Design and Evaluation of an Edge Concurrency Control Protocol for Distributed Graph Databases

Paul Ezhilchelvan[1], Isi Mitrani[1], Jack Waudby[1], and Jim Webber[2]

[1] School of Computing, Newcastle University, NE4 5TG, UK
{paul.ezhilchelvan,isi.mitrani,j.waudby2}@ncl.ac.uk
[2] Neo4j UK, Union House, 182-194 Union Street, London, SE1 0LH
jim.webber@neo4j.com

Abstract. A new concurrency control protocol for distributed graph databases is described. It avoids the introduction of certain types of inconsistencies by aborting vulnerable transactions. An approximate model that allows the computation of performance measures, including the fraction of aborted transactions, is developed. The accuracy of the approximations is assessed by comparing them with simulations, for a variety of parameter settings.

Keywords: Graph databases · Reciprocal consistency · Edge-order consistency · Arbitration · Stochastic modelling · Simulation

1 Introduction

Existing large-scale distributed data stores such as Google Docs, Dynamo [4] and Cassandra [3] implement an 'eventually consistent' update policy (see [14]). That is, update requests are processed as soon as they arrive. In some cases this is a reasonable choice. For a non-partitioned system there are several solutions to dealing with what is effectively lag between replicas. However, when a database is partitioned among several hosts, the eventual consistency approach raises serious problems, especially when there are explicit or (application) implied relationships between the data stored in different partitions.

For example, a patient might observe an appointment has been booked in their timeline on partition A, while the corresponding clinician in partition B hasn't yet blocked off that slot. Eventual consistency makes it possible for another patient to book into that slot either overwriting or double-booking the clinician. While each partition on its own will be eventually consistent, the system as a whole has violated a constraint.

This is similar in a sense to problems that can occur in traditional databases with Snapshot Isolation (SI), but unlike SI there are no mechanisms in eventually consistent databases to detect distributed constraint violations. For distributed graph databases this is a critical problem because explicit relationships (edges) routinely span across partitions, and unless both partitions agree reciprocally on the existence, direction, and content of the edge then the database has become corrupted.

Another example, by Bailis and Ghodsi [2] refers to an ATM service where eventual consistency can allow two users to simultaneously withdraw more money than their (joint) bank account holds; such an anomaly, on being detected, is reconciled by invoking exception handlers. Given that an ATM service is expected to be available 24/7 and that account holders are permitted to access only their own accounts, the eventually consistent approach is appropriate.

A vast majority of common graph database applications, however, allow data modified by one (user) transaction to be read by an arbitrary number of other (users') transactions (see Robinson et al, [12]). In such cases, data corrupted by one transaction and read by subsequent transactions, can lead to further corruption from which it is impossible to recover. This process, which was studied at some detail by Ezhilchelvan et al, [5], can in time cause the entire database to become unusable. That is a situation that is certainly worth avoiding.

In this paper we propose a new update protocol where conflicting updates are detected and handled. Corruption is thus prevented, but the price paid for this improvement is that some transactions are aborted. In order to evaluate just how heavy is that price, we also construct and analyse an approximate model that allows us to compute the average number of transactions aborted per unit time and other performance measures.

To simplify the protocol presentation and analysis, we assume that the hosts are reliable and data items in a database are not replicated. Provisions for crash-tolerance can be incorporated as an orthogonal aspect by using well-known techniques (e.g., single server abstraction) and supportive technologies (e.g., Raft [8], Paxos [7]).

The problem context and the proposed protocol are described in sections 2 and 3. The approximate model and its analysis are presented in section 4. Some numerical and simulation results are reported in section 5, while section 6 outlines the conclusions.

2 Problem description

A graph database consists of *nodes* representing entities, and *edges* representing relations between them (see [12]). For example, node X may represent an entity of type Author and Y an entity of type Book. X and Y will have an edge between them if they have a relation, e.g. X is an author of Y.

The popularity of the graph database technology owes much to this simple structure from which sophisticated models can be easily built and be efficiently used for query or transaction processing. Examples of operations performed on a graph database are: finding shortest paths between two locations in a transport network, performing product recommendations, looking for cancerous patterns in biological data, etc.

When nodes are connected by an edge, the database stores some reciprocal information at the origin and destination records of that edge. For example, if there is an edge from node X to node Y, then node X would have an *outgoing* record *wrote* and node Y would have an *incoming* record *wrote*, which can be interpreted as *written by*. Maintaining this reciprocal information enables an edge to be traversed in either direction.

An edge e is said to be *reciprocally consistent*, if its origin and destination records, denoted as e_1 and e_2, at the nodes that it connects, have mutually consistent, reciprocal entries.

In a distributed graph database, graph data is partitioned and each partition is hosted by a server in a cluster. Partitioning a graph is non-trivial and even the most optimal partitioning algorithms (e.g., [10], [11]) seek only to minimise, and cannot eliminate, the presence of distributed edges. The outgoing and incoming records of a distributed edge are on different hosts[1]. It has been estimated in [13], that a fraction varying between 25% and 75% of all edges would be distributed. Maintaining reciprocal consistency across a distributed edge is challenging because its e_1 and e_2 records cannot be updated simultaneously. The time interval that elapses between those updates permits interference among concurrent transactions.

Suppose, for example, that nodes F and S, referring to a flight and a seat in an airline database, are stored on hosts $H1$ and $H2$ respectively, with the edge between them indicating availability. Two transactions, U and V, write 'S is available in F' and 'S is booked in F', respectively. Each update operation is carried out first on one of the hosts and then, after a small but non-zero 'network delay', on the other host. These two phases of the update are referred to as 'part 1' and 'part 2', respectively. The delay interval between them, D, is a random variable which may, in principle, be unbounded.

Such an implementation, if left uncontrolled, makes possible the introduction of faults in the edge records. This is illustrated in Figure 1, which shows three possible conflict scenarios between transactions U and V (time flows downwards). In case (a), transaction U performs part 1 of the update on $H1$ at time t and part 2 on H_2 at time $t + D$. At some point between t and $t + D$, transaction V performs part 1 on H_2, and part 2 on H_1 some time later. The result of this occurrence is a violation of reciprocal consistency: the H_1 entry ends up saying 'seat S is booked in F', while the H_2 entry says 'seat S is available in F'.

A similar conflict is shown in case (b), except that here part 1 of V is performed on H_2 *before* time t, and part 2 on H_1 after t. Finally, there is the possibility (c), where both transaction traverse the edge in the same direction, but U overtakes V during the network delay. The result of that conflict is that $H1$ claims 'seat S is available in F', while H_2 says 'seat S is booked in F'.

In order to prevent such conflicts, only transactions whose distributed updates are 'interference-free', should be allowed to proceed. That is, if part 1 of an update for a given edge precedes part 1 of another update for the same edge, then so should part 2, and vice versa. One could, of course, avoid such conflicts

[1] This is avoided in *edge-partitioned* graph databases, where all instances of a given edge type reside in the same partition, and nodes are replicated. Then the problem is to ensure that updates to nodes are consistent across partitions.

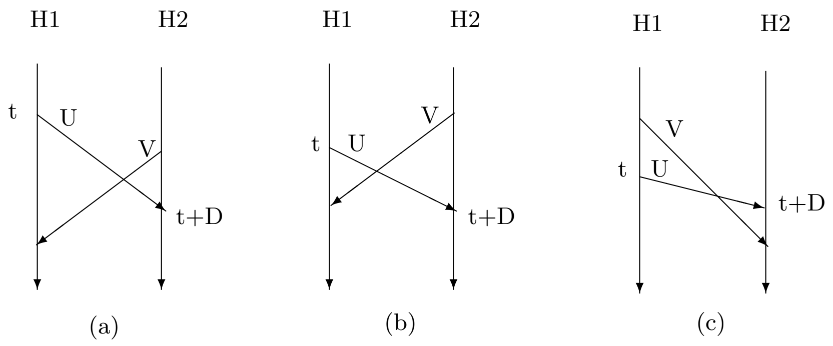

Fig. 1. Three possible conflict scenarios

by using a strong consistency mechanism, but the availability and throughput penalties are generally prohibitive.

A different type of possible conflicts arises when transactions update more than one edge during their lifetime. For example, suppose that transactions A and B both update edges e and e', and do so without interference either among themselves or with other transactions. It may happen that e is updated by A before B, while e' is updated by B before A, as illustrated in Figure 2 (here time flows from left to right and the conflict-free updates are collapsed to single instants). Such an occurrence, if allowed, would violate the property of 'edge-order consistency' between transactions.

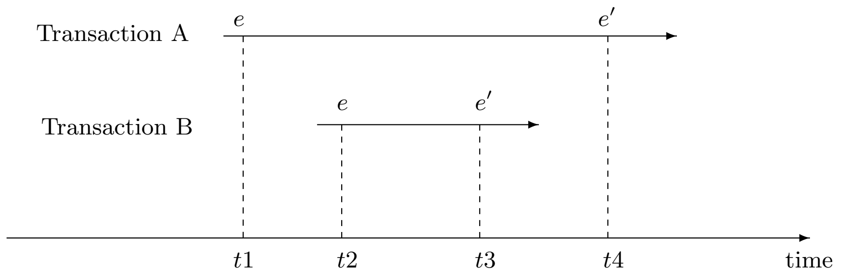

Fig. 2. Edge-order consistency violation

Inconsistencies of this type are to be avoided because they compromise an important database property known as serializability.

3 Edge Concurrency Control Protocol

Our protocol employs two distinct mechanisms, referred to as 'collision detection' and 'order arbitration', respectively. These are aimed at enforcing (i) reciprocal consistency for distributed updates and (ii) edge-order consistency between transactions. Collision detection is applied at every update and may trigger an immediate abort. Transactions that survive collision detection may, if necessary, go to order arbitration. The latter may also result in an abort.

Like many concurrency control protocols in the literature (e.g., [6], [9]), our protocol treats updates as being provisional initially. They become permanent only if, and when, the transaction that contains them is allowed to commit.

Provisional updates on a given record can occur only one at a time and each is time-stamped using the local host's clock. Thus, when a transaction attempts to update a given record, it can identify all other transactions, called *predecessors* (if any), that have earlier updated that record provisionally. Read operations receive the latest committed version of a record and ignore any provisionally updated values.

Collision detection. Remember that an update operation by a transaction for a distributed edge has a part 1, carried out at the first host visited, and part 2, performed at the second host after a network delay. The corresponding provisionally updated records are now given labels 1 and 2, respectively, and are associated with the the id of that transaction. These labels act as 'history' meta-data indicating the host where a transaction started and completed its update. The following rule is applied:

Cancellation rule: If, by the time part 2 is performed, a previous provisional update labeled 1 has been observed, but the corresponding label 2 has not been observed, then this update is cancelled. In other words, an update is cancelled if it has observed the start of a previous attempt, but not its completion.

When an update is cancelled, the transaction containing it is aborted and all its provisional records are erased.

According to the this rule, update U in Figure 1, cases (a) and (b), is cancelled because it observes a predecessor V with label 1 in $H2$, but has not observed V's label 2 in $H1$. Whether update V is cancelled or not, depends on whether U's provisional updates remain or have been erased by the time V performs part 2. In case (c), U is cancelled but V is not, because it does not observe U's label 1.

Order arbitration. The purpose of this mechanism is to detect and prevent edge-order inconsistencies between transactions. It only applies to transactions that contain more than one distributed update. Those with a single update that have not been aborted by collision detection are allowed to commit and depart.

Using the records relating to provisional updates, each multi-update transaction maintains a 'predecessor list' containing all predecessor transactions it encountered during its provisional updates. If that list is empty when the transaction successfully completes all its provisional updates, then it commits and departs. Otherwise it goes to arbitration.

The arbiter is a special service, assumed here to have been implemented in a dedicated host. A list called the *hit-list* is maintained. It contains transactions which, if allowed to commit, risk violating edge-order consistency. Transactions arriving for arbitration join a queue and are served in order of arrival. If the transaction at the head of the queue is not present in the hit-list, it commits. All transactions in its predecessor list are added to the hit list if not already there. If it is in the hit list, it aborts and all its provisional updates are erased. What has happened in this case is an overtaking: the current transaction was named as a predecessor by a transaction that committed earlier.

We can informally argue that our approach is correct by considering the edge-order inconsistency depicted in Fig 2. It can be seen that A will have B in its predecessor list while updating e', and B will observe A as a predecessor while updating e. Both A and B must approach the arbiter because they update more than one edge and have a non-empty predecessor list. If the first transaction to be processed by the arbiter is allowed to commit, the second one will be entered in the hit list and will abort. Thus only one of A and B, but not both, can commit and edge order inconsistency is always avoided.

Note that this approach to arbitration is pessimistic. It aborts a transaction as soon as it detects a risk of edge-order violation, even though the actual violation may not occur. Consequently, some transactions are aborted unnecessarily, just because they are overtaken by their successors. To eliminate unwarranted aborts, the arbiter would have to keep much more detailed information about the updates performed by all transactions, and would have to do considerably more processing.

We now proceed to the task of evaluating certain performance measures, such as the average number of transactions that are aborted per unit time, the offered load at the arbiter, and the average time a transaction remains in the system. Since the processes involved are rather complex, such an evaluation will inevitably entail approximations. That, in turn, will necessitate an assessment of the accuracy of those approximations.

4 Approximate Model

We are concerned with updates performed on distributed edges in a graph database (i.e., edges whose source and destination nodes are stored on different hosts). These edges are divided into T types, numbered 1, 2, …, T. The number of edges of type i is N_i, and the probability that an update operation is aimed at an edge of type i is p_i. All edges of a given type are equally likely to be addressed, so that the probability of accessing a particular edge of type i is p_i/N_i.

Transactions arrive into the system in a Poisson stream, at the rate of λ per second. Each transaction performs a random number, K, of updates for different distributed edges. The distribution of K is arbitrary: $P(K = k) = r_k$ ($k = 1, 2, \ldots$). The average number of updates per transaction is κ. Thus, the arrival rate of updates at a *particular* distributed edge of type i, ξ_i, is equal to

$$\xi_i = \frac{\kappa \lambda p_i}{N_i} \;\; ; \;\; i = 1, 2, \ldots, T \; . \qquad (1)$$

The first approximation is to assume that the arrival process of updates for a particular edge of type i is Poisson with rate ξ_i.

We wish to estimate the probability, u_i, that an update, U, for an edge of type i, is cancelled due to a collision with another update, V, for the same edge. That is, either V arrives in the opposite host during the network delay of U (Figure 1, case (a)), or U arrives in the opposite host during the network delay of V (case (b)), or U arrives in the same host during the network delay of V *and* its network delay completes before that of V (case (c)).

Assume that the network delays are i.i.d random variables distributed exponentially with parameter δ (mean $1/\delta$). This may or may not be an approximation.

Updates for a particular edge of type i arrive in a particular one of the two hosts involved at rate $\xi_i/2$. Moreover, a given network delay completes before another with probability 1/2. Hence, we can estimate the probabilities of cases (a), (b), and (c), $u_i^{(a)}$, $u_i^{(b)}$ and $u_i^{(c)}$, as

$$u_i^{(a)} = u_i^{(b)} = \frac{\xi_i}{\xi_i + 2\delta} \;\; ; \;\; u_i^{(c)} = \frac{1}{2}\frac{\xi_i}{\xi_i + 2\delta} \;\; ; \;\; i = 1, 2, \ldots, T \; , \tag{2}$$

where ξ_i is given by (1) and δ is the parameter of the network delay. The overall probability, u_i, that at least one of those events will happen, is

$$u_i = 1 - (1 - u_i^{(a)})(1 - u_i^{(b)})(1 - u_i^{(c)}) \approx \frac{2.5\xi_i}{\xi_i + 2\delta} \;\; ; \;\; i = 1, 2, \ldots, T \; . \tag{3}$$

The last approximation in the right-hand side holds when the rate ξ_i is small compared to δ.

The unconditional probability, u, that an arbitrary update is cancelled by the collision detection mechanism, is given by

$$u = \sum_{i=1}^{T} p_i u_i \; . \tag{4}$$

The probability, v_k, that a transaction containing k updates is aborted because one of them is involved in a collision, is equal to

$$v_k = 1 - (1 - u)^k \; , \tag{5}$$

and the unconditional probability, v, that a transaction is aborted due to a collision is given by

$$v = \sum_{k=1}^{\infty} r_k v_k \; . \tag{6}$$

Now consider the average run time, a_k, of a transaction that contains k update operations. Assume that each update takes time b, on the average. Those times include read operations and computations, as well as network delays. If the first $j - 1$ provisional updates are completed successfully but the j-th update is cancelled as a result of a collision, then the average run time would be jb. Hence, a_k is given by

$$a_k = \sum_{j=1}^{k} jb(1-u)^{j-1}u + kb(1-u)^k \ , \tag{7}$$

where u is given by (4)

With a little manipulation, this expression can be simplified to

$$a_k = b\sum_{j=1}^{k}(1-u)^{j-1} = b\frac{1-(1-u)^k}{u} \ . \tag{8}$$

The unconditional average run time of a transaction, a, is equal to

$$a = \sum_{k=1}^{\infty} r_k a_k \ . \tag{9}$$

If all provisional updates in a transaction are completed successfully, and if either there was only one update, or there were no predecessors, then the transaction commits. Otherwise it goes to the arbiter. The time that a transaction spends queueing and being served by the arbiter will be referred to as the 'arbitration time'.

Assume (this is another approximation) that each transaction joins the arbiter queue with probability α, independently of the others. That is, the arrival process is Poisson, with rate $\lambda\alpha$. The arbiter's average service time, s, is a given parameter. Thus the offered load at the arbiter is $\rho = \lambda\alpha s$.

Treating the arbiter as an $M/M/1$ queue, we estimate the average arbitration time, w, as

$$w = \frac{s}{1-\rho} \ , \tag{10}$$

provided that $\rho < 1$. If $\rho \geq 1$, then $w = \infty$. The total average time that a transaction spends in the system is

$$W = a + \alpha w \ , \tag{11}$$

where a is given by (9).

We shall now develop an iterative fixed-point approximation for α. Denote by $d_{j,k}$ the average lifetime of the j'th update within a transaction containing k updates, *excluding* any possible arbitration time. By an argument similar to the one that led to (8), we obtain

$$d_{j,k} = b\sum_{i=1}^{k+1-j}(1-u)^{i-1} = b\frac{1-(1-u)^{k+1-j}}{u} \ . \tag{12}$$

The lifetime of a randomly chosen update within a transaction containing k updates, d_k (again excluding arbitration), is given by

$$d_k = \frac{1}{k}\sum_{j=1}^{k} d_{j,k} = b\frac{(k+1)u + (1-u)^{k+1} - 1}{ku^2} \ . \tag{13}$$

Hence, the total average time spent in the system by an arbitrary update (*including* the arbitration time), d, is equal to

$$d = \sum_{k=1}^{\infty} r_k d_k + \alpha w \ , \tag{14}$$

where w is given by (10).

Now, let γ_i be the probability that an update of type i has a predecessor, i.e. the probability that such an update arrives while a preceding update for the same edge is still in the system. Assuming that the update residence times are distributed exponentially with mean d given by (14), this can be approximated as

$$\gamma_i = \frac{\xi_i d}{1 + \xi_i d} \ , \tag{15}$$

where ξ_i is given by (1).

The unconditional probability, γ, that an arbitrary update has a predecessor, is

$$\gamma = \sum_{i=1}^{T} p_i \gamma_i \ . \tag{16}$$

If a transaction contains k updates, the probability that at least one of them has a predecessor, α_k, is

$$\alpha_k = 1 - (1 - \gamma)^k \ . \tag{17}$$

Remembering that a transaction goes to the arbiter if it has more than one update *and* all updates avoid collisions *and* at least one of them has a predecessor, we write

$$\alpha = \sum_{k=2}^{\infty} r_k (1 - u)^k \alpha_k \ . \tag{18}$$

Note that the right-hand side of (18) depends on α, via (14) and (15). In other words, we have a fixed-point equation of the form

$$\alpha = f(\alpha) \ . \tag{19}$$

This can be solved by a simple iterative scheme. Start with an initial guess, α_0, say $\alpha_0 = 0$. At iteration n, compute

$$\alpha_n = f(\alpha_{n-1}) \ , \tag{20}$$

stopping when two consecutive iterations are sufficiently close to each other.

The probability α allows us to evaluate the offered load at the arbiter queue, and hence estimate the average response time of a transaction, W. Another important performance measure is the rate of aborts, R, i.e. the average number of transactions that are aborted per unit time. Note that a transaction may be aborted due to a collision, with probability v given by (6), or it may be aborted because it finds itself on the arbiter's hit list. Denoting the probability of the latter occurrence by β, we can write

$$R = \lambda[v + (1 - v)\beta] . \tag{21}$$

To find an expression for the probability β, note that a transaction, A, is aborted by the arbiter if (i) A goes to the arbiter and (ii) another successfully committing transaction, B, which arrived at the arbiter before A, had A in its list of predecessors (A would then have been added to the hit list). That is, B arrives in the system during the run time of A, tries to update one of the edges that A has updated, completes before A, goes to the arbiter and is allowed to commit.

Suppose that A contains k updates, and let t be the instant when the j-th of those updates is attempted. The average interval from t until the completion of A, given that all updates succeed, is $(k + 1 - j)b$. If the j-th update is of type i, let h_i be the average interval from t until the completion of the next transaction, B, that updates the same edge and then goes to the arbiter. That average can be estimated as

$$h_i = \frac{1}{\xi_i \alpha} + \frac{\kappa - r_1}{2(1 - r_1)} b , \tag{22}$$

where α is given by (18). The multiplier of b in the right-hand side is half of the average number of updates in a transaction, given that there are more than one.

Denote by β_{ijk} the probability that B arrives after the j-th update out of the k in A, and completes before A, and A goes to the arbiter but is aborted because B commits, given that the j-th update is of type i. We write

$$\beta_{ijk} = \alpha(1 - \beta) \frac{(k + 1 - j)b}{h_i + (k + 1 - j)b} . \tag{23}$$

Removing the conditioning on the type of update, we get the probability, β_{jk}, that A is aborted by the arbiter due to the j-th of its k updates:

$$\beta_{jk} = \sum_{i=1}^{T} \beta_{ijk} p_i . \tag{24}$$

The probability, β_k, that at least one of the k updates will cause A to be aborted, is

$$\beta_k = 1 - \prod_{j=1}^{k} (1 - \beta_{jk}) . \tag{25}$$

Finally, the unconditional probability, β, that an arbitrary transaction is aborted by the arbiter, can be expressed as

$$\beta = \sum_{k=2}^{\infty} \beta_k r_k . \tag{26}$$

The right-hand side of this equation depends on α, which has already been computed, and also on β. Thus, we have another fixed-point equation which can be be solved by an iterative procedure of the type (20).

One might wish to measure the performance of the system by a cost function of the form

$$C = c_1 W + c_2 R \, , \tag{27}$$

where c_1 and c_2 are some coefficients reflecting the relative importance given to the average response time and number of aborts. There are trade-offs that may need to be controlled. If, for example, the arbiter is overloaded, leading to large or infinite response times, a 'voluntary abort' policy may be introduced. If a transaction cannot commit upon completion (because its predecessor list is non-empty), it tosses a biased coin and, with probability σ, aborts instead of going to the arbiter. The offered load at the arbiter queue would then be reduced to $\rho = \lambda\alpha(1-\sigma)s$. The optimal value of σ would be chosen so as to minimize the cost function C.

5 Numerical and Simulation Results

The purpose of this section is to assess the accuracy of the model estimates by comparing them with simulations. In order to reduce the number of parameters to be set, we focus on the smallest and most frequently accessed class of edges, ignoring the larger classes where conflicts are very unlikely to occur. The examples we have chosen contain a single class with N distributed edges, each of which is equally likely to be the target of an update. The size and traffic parameters are typical of a large scale-free graph database (see also [5]).

In the first example, N is varied between 5000 and 25000 edges. The arrival rate is fixed at $\lambda = 1000$ transactions per second. The average network delay is assumed to be 5 milliseconds (i.e., $\delta = 200$). That is also the value of b (the average time per update). The distribution of the number of updates in a transaction is geometric, with mean $\kappa = 5$. The average arbiter service time is $s = 0.01$ and that value will be kept fixed in the following examples.

In Figure 3, the total average number, R, of transaction aborted per unit time by the collision detection and by the order arbitration parts of the protocol, is plotted against the number of edges. The estimated points are computed by the algorithm described in section 3, while each simulated point represents the result of a simulation run where one million transactions pass through the system.

Intuitively, we expect that when the number of edges increases, there will be fewer collisions and instances of overtaking, and therefore fewer aborts. Indeed, that is what is observed. The model consistently underestimates the number of aborts, but the relative errors are not large. They vary from 9% at $N = 5000$ to 5% at $N = 25000$. That underestimation is probably caused by the simplifying assumptions used in deriving the approximate estimates. On the other hand, the times taken to produce the two plots were vastly different: the model plot took a small fraction of a second to compute, while the simulation runs were several orders of magnitude slower.

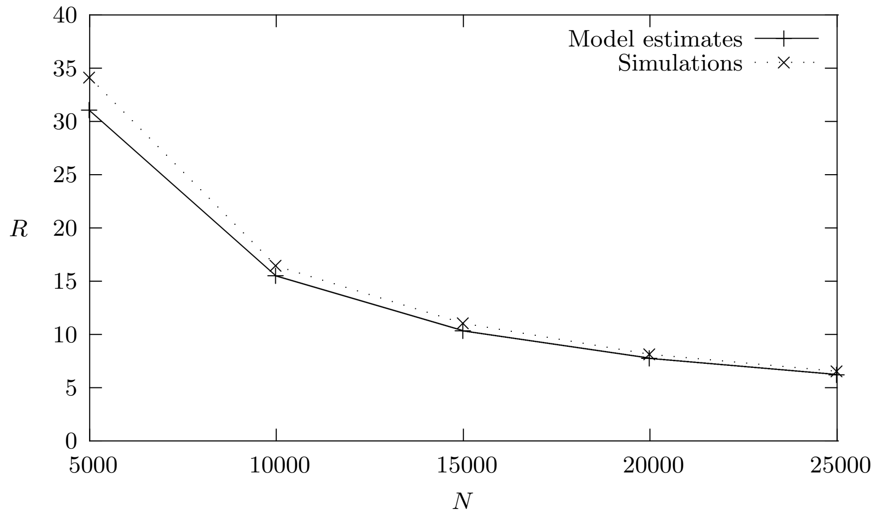

Fig. 3. Abort rate as a function of N
$\lambda = 1000$, $\kappa = 5$, $\delta = 200$, $b = 0.005$, $s = 0.01$

From now on, the number of edges will be fixed at $N = 10000$ and the effect of different parameters will be explored. In the second example, the arrival rate λ is varied between 700 and 1200 transactions per second, while the other parameters are kept as before.

In Figure 4, the average number of aborted transactions per second, R, is plotted against the arrival rate λ, using both the model approximation and simulations. Each simulated point is again the result of a run where one million transactions pass through the system. Once more, we observe that the model slightly under-estimates the values of R, but the relative errors are quite small; they are on the order of 6% or less, over the entire range.

The average response time of a transaction, W, was about 25 milliseconds; its value changed very little over this range of arrival rates.

For these parameter values, the model predicts that the arbiter queue becomes unstable when the arrival rate is about $\lambda = 1500$. The simulation agrees. The observed rate at which transactions join the arbiter queue exceeds the service rate, $\mu = 100$, for that value of λ.

For the next experiment, the average network delay is doubled to 10 milliseconds, $\delta = 100$. Intuitively, this should have the effect of increasing the rate at which transactions are aborted, and also should increase the offered load at the arbiter queue.

Figure 5 confirms our intuition. The relative errors of the model estimates are still quite low, on the order of 9% or less. The arrival rate is now varied

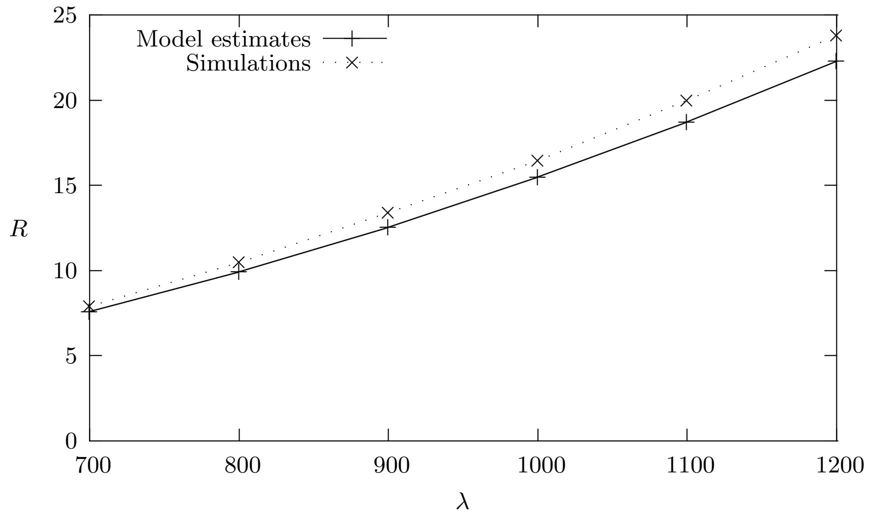

Fig. 4. Abort rate as a function of λ
$\kappa = 5$, $\delta = 200$, $b = 0.005$, $s = 0.01$

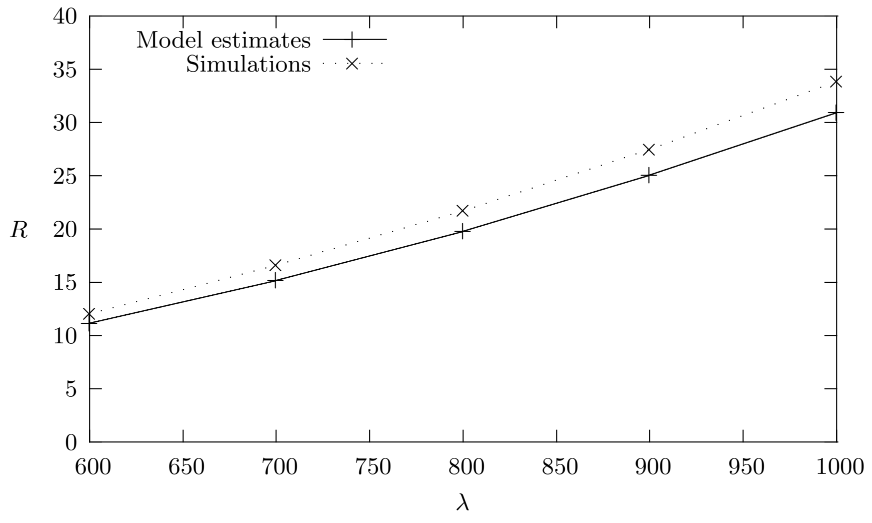

Fig. 5. Larger network delays
$\kappa = 5$, $\delta = 100$, $b = 0.01$, $s = 0.01$

between $\lambda = 600$ and $\lambda = 1000$. Both the model and the simulation agree that the arbiter queue becomes unstable when $\lambda = 1100$.

In the fourth experiment, the network delay is back to 5 milliseconds, but the number of updates in a transaction, K, has a different distribution and mean. The assumption now is that K is uniformly distributed on the range [1,19], with a mean of 10. The results are illustrated in Figure 6

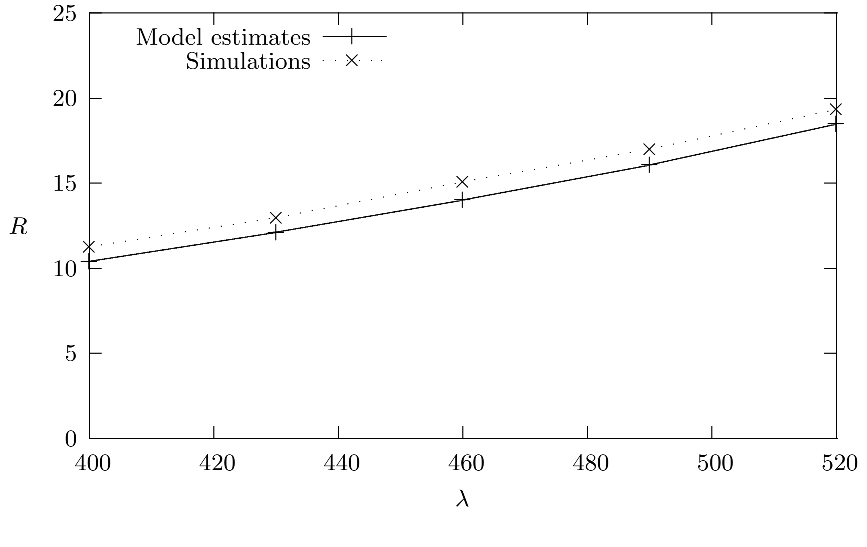

Fig. 6. Different distribution of updates
$\kappa = 10$, $\delta = 200$, $b = 0.005$, $s = 0.01$

The larger number of updates per transaction leads to both higher likelihood of collisions and more visits to the arbiter. The saturation point for the arbiter queue is now a little below $\lambda = 550$. As Figure 6 illustrates, the model approximation is still accurate, with relative errors on the order of 8% or less.

It is perhaps worth noting that in the last three examples, the rate of aborts increases roughly linearly with λ. For all arrival rates in example 2, between 1% and 2% of the incoming transactions are aborted. In example 3 that fraction is between 2% and 3%, while in example 4 it is between 3% and 4%.

6 Conclusion

We have addressed the information corruption problem caused by interferences among transactions which update distributed edges. The proposed concurrency control protocol has two distinct mechanisms: collision detection and arbitration between transactions. That protocol has an impact on system performance, in terms of aborted transactions and load on the arbiter. To evaluate this impact, an approximate model was developed and solved. It provides estimates for the average number of transactions that are aborted per unit time, the probability that a transaction will need to go to arbitration, and the average response time of a transaction. The accuracy of the solution was examined by comparisons with simulations and was found to be very high under a variety of parameter settings.

Similar to the edge-order inconsistency examined here, there may also be *node-order* inconsistency, occuring when transactions interfere while updating the same set of nodes. Eliminating node-order inconsistencies will be addressed in future work. It is well known in the database literature that there is a hierarchy of approaches which achieve various degrees of concurrency control (see [1]). Selecting an approach for a given application typically involves a trade-off between consistency requirements and performance. The most stringent common form of concurrency control is *serializability*, which maintains an abstraction of transactions being executed in some serial order. That requirement incurs the highest performance overhead.

References

1. Adya, A.: Weak consistency: a generalized theory and optimistic implementations for distributed transactions. PhD thesis, Massachusetts Institute of Technology (1999)
2. Bailis, P. and Ghodsi, A.: Eventual Consistency Today: Limitations, Extensions, and Beyond. ACMQueue **11**(3), 20–32 (2013)
3. Apache Cassandra, http://cassandra.apache.org/. Last accessed 11 Dec 2019
4. DeCandia, D. Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. SIGOPS Oper. Syst. Rev. **41**(6), 205–220 (2007)
5. Ezhilchelvan P., Mitrani I., and Webber J. (2018). On the degradation of distributed graph databases with eventual consistency. In: Bakhshi R., Ballarini P., Barbot B., Castel-Taleb H., Remke A. (eds.) EUROPEAN PERFORMANCE ENGINEERING WORKSHOP 2018, LNCS, vol. 11178, pp. 1–13. Springer, Cham (2018)
6. Kung, H.T. and Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. on Database Systems (TODS) **6**(2), 213–226 (1981)
7. Lamport, L.: The part-time parliament. ACM Trans. on Computer Systems (TOCS) **16**(2) 133–169 (1998)
8. Ongaro, D., and Ousterhout, J.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference, pp. 305–319. USENIX Association, Philadelphia, PA (2014)
9. Escriva, R., Wong, B. and Sirer, E.G.: Warp: Lightweight Multi-Key Transactions for Key-Value Stores. CoRR:abs/1509.07815, (2015)
10. Huang, J. and Abadi, D.J.: Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. VLDB Endowment **9**(7), 40–551 (2016)
11. Firth, H. and Missier, P.: TAPER: query-aware, partition-enhancement for large, heterogeneous graphs. Distributed and Parallel Databases **35**(2), 85–115 (2017)
12. Robinson, I., Webber, J. and Eifrem, E.: Graph Databases, New Opportunities for Connected Data. O'Reilly Media, Inc (2015)

13. Stanton, I. and Kliot, G.: Streaming Graph Partitioning for Large Distributed Graphs. In: 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 1222–1230, ACM, Beijing, China (2012)
14. Vogels, W.: Eventually Consistent. Comm. ACM, **52**(1), 40–44 (2009)