, 23]. However, different from our approach, the objective of those studies is to provide an analysis of the storage cost for maintaining data item versions (vs the data update frequency), and not to provide an analytical expression for transaction execution latency and related system throughput. Also, the level of abstraction considered in our analysis makes the model valid independently of the real policy adopted by the database system to retrieve data item versions (e.g. explicit storing [20] vs dynamic regeneration of the required version via rollback segments [17]). This makes the model suited for a variety of implementations for version management mechanisms inside the database.

We validate our analytical model via an extended simulation study relying on synthetic workload descriptions (e.g. in terms of machine instructions for specific transaction operations) analogous to those used for the validation of analytical models describing the performance of other types of concurrency control mechanisms, namely 2-Phase-Locking and Optimistic [25].

The remainder of this paper is structured as follows. In Section 2 we discuss literature works on database modeling/evaluation, which are closely related to our analy-

sis. The performance model for snapshot-isolation oriented MVCC is provided in Section 3. Finally, the model validation is presented in Section 4.

2 Related Work

A large volume of research results exist in literature, which cope with the evaluation of database systems and concurrency control algorithms. However, most of them are focused on the evaluation of lock-based and optimistic approaches for conflict management among concurrent transactions. For these classical concurrency control strategies, analytical results have been presented, e.g., in [14, 21, 22] for the case of centralized database systems, and in [6, 7] for the case of distributed/replicated databases. In [25] a general methodology for modeling and analytical evaluation of both centralized and distributed systems is provided, still coping with lock-based and optimistic concurrency control schemes.

For what concerns simulative studies, locking protocols, and their impact on performance, have been extensively addressed by several works [3, 18, 19]. On the other hand, the work in [2] provides simulative studies specifically aimed at evaluating optimistic concurrency control, and at comparing this type of concurrency control with locking strategies. One of the main findings of this work is probably that the amount of computing power is a fundamental aspect determining which of the two concurrency control approaches has the chance to provide better performance levels.

For what concerns MVCC strategies, complete performance studies in literature are exclusively based on simulation results [11]. In fact, analytical models have been proposed exclusively for the evaluation of storage management tradeoffs vs the data item update frequency [1, 23]. These are oriented to the evaluation/prediction of space occupancy for the different versions of the data items under specific data access patterns (in order to provide facilities for storage size planning). Compared to these studies, we tackle the different issue of providing a complete performance model of MVCC (in the form of snapshot-isolation algorithms) that allows transaction latency and system throughput analysis and prediction.

3 Performance Analysis of MVCC

3.1 An Overview of MVCC Protocols Ensuring Snapshot-Isolation

In typical, pragmatical implementations, MVCC [8] protocols ensuring snapshot-isolation can be considered as a middle ground between locking and optimistic concurrency control [20]. In fact, some form of locking is used, however it involves only exclusive locks associated with write accesses to data items.

With snapshot-isolation, each transaction is associated with a so called *Start-Timestamp*, whose value is set upon the first data access operation executed by the transaction. This value is used to determine the set of transactions that are concurrent with T. In particular, this set is formed by the transactions that are active when Start-Timestamp is set for T, plus the transactions with timestamp grater than Start-Timestamp.

When a transaction T tries to write a data item x that has not yet been accessed by this same transaction, *version check* is performed to determine whether no concurrent transaction that wrote x has already been committed. In the positive case, version check is said to have failed, and T is immediately aborted. Otherwise, T tries to acquire an exclusive write lock on x, which can lead to a wait phase in case the lock is currently held by any other active transaction T'. In the latter case, if T' is eventually committed, then T gets aborted in order to avoid the so called *lost update phenomenon* [8]. Upon lock acquisition, T is allowed to create a new version of x.

If T wants to read/write a data item x previously written during its execution, the version of x just created by T is immediately supplied. Instead, a read operation on a data item x not previously written by T is served by accessing the version of x that has been committed by the most recent transaction not concurrent with T. In this way all read operations are never blocked and do not cause transaction abort.

When T commits or aborts, all the acquired write locks are released. In case of commit, all the data item versions created by T become visible.

3.2 Modeling Assumptions

We consider an open system in which transactions arrive according to a Poisson Process with mean value λ. Compared to closed system approaches (e.g. [14]), the open approach is more suited for scenarios with larger user population, like in, e.g., Web-based applications.

Begin, write and commit operations are assumed to require a mean number of CPU instructions denoted with I_b, I_w and I_c, respectively. CPU instructions to support read accesses are modeled in a slightly more complex way, as a reflection of the fact that a read access can require traversing the history of data item versions to retrieve the correct one. This is modeled by assuming for a read access a baseline of I_r^F CPU instructions, plus I_r^V CPU instructions for each traversed version. In the case of transaction abort, we assume the execution of a mean number of I_a CPU instructions. Also, the transaction is rerun after a randomly distributed back-off time with mean value $T_{backoff}$.

Each disk access, for serving buffer misses for specific data items, is assumed to require a fixed latency $t_{I/O}$. Although this is done for simplicity of model construction, given the optimized algorithms used by operating system kernels to schedule disk accesses, it is commonly accepted that the disk delay can be well captured via a constant value

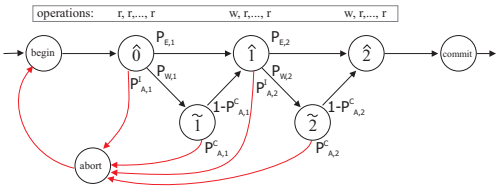


Figure 1. Base Transaction Execution Model.

expressing the average latency in such an optimized scenario [24, 25].

We do not explicitly model deadlocks and related transaction abort/restart since, as already shown by other studies (see, e.g., [5, 13]), these affects have no significative impact on performance. Given that those studies deal with 2-Phase-Locking, the previous assumption reveals even more realistic in case of MVCC since it does not use read locks, hence further reducing the deadlock probability.

The CPU is modeled as an M/M/k queue, where k is the number of CPUs, each of which is assumed to have a processing speed denoted as $MIPS$.

We first present a basic version of the analytical model, relying on the following additional approximations and assumptions: (1) transactions belong to a unique class with a mean number of N_w write and N_r read operations, (2) transactions perform accesses uniformly distributed over the whole set of D data items within database. Both these assumptions will be then removed while presenting an extended version of the analytical model.

Also, as in previous concurrency control evaluation studies [14, 22, 25], we assume the system is stable and ergodic, so that quantities like the contention probability and the mean transaction response time exist and are finite, and defined to be either long-run averages or steady-state quantities.

Finally, as also assumed in existing performance models of database concurrency controls [24, 25, 22], we consider an interleaving of read/write operations by a single transaction such that the N_r reads are uniformly mixed with the N_w writes.

3.3 Basic Analytical Model

3.3.1 Transaction Execution Model

The execution of a transaction is modeled through a directed graph. Figure 1 shows an example for a transaction with $N_w = 2$. Each node represents the state of a transaction, corresponding to a specific phase for the execution of the transaction.

The label of an arc from a node p to a node q represents the transition probability from state p to state q. If the label is omitted, than the transition probability is intended to be 1. Obviously, the sum of all transition probability values for outgoing arcs from a node must be 1.

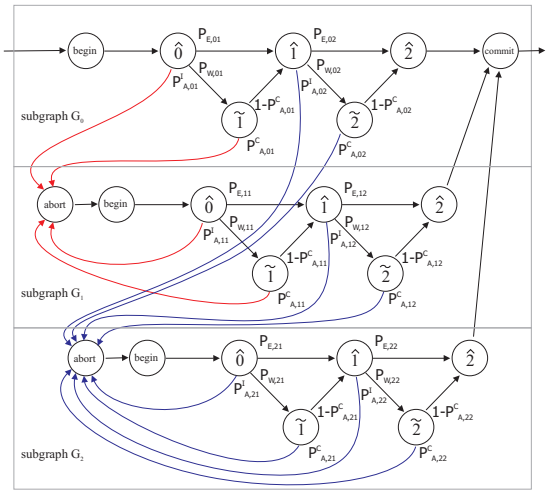


Figure 2. Modified Transaction Execution Model.

The states labelled with *begin*, *commit* and *abort* are used to model the execution of the respective operations. Instead, for what concerns read/write accesses to data items, we use a different state labelling approach to denote the corresponding phases. Considering that the sequence of N_r read operations performed by a transaction is uniformly distributed across the N_w write operations, on average we will have a write access to a data item after executing $N_r^S = N_r/(N_w + 1)$ read operations (see Figure 1). According to this rule, state $\hat{0}$ represents the phase in which the initial N_r^S read operations are performed before the first write access, and states $\hat{i}$ (with $1 \leq i \leq N_w$) represent phases in which a write operation has been issued, followed by a mean number of N_r^S read operations.

According to the MVCC description provided in Section 3.1, when a write operation needs to be carried out, version check is performed. If version check for the i-th write fails, the transaction is aborted. The corresponding state transition probability is denoted as $P_{A,i}^I$. (The related arc starts from state $\widehat{i-1}$ and ends to state *abort*.) On the other hand, if version check succeeds (this occurs with probability $1 - P_{A,i}^I$) a wait phase for lock acquisition occurs with probability P_{cont}, corresponding to the probability that an exclusive write lock is being held by another transaction.

Note that, by assumption (2) in Section 3.2, P_{cont} is independent of the accessed data item. Thus, the probability of transition from state $\widehat{i-1}$ to state $\tilde{i}$ can be expressed as $P_{w,i} = (1 - P_{A,i}^I)P_{cont}$. On the other hand, the probability that a lock is immediately granted after version check is $1 - P_{cont}$. Thus, the probability of transition from state $\widehat{i-1}$ to state $\hat{i}$ is $P_{E,i} = (1 - P_{A,i}^I)(1 - P_{cont})$.

A transaction in a waiting state $\tilde{i}$ gets aborted with prob-

ability $P^C_{A,i}$, which we will subsequently evaluate.

When a read/write operation is executed, the accessed data item might be already available into the buffer pool, otherwise a disk access is needed. We denote with P_{BH1} the expected buffer hit probability. However, as suggested in [25], in order to provide a more accurate evaluation of the effects of buffer hits in case of transaction restart after an abort, a different value of the expected buffer hit probability P_{BH2} is considered when, in a rerun, the transaction accesses a data item already accessed prior to the abort. Both P_{BH1} and P_{BH2} are intended as input parameter for our model, whose value will reflect specific choices for what concern buffer pool size and related replacement policies.

According to the previous considerations, the graph modeling transaction execution is extended as in Figure 2. Specifically, the graph is partitioned in $N_w + 1$ subgraphs $G_0, G_1, \ldots, G_w$. Subgraph G_0 represents the first transaction run, for which we consider P_{BH1} as the buffer hit probability for all read/write operations. Subgraphs G_k (with $1 \leq k \leq N_w$) represent reruns of the transaction where data items accessed until the k-*th* write have already been accessed in a previous run. Hence, for these data items, we use P_{BH2} as the buffer hit probability, while P_{BH1} is used as the buffer hit probability for subsequent data accesses during the same run. For example, referring to Figure 2, if the transaction aborts in state $\tilde{1}$ of subgraph G_0, the subsequent run is represented by subgraph G_1, where P_{BH2} is the buffer hit probability for all read operations occurring up to the 1-*st* write.

In the extended graph, we use the subscript '$_{ki}$' to label arcs of subgraph G_k. Hence, we have $P_{W,ki} = (1 - P^I_{A,ki})P_{cont}$ and $P_{E,ki} = (1 - P^I_{A,ki})(1 - P_{cont})$. For a run associated with a generic subgraph G_k, we denote with $\hat{P}_k(i)$ the probability to reach state $\hat{i}$ (i.e. the transaction does not abort before). This probability value iteratively depends on the probability to reach state $\widehat{i-1}$, thus

$$\hat{P}_k(0) = 1,$$

$$\hat{P}_k(1) = \hat{P}_k(0)(1 - (P^I_{A,k1} + P_{W,k1}P^C_{A,k1})),$$

and, for a generic state $\hat{i}$,

$$\hat{P}_k(i) = \hat{P}_k(i-1)(1 - (P^I_{A,ki} + P_{W,ki}P^C_{A,ki})).$$

Note that, by construction, $\hat{P}_k(commit) = \hat{P}_k(N_w)$.

3.3.2 Transaction Response Time

Depending on the experienced aborts, successful completion of a transaction will require a number N of (re)runs. We denote with N_{G_k} the amount of those runs described by subgraph G_k (i.e. runs where data items accessed until the k-*th* write have already been accessed in a previous run). Note that $N_{G_0} = 1$ and, after some algebra, we obtain

$$N_{G_k} = \frac{1}{\hat{P}_k(k)} \sum_{j=0}^{k-1} N_{G_j} \hat{P}_j(k-i)(P^I_{A,jk} + P_{W,jk}P^C_{A,jk}).$$

We denote with R_{begin}, $R_{k\hat{i}}$, $R_{k\tilde{i}}$, R_{com} and R_{abt}, respectively, the mean residence time for states $begin$, $\hat{i}$, $\tilde{i}$, $commit$ and $abort$. Runs represented by subgraph G_k spend R_{begin} time in state $begin$, plus time in other states, according to the probability for these states to be reached. Hence we get

$$\hat{R}_{ki} = \hat{P}_k(i)R_{k\hat{i}},$$

$$\tilde{R}_{ki} = \hat{P}_k(i-1)P_{W,ki}R_{k\tilde{i}},$$

$$R_{k\,com} = \hat{P}_k(commit)R_{com},$$

$$R_{k\,abt} = (1 - \hat{P}_k(commit))R_{abt}.$$

State $\hat{0}$ of each subgraph is always visited in each run, thus $\hat{R}_{k0} = R_{k\hat{0}}$. Therefore, the mean time for a run represented by subgraph N_{G_k} is

$$R_{G_k} = R_{begin} + \hat{R}_{k0} + \sum_{i=1}^{N_w} (\hat{R}_{ki} + \tilde{R}_{ki}) + R_{k\,com} + R_{k\,abt.}$$

The mean transaction response time is

$$R_{tx} = \sum_{k=0}^{N_w} N_{G_k} R_{G_k}.$$

3.3.3 Lock Holding Time

A write lock is acquired when visiting each state $\hat{i}$ (with $1 \leq i \leq N_w$), and is released at end of the run. If the run terminates with transaction commit, then all its locks are released upon completion of the phase associated with the state $commit$. Instead, if the run terminates with transaction abort, then the locks are released upon entering the state $abort$. In other words, as in models for 2-Phase-Locking strategies [25], we consider lock release in case of abort as an instantaneous action, which does not contribute to lock holding time. Hence, locks are held by a transaction in the time interval between the acquisition and either the start of the abort phase, or the end of the commit phase. Using the expressions previously defined for the mean time spent in each state, the mean lock holding time for the i-*th* acquired lock in a generic run represented by subgraph G_k can be expressed as

$$T_{H,ki} = \sum_{j=i}^{N_w} \hat{R}_{ki} + \sum_{j=i+1}^{N_w} \tilde{R}_{ki} + R_{k\,com}.$$

Hence, the mean lock holding time for the i-*th* acquired lock, evaluated across all the (re)runs of the transaction, can be expressed as

$$T_{H,i} = \sum_{k=0}^{N_w} N_{G_k} T_{H,ki},$$

and the mean lock holding time is

$$T_H = \frac{1}{N_w} \sum_{i=1}^{N_w} T_{H,i}.$$

3.3.4 Lock Contention Probability

As already hinted, due to assumption (2) in Section 3.2, the probability of contention P_{cont} is uniform across all the data items, thus being independent of the specific accessed data. Given that transactions arrive according to a Poisson Process, the analysis in [25] for the case of 2-Phase-Locking still holds in our case. Hence, the probability of contention can be expressed as the expected data utilization factor, namely

$$P_{cont} = \frac{\lambda N_w T_H}{D}.$$

3.3.5 Lock Waiting Time

Now we evaluate $R_{k\tilde{i}}$, namely the average residence time in state $\tilde{i}$ of subgraph G_k. As for the analysis in [25], we consider the approximation in which at most one transaction is queued for write lock acquisition on whichever data item. In our case this approximation is further supported by the fact that, differently from 2-Phase-Locking, in MVCC if a transaction T' commits, then any transaction T waiting for a lock held by T' gets immediately aborted. Hence, if T' commits, T needs to wait for the completion of at most one transaction. We approximate $R_{k\tilde{i}}$ as the mean residual time required by T' to terminate the current run (with either commit or abort), evaluated at the time of conflict occurrence, namely when T enters state $R_{k\tilde{i}}$. The probability that, at the time of conflict occurrence, T' is executing a run modeled by subgraph G_k is

$$P_{cont,k} = \frac{N_{G_k} T_{H,k}^{Tot}}{T_H^{Tot}},$$

where

$$T_{H,k}^{Tot} = \sum_{i=1}^{N_w} T_{H,k,i},$$

and

$$T_H^{Tot} = \sum_{k=0}^{N_w} N_{G_k} T_{H,k}^{Tot}.$$

Thus, at conflict time, the probability values for T' to be in states $\hat{i}$ and $\tilde{i}$ (with $1 \leq i \leq N_w$) within subgraph G_k are

$$P_{cont,k\hat{i}} = \frac{\hat{R}_{ki}}{T_{H,k}^{Tot}} i,$$

$$P_{cont,k\tilde{i}} = \frac{\tilde{R}_{ki}}{T_{H,k}^{Tot}} (i-1),$$

and, finally, the probability for T' to be in state *commit* is

$$P_{cont,k\,com} = \frac{R_{k,com}}{T_{H,k}^{Tot}} N_w.$$

Now we introduce the conditional probability $\hat{P}_k(j|i)$ to reach state $\hat{j}$ during a (re)run associated with subgraph G_k, given that state $\hat{i}$ (with $i \leq j$) has already been reached during that same run. For $j = i$ we have

$$\hat{P}_k(j|i) = 1,$$

and, for $j > i$, we have the following iterative expression

$$\hat{P}_k(j|i) = \hat{P}_k(j-1|i)(1-(P_{A,k\,i+1}^{I} + P_{W,k,i+1} P_{A,k,i+1}^{C})).$$

If, at conflict time, T' was executing in state $\hat{i}$ (with $1 \leq i \leq N_w$), then we approximate the residual lock holding time as

$$\tilde{R}_{k\hat{i}} = \frac{\hat{R}_{k,i}}{2} + B_{k\hat{i}},$$

where we consider an average residual time for state $\hat{i}$ equal to half the permanence time in this same state, and where $B_{k\hat{i}}$ is the additional time to terminate the current run given that T' has reached state $\hat{i}$, that is

$$B_{k\hat{i}} = \sum_{j=i+1}^{N_w} \hat{P}_k(j|i)(R_{k\hat{i}} + P_{W,ki} R_{k\tilde{i}}) + \hat{P}_k(N_w|i) R_{com}.$$

Similarly, if at conflict time T' is executing in state $\tilde{i}$ (with $2 \leq i \leq N_w$) we have

$$\tilde{R}_{k\tilde{i}} = \frac{\tilde{R}}{2} + B_{k\tilde{i}},$$

where

$$B_{k\tilde{i}} = \sum_{j=i}^{N_w} \hat{P}_k(j|i) R_{k\hat{i}} + \sum_{j=i+1}^{N_w} \hat{P}_k(j|i)(P_{W,ki} R_{k\tilde{i}})$$

$$+ \hat{P}_k(N_w|i) R_{com}.$$

Finally, if at conflict time T' is executing in state $commit$, we have

$$\tilde{R}_{com} = \frac{R_{com}}{2}.$$

Overall, we express $R_{k\tilde{i}}$ as

$$R_{k\tilde{i}} = \sum_{k=0}^{N_w} P_{cont,k} \Big(\sum_{i=1}^{N_w} P_{cont,k\hat{i}} \tilde{R}_{k\hat{i}} + \sum_{i=2}^{N_w} P_{cont,k\tilde{i}} \tilde{R}_{k\tilde{i}} +$$

$$P_{cont,k\,com} \tilde{R}_{com} \Big).$$

3.3.6 Version Check Failure Probability

Version check for transaction T upon write access to data item x fails if a concurrent transaction wrote x and committed. Unless we have reached system saturation, the rate of commit events is equal to the transaction arrival rate λ. By approximating commit events occurrence as a Poisson Process, for assumption (2) in Section 3.2, we have that version check failure probability corresponds to the probability that the requested data has been updated by at least one concurrent transaction during the time period from the startup of transaction T and the data access instant. Hence, the version check failure probability $P^I_{A,ki}$ while performing the i-*th* write during an run modeled by subgraph G_k can be expressed as

$$P^I_{A,ki} = (1 - \exp(-\frac{\lambda N_w}{D}\vec{R}_{ki})),$$

where $\vec{R}_{ki}$ is time between the startup of T and version check occurrence. This time can be evaluated as

$$\vec{R}_{ki} = \sum_{j=0}^{i-1}(R_{k\hat{i}} + P_{W,ki}R_{k\,\widetilde{i+1}}).$$

3.3.7 Version Access Cost Model

Existing implementations of multiversion concurrency control rely on different approaches for the management of data item versions. Some products (e.g. Oracle Database [17]), explicitly store only the most recent committed data item versions, so to reduce space usage, and exploit the information stored in the DBMS log to reconstruct data pages when an older data item version is required. Instead, other products use explicit version storing (e.g. PostgreSQL [20]). Given that our aim is to provide an analytical model independent of specific implementation issues, we model the cost of a read operation as $I^F_r + I^V_r N^V_{read}$, where N^V_{read} is the number of backward traversed data item versions in order to retrieve the correct one. With this approach, further implementation dependent management costs (e.g. garbage collection cost) could be modeled as additional workload on hardware resources, which we neglect in the present analysis for simplicity.

For solving the previous read cost model, we now evaluate the average number of backward traversed versions for each read operation in state $\hat{i}$ of whichever subgraph G_k, namely $N^V_{read,ki}$. Given assumption (2) in Section 3.2, committed versions of a data item are born with an approximated rate $\sigma = \lambda N_w/D$. Denoting with $\Delta T_{s,ki}$ the time interval between transaction startup and the arrival in state $\hat{i}$ of subgraph G_k, we can then approximate $N^V_{read,ki}$ as

$$N^V_{read,ki} = \Delta T_{s,ki}\,\sigma.$$

Note that this value corresponds to the average number of versions committed during the time interval $\Delta T_{s,ki}$. Using $\vec{R}_{ki}$ previously introduced, we approximate $\Delta T_{s,ki}$ as

$$\Delta T_{s,ki} = \vec{R}_{ki} + R_{k\hat{i}}/2.$$

3.3.8 Hardware Resource Model

The CPU load (number of instructions) due to the execution of a run represented by subgraph G_k is

$$C_k = I_b + N^S_r(I^F_r + I^V_r N^V_{read,k\,0}) + I_{vc} +$$
$$+\hat{P}_k(i)\sum_{i=1}^{N_w-1}(I_w + N^S_r(I^F_r + I^V_r N^V_{read,ki}) + I_{vc}) +$$
$$+\hat{P}_k(N_w)(I_w + N^S_r(I^F_r + I^V_r N^V_{read,k\,N_w})) +$$
$$+\hat{P}_k(commit)I_c + (1 - \hat{P}_k(commit))I_a.$$

where we denote with I_{vc} the average number of CPU instructions to perform version check. Note that version check occurs in states $\hat{i}$ (with $0 \le i \le N_w - 1$). The CPU utilization can be expressed as

$$\rho = \frac{\lambda \sum_{k=0}^{N_w}(N_{G_k}C_k)}{k\ MIPS}$$

We denote with $p[queuing]$ the wait probability for CPU requests, which can be easily computed by leveraging classical queuing theory results on M/M/k queues [16]. Then, defining $\gamma = 1 + p[queuing]/(k(1-\rho))$, we can evaluate the average response time for each state of the graph as

$$R_b = \gamma\frac{I_b}{MIPS},$$
$$R_{com} = \gamma\frac{I_c}{MIPS},$$
$$R_{abt} = \gamma\frac{I_a}{MIPS},$$
$$R_{k\hat{i}} = \gamma\frac{N^S_r(I^F_r + I^V_r N^V_{read,ki}) + I_w + I_{vc}}{MIPS} + T_{IO}G_{ki}$$

where $I_w = 0$ for $i = 0$, $I_{vc} = 0$ for $i = N_w$, and G_{ki} is expressed as

$$G_{ki} = N^S_r P_{BH1}$$

for k=0 and i=0,

$$G_{ki} = N^S_r P_{BH2}$$

for $1 \le k \le N_w$ and $i = 0$,

$$G_{ki} = N^S_r P_{BH2} + P_{BH2}$$

for $1 \le k \le N_w$ and $i = k$,

$$G_{ki} = (N^S_r + 1)P_{BH2}$$

for $1 \le k \le N_w$ and $1 \le i < k$,

$$G_{ki} = (N^S_r + 1)P_{BH1}$$

for $1 \le k \le N_w$ and $k < i \le N_w$.

3.3.9 Numerical Resolution

The proposed model, analogously to, e.g., those in [6, 14, 25], can be solved via an iterative approach. Once assigned numerical values to all parameters described in Section 3.2 and to I_{vc}, P_{BH1} and P_{BH2}, and once the initial values of all other probabilities are set equal to 0, all model parameters can be evaluated via the provided equations, and can be used as the input for the next iteration. We have experimentally observed that, if the chosen initial values define a stable system, then the computation converges in a few iterations.

3.4 Extended Analytical Model

We provide in this section an extension of the model, which is able to handle both variable length transactions and non-uniform data access. In practice, this means removing assumptions (1) and (2) in Section 3.2.

3.4.1 Variable Length Transactions

We adopt a transaction clustering approach based on the average number of operations executed by transactions within a same class. Specifically, transactions with a similar number of read and write operations are grouped into a class C^{rw}, where r and w identify the corresponding number of expected reads and writes. Further we denote with R and W two sets of integers, which are used to list the average number of read and write operations of different classes. Thus, for each C^{rw}, $r \in R$ and $w \in W$. We denote with $X = \{(r, w)\}$ the set of all (r, w) pairs characterizing the workload, hence $|X|$ is the total number of classes. A transaction belongs to class C^{rw} with probability P^{rw}_{TC}, thus the average arrival rate of transactions of class C^{rw} is $\lambda^{rw} = \lambda P^{rw}_{TC}$. Now we redefine some parameters appearing in the basic model in order to capture the presence of transaction classes. To this end, we use the superscript '^{rw}' to denote the parameter redefinition for each class C^{rw}. We have

$$P^{rw}_{W,ki} = (1 - P^{I,rw}_{A,ki})P_{cont},$$

where $P^{I,rw}_{A,ki}$ is version check failure probability for a transaction of class C^{rw}, and

$$\hat{P}^{rw}_k(i) = \hat{P}^{rw}_k(i-1)(1 - (P^{I,rw}_{A,ki} + P^{rw}_{W,ki}P^{C}_{A,ki})).$$

The expected number of runs whose execution is represented by subgraph G_k for a transaction of class C^{rw} is

$$N^{rw}_{G_k} = \frac{1}{\hat{P}^{rw}_k(k)} \times$$

$$\times \sum_{j=0}^{k-1} N^{rw}_{G_j} \hat{P}^{rw}_j(k-i)(P^{I,rw}_{A,jk} + P^{rw}_{W,jk}P^{C}_{A,jk}).$$

The mean times spent in the different states by a transaction of class C^{rw} in a run are the following

$$\hat{R}^{rw}_{ki} = \hat{P}^{rw}_k(i)R^{rw}_{k\hat{i}},$$

$$\tilde{R}^{rw}_{ki} = \hat{P}^{rw}_k(i-1)P_{W,ki}R_{k\tilde{i}},$$

$$R^{rw}_{k\,com} = \hat{P}^{rw}_k(commit)R^{rw}_{com}$$

and

$$R^{rw}_{k\,abt} = (1 - \hat{P}^{rw}_k(commit))R^{rw}_{abt}.$$

The mean execution time for a run modeled by subgraph G_k is

$$R^{rw}_{G_k} = R_{begin} + \hat{R}^{rw}_{k0} + \sum_{i=1}^{w}(\hat{R}^{rw}_{ki} + \tilde{R}^{rw}_{ki}) + R^{rw}_{k\,com} + R^{rw}_{k\,abt}.$$

and the mean transaction response time for class C^{rw} is

$$R^{rw}_{tx} = \sum_{k=0}^{w} N^{rw}_{G_k}R^{rw}_{G_k}.$$

Concerning lock holding time equations in Section 3.3.3, the corresponding expressions for transactions of class C^{rw} are

$$T^{rw}_{H,ki} = \sum_{j=i}^{w} \hat{R}^{rw}_{ki} + \sum_{j=i+1}^{w} \tilde{R}^{rw}_{ki} + R^{rw}_{k\,com},$$

$$T^{rw}_{H,i} = \sum_{k=0}^{N_w} N^{rw}_{G_k}T^{rw}_{H,ki},$$

and

$$T^{rw}_{H} = \frac{1}{w}\sum_{i=1}^{w} T^{rw}_{H,i}.$$

Contention probability against transactions of class C^{rw} can be expressed using the average transaction arrival rates for the different classes, that is

$$P^{rw}_{cont} = \frac{\lambda^{rw} w T^{rw}_H}{D},$$

thus the average contention probability becomes

$$P_{cont} = \sum_{(r,w)\in X} P^{rw}_{cont}.$$

Version check failure probability for a transaction of class C^{rw} is therefore

$$P^{I,rw}_{A,ki} = 1 - \exp(-\frac{\lambda w_{avg}}{D}\vec{R}^{rw}_{ki}),$$

where

$$\vec{R}^{rw}_{ki} = \sum_{j=0}^{i-1}(R^{rw}_{k\hat{i}} + P^{rw}_{W,ki}R^{rw}_{k\,\widetilde{i+1}})$$

and

$$w_{avg} = \sum_{(r,w)\in X} \frac{\lambda^{rw} w}{|X|}.$$

The average lock waiting time becomes a weighted average across the waiting times caused by transactions of different classes. Hence

$$R_{k\tilde{i}} = \frac{1}{P_{cont}} \sum_{(r,w)\in X} P^{rw}_{cont} R^{rw}_{k\tilde{i}},$$

where $R^{rw}_{k\tilde{i}}$ is the residual execution time of transactions specialized for each single class.

Finally, to evaluate the average number of accessed versions for read operations, since committed versions of a data item are generated with an average rate

$$\sigma = \sum_{(r,w)\in X} \frac{\lambda^{rw} w}{D},$$

we have for read operations by a transaction of class C^{rw} the following expression

$$N^{V,rw}_{read,ki} = \Delta T^{rw}_{s,ki}\, \sigma,$$

where

$$T^{rw}_{s,ki} = \vec{R}^{rw}_{ki} + R^{rw}_{k\hat{i}}/2.$$

Expressions for the hardware resource model in Section 3.3.8 still hold when considering per class parameters.

3.4.2 Non-uniform Data Access

We now consider non-uniform data access probability. For each data item $x \in D$ we denote as $P_D(x)$ the corresponding data access probability. For simplicity, we consider in this section fixed length transactions, even though, in a similar manner to what was done in the previous section, it is possible to consider several transaction classes characterized by different lengths.

Differently from the uniform access case, the contention probability depends on the accessed data item. Data item x is locked for an approximated average time fraction $\lambda P_D(x) N_w T_H$, which we denote with $P_{cont}(x)$. Thus contention probability is

$$P_{cont} = \sum_{x\in D} P_D(x) P_{cont}(x) = \sum_{x\in D} P^2_D(x) \lambda N_w T_H.$$

To evaluate version check failure probability we note that committed versions of data item x are born with an average rate $\sigma(x) = \lambda P_D(x) N_w / D$, thus

$$P^I_{A,ki} = \sum_{x\in D} P_D(x)(1 - \exp(-\frac{\sigma(x)}{D}\vec{R}_{ki})),$$

where $\vec{R}_{ki}$ is the same as in Section 3.3.6. Also, the average number of accessed data item versions depends on the data access distribution. Hence, similarly to what done in Section 3.3.7 we have

$$N^V_{read,ki}(x) = \Delta T_{s,ki}\, \sigma(x).$$

Therefore, the average number of accessed versions for a read operation in state $\hat{i}$ of subgraph G_k is

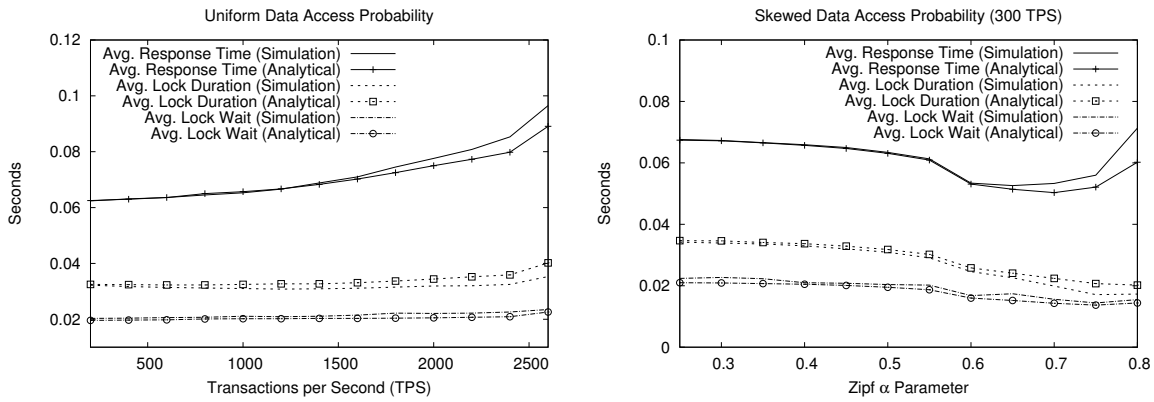
$$N^V_{read,ki} = \sum_{x\in D} P_D(x) N^V_{read,ki}(x).$$

4 Model Validation

In this section we present a simulation study aimed at evaluating the accuracy of the proposed analysis. To this end, we have developed a discrete event simulator, which explicitly models the behavior of all the components characterizing the database. These include the buffer pool, for which we have implemented in the simulator a classical Least-Recently-Used replacement policy.

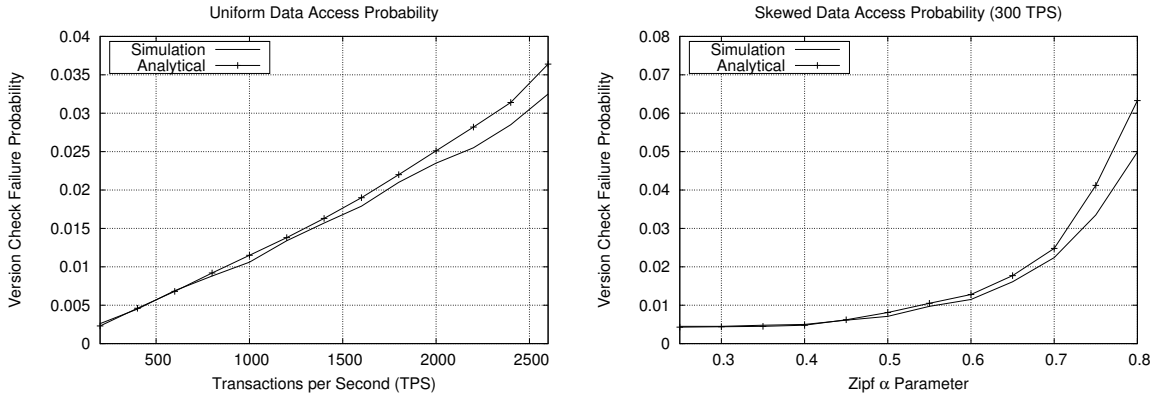
For space constraints, we report only the validation results obtained while considering parameter settings analogous to those typically used in previous concurrency control evaluation studies, e.g. [24, 25] (even though the accuracy of our analytical model was verified for a much wider range of parameter values). Specifically, we consider a database consisting of 10000 data items, and a buffer pool having size equal to the 20% of the data set of the database. Concerning the number of instructions required for the different phases of the execution of a transaction (e.g. the begin phase and the data item read phase), we have used in both the simulator and the analytical model values compliant with those used in the studies presented in [24, 25]. However, our experiments are carried out considering more modern hardware. Specifically, for both simulation and analytical model, the database system is assumed to be hosted by a 8-CPU machine with processor speed equal to 1GHz.

In a first set of observations, we aimed at verifying the accuracy of the basic analytical model, namely the one relying on the hypotheses of single transaction class and uniform data access. For this setting, we report in Figure 3(a) the transaction execution latency, the lock waiting time and the lock holding time (lock duration) vs the transaction arrival rate. In compliance with the indications in, e.g. [25], transactions of the unique class perform an expected amount of 20 data item accesses, with 20% of them being write operations. By the plotted results, we can see that the model provides a very good accuracy when comparing its latency prediction and the simulation outputs. Slight discrepancies between analytical and simulative data can be observed for transaction workload close to the system saturation point (i.e. on the order of 2500 transactions per second). To further observe the behavior of the model, we plot



(a) Uniform Data Access. (b) Skewed Data Access.

Figure 3. Latency Results.

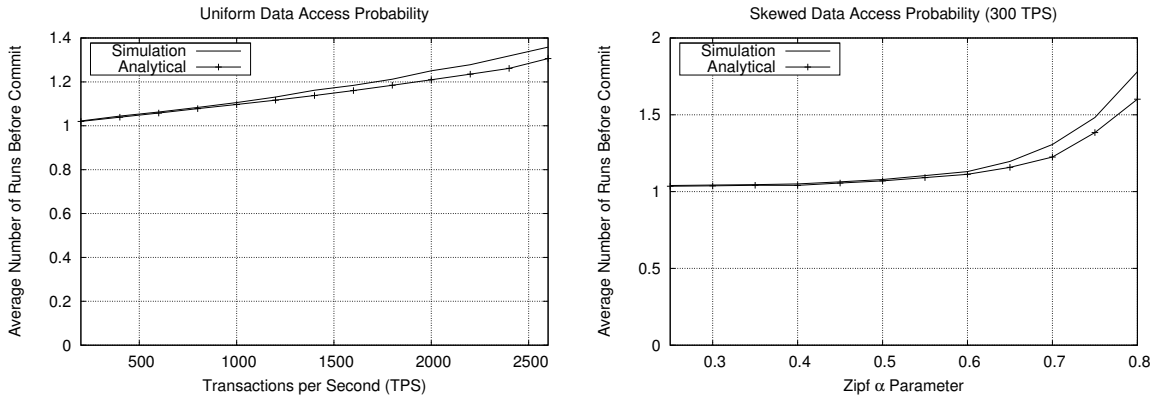


(a) Uniform Data Access. (b) Skewed Data Access.

Figure 4. Version Check Failure Probability.

in Figure 4(a) and in Figure 5(a), respectively, the probability of version check failure and the expected number of transaction (re)runs required for successful completion. The first metric is proper of snapshot-isolation based MVCC, hence allowing us to observe the model accuracy from a perspective not considered by models oriented to the evaluation of classical 2-Phase-Locking and optimistic concurrency control protocols. On the other hand, the average number of (re)runs for successful completion is a general parameter characterizing transaction management under whichever situation possibly inducing aborts. By the plotted results we have again very good compliance between analytical and simulative values, unless for workload close to the saturation point.

In a second set of experiments, we have considered non-uniform data access, so to evaluate the accuracy of the model extension provided in Section 3.4.2. We have focused on a single transaction class, with data access pattern ruled by a Zipf distribution function with parameter α. For this setting, we have fixed the transaction workload (at 300 transactions per second - TPS) and we have varied the value of α in between 0.3 and 0.8 (as in the classical range observed for data access skew in Web contexts [4, 10]). The results for latencies, version check failure probability and expected number of (re)runs are reported, respectively, in Figures 3(b), 4(b) and 5(b). Compared to the uniform data access case, the skewed data access settings show non-monotonic behavior for what concerns transaction execution latency. This is due to the mixed effects of both increased buffer hit and increased contention as the parameter α of the Zipf distribution grows. The effects show different balances while α gets increased so monotonic behavior is not guaranteed. However, also in this case the analytical model provides results well matching the simulative data. An increased discrepancy (compared to the uniform data access case) is observed near the saturation point (which is reached for α values close to 0.8). This is mainly due to that, as the skew increases, the probability for a transaction to abort because of an access to a highly popular data item correspondingly increases. The subsequent re-execution of such transactions leads, in its turn, to an overall increase of the skewness of the initially assumed data access distribution, namely $P_D(x)$. Extending the proposed model to capture this phenomenon is part of our future work.



(a) Uniform Data Access. (b) Skewed Data Access.

Figure 5. Average Number of Runs Before Commit.

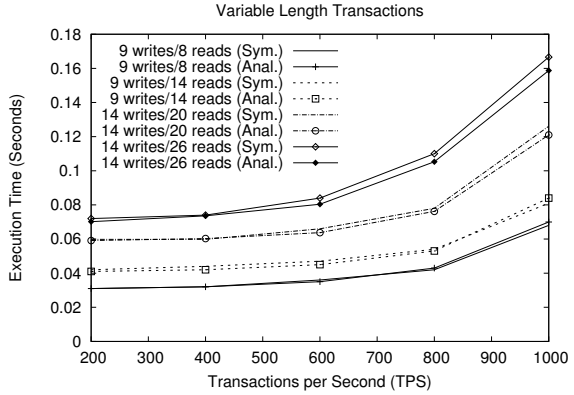


Figure 6. Latency Results for Variable Length Transactions (Uniform Data Access Case).

Finally, we have considered uniform data access but differentiated transaction classes. This has been done to evaluate the accuracy of the extension of the analytical model provided in Section 3.4.1. For this setting, we have considered 8 different transaction classes, with different length in terms of requested data items, spanning from about 20 up to 40 accessed data items, and with different percentages of read vs write operations. In Figure 6, we report the expected execution latency for 4 of the considered classes, as evaluated via both the analytical model and the simulator. Again, we observe a very good compliance between analytical and simulative data.

References

[1] K. A. Merchant, P. Yu, and M. Chen. Performance analysis of dynamic finite versioning for concurrent transaction and query processing. *ACM SIGMETRICS Performance Evaluation Review*, 20(1), June 1992.

[2] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4), December 1987.

[3] N. Al-Jumaha, H. Hassaneinb, and M. El-Sharkawia. Implementation and modeling of two-phase locking concurrency. *Information and Software Technology*, 42(4), pp.257–273, March 2000. Elsevier Science.

[4] M. Arlitt and T. Jin. A workload characterization study of the 1998 wold cup web site. *IEEE Network*, 14(3), pp.30–37, 2000.

[5] R. Balter, P. Berard, and P. Decitre. Why control of the concurrency level in distributed systems is more fundamental than deadlock management. In *Proceedings of the first ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada*, pages 183–193. ACM New York, NY, USA, 1982.

[6] B.Ciciani, D.M.Dias, and P.S.Yu. Analysis of concurrency-coherency control protocols for distributed transaction processing systems with regional locality. *IEEE Transactions on Software Engineering*, 18(10), pp.899–914, October 1992.

[7] B.Ciciani, D.M.Dias, and P.S.Yu. Dynamic and static load sharing in hybrid distributed-centralized systems. *Computer Systems Science and Engineering*, 7(1), pp.25–41, January 1992.

[8] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, volume 99, pp.1–10, May 22-25, 1995. San Jose, California, United States.

[9] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] L. Breslau, P. Cao, L. Fan, G. Phillipps, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE INFOCOM*, 1999.

[11] M. J. Carey and W. A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems*, 4(4), pp.338–378, November 1986.

[12] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2), pp.492–528, 2005.

[13] J. Gray, P. Homan, R. Obermarck, and H. Korth. A straw man analysis of probability of waiting and deadlock. *IBM Research Report RJ 3066*, 1981.

[14] I.K.Ryu and A.Thomasian. Analysis of database performance with dynamic locking. *Journal of the ACM*, 37(3), pp.491–523, July 1990.

[15] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *VLDB'07*, pages 1263–1274. VLDB Endowment, 2007.

[16] L. Kleinrock. *Queuing Systems (Vol1 and Vol2)*. Wiley-Interscience, 1975.

[17] Oracle Inc. www.oracle.com/technology/documentation/database10g.html.

[18] D. R. Ries and M. Stonebraker. Locking granularity revisited. *ACM Transactions on Database Systems*, 4(2), 1974.

[19] D. R. Ries and M. Stonebraker. Effects of locking granularity in a database management system. *ACM Transactions on Database Systems*, 2(3), September 1977.

[20] The PostgreSQL Global Development Group. Postgresql 8.2.6 Documentation.

[21] A. Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys*, 30(1), March 1998.

[22] A. Thomasian and I. Ryu. Performance analysis of two-phase locking. *IEEE Transactions on Software Engineering*, 17(5), pp.386–402, May 1991.

[23] P. Yu and M. Chen. Performance analysis of dynamic finite versioning schemes: storage cost vs. obsolescence. *IEEE Transactions on Knowledge and Data Engineering*, 8(6), December 1996.

[24] P. Yu, D. Dias, J. Robinson, B. Iyer, and D. Cornell. On coupling multi-systems through data sharing. *Proceedings of the IEEE*, 75(5), pp.573–587, May 1987.

[25] P. S. Yu, D. M. Dias, and S. S. Lavenberg. On the analytical modeling of database concurrency control. *Journal of the ACM*, 40(4), pp.831–872, September 1993.