Queueing Models and
Service Management
Vol. 3, No. 2, page 235-253, 2020

QMSM

Modeling the Gradual Degradation of Eventually-Consistent Distributed Graph Databases

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

[1]School of Computing, Newcastle University
NE4 5TG, United Kingdom
[2]Neo4j UK, Union House, 182-194 Union Street, London
SE1 0LH, United Kingdom
(Received March 2020 ; accepted June 2020)

Abstract: Under the 'eventual consistency' approach to updates in a distributed graph database, it is possible that edge information may be corrupted. Errors may then be propagated to other parts of the database by subsequent queries. The process by which this occurs is modeled, with the aim of estimating the time that it takes for a clean database to become degraded to the point of being unusable. A fluid approximation is developed and two solution methods are proposed. The accuracy of those solutions is examined thoroughly, for databases with different sizes, structures and parameter settings, using simulations as a basis of comparison.

Keywords: Analytical evaluation, distributed edges, graph databases, half-corrupted edges, networked servers, simulations.

1. Introduction

Managing distributed data typically means adopting either strongly or eventually consistent update policies. In the former approach, replicas of a data item are updated in an identical order at all servers [8]. When update requests are launched concurrently, users will see an identical update sequence, irrespective of the actual physical server they use for accessing the data. However, this single-server abstraction imposes a significant performance penalty, since update requests need to be ordered (and replicated) prior to being acted upon. Further, according to the CAP theorem [2], [7], if the network partitions the servers, request ordering and access to data may be interrupted and the availability of services can be reduced.

In view of these disadvantages, large distributed data stores such as Google Docs and Dynamo [5], opt instead for an eventually consistent update policy ([15], [1]), whereby update requests are processed as soon as they arrive. This enhances system performance and availability but leaves a time window in which values of a replicated data item at different servers can be inconsistent. These inconsistencies lead to incorrect data operations (see Chapter 5 in [10]).

In this paper, we focus on the effects of adopting the eventually consistent policy in systems where operations that result in state anomalies are not immediately or readily ob-

*Corresponding author
Email : isi.mitrani@newcastle.ac.uk

served, and can lead to large scale propagation of erroneous states. Reconciliation at a later time can become impossible.

The systems of interest here are Graph Databases [11], which are a rapidly growing database technology at present. In particular, we examine those databases, such as JanusGraph (https://janusgraph.org/), that are built by porting Graph Data on eventually consistent backend servers running Cassandra or HBase.

A graph database consists of *nodes* and *edges*, representing entities and relations between them, respectively. For example, node A may represent a person of type Author and B an item of type Book. A and B will have an edge between them if they have a relation, e.g. A is an author of B. The popularity of the graph database technology owes much to this simple structure from which sophisticated models can be easily built and efficiently queried. Examples of operations performed on a graph database are: discovering influential people in social networks, ranking the most relevant pages in the web, etc.

When nodes are connected by an edge, the distributed graph database stores some reciprocal information. For example, if there is an edge between A and B, then A would have a field *wrote B* and B would have a field *written by A*; similarly, if there is an edge between a music fan F and a singer S, then they would have the reciprocal fields *following S* and *being followed by F*. Storing this reciprocal information in a distributed database is a non-trivial problem since the node information stored across different servers must remain mutually compatible. Any updates in one connected node at one server must also be reflected in the other node(s) at a different server(s).

It has been observed that, when a large database is partitioned for scalability reasons across multiple servers, a non-negligible fraction of edges end up being distributed [4], [12], [13]. In [14], it was noted that this fraction can vary between 0.25 and 0.75 during online partitioning of large graph databases.

When a query writes a distributed edge, the eventual consistency policy cannot ensure that the necessary updates are implemented in identically-ordered fashion across the servers involved. Suppose that two queries operate (nearly) simultaneously on a given distributed edge, each starting from a different server. Then the two updates can be implemented in a different order at the two servers, leading to a mutual incompatibility between two nodes (if servers are replicated for availability, it is conceivable that all replicas of each server implement the updates in opposite order). While the nodes of a distributed edge are in mutually incompatible states, there may be a stream of queries reading one node and another stream reading the other. One of the two streams is obviously reading incorrect information (from a global point of view).

The above scenario is not hypothetical. The operation manual of JanusGraph (see [https://docs.janusgraph.org/advanced-topics/eventual-consistency]), cautions the users about such conflicts occurring in real systems. They recommend two safety mechanisms that can be employed to eliminate such occurrences. The first method employs traditional locking and unlocking of database records to enforce concurrency and it is well-known that locking incurs considerable degradation of performance (this is also stated in the manual). The alternative advocated is to simply let the inconsistency arise during updates and use subsequent

read operations to detect its existence and somehow perform a reconciliation. The overhead of such 'read repairs' tends to be high since a query that intends to read only one end of a distributed edge must now read both ends by traversing the network.

Furthermore, to the best of our knowledge, no reconciliation mechanism has been proposed for users who seek to store graph data on eventually consistent backends. In fact, it is not difficult to construct examples where two or more separate conflicts are related, and repairing any one of them without being aware of all others cannot ensure correct reconciliation.

Given the performance-degrading aspects of known mechanisms for repairing conflicts, users of graph data on eventually consistent backends tend not to use any repair mechanism. Instead, they assume that the consequences of conflicting updates can be ignored because the probability of their occurrence is small. This paper aims to show that such an assumption is very dangerous.

A query that reads incorrect information about one edge, and then updates another edge, introduces incorrect information at both nodes of the second edge. Those two nodes may then be mutually compatible, yet incorrect. Such errors cannot be detected by simple compatibility tests. Moreover, they are propagated throughout the database by subsequent queries that carry out updates based on incorrect information. Eventually, the quality of information held by the database becomes so degraded that the database becomes unusable.

The contribution of this paper is to construct, analyze and solve a model of the above process of degradation. As far as we know, this has not been done before. The present work is thus the first in offering an accurate quantitative assessment of the damage that the eventually consistent update policy can inflict on a distributed graph database.

We provide two easily implementable and efficient solutions that allow us to determine the time it takes for the database to become significantly corrupted. The first of these solutions is faster, but its accuracy deteriorates when applied to very large, structureless databases. The second is slower, but has universal applicability. Both solutions are used in experiments aimed at examining quantitatively the effect that various parameters have on the degradation process. At the same time, their accuracy is evaluated by comparisons with simulations.

The model is described in section 2. Section 3 develops a fluid approximation and presents the first solution, based on fixed point iterations. That solution is applied and evaluated in section 4, where its limitations are also explored. Section 5 presents and evaluates the second solution, which combines fixed point iterations with a piecewise-linear approximation.

A shorter version of this paper was presented in [6]. The present extension includes a significant amount of additional material. In particular, the second solution method is new, as are the experimental results concerning large uniform and structured databases.

2. The Model

A graph database contains, for each node, a list of adjacency relations describing the incoming and outgoing edges associated with that node. When an edge is updated, the corresponding entries in both the origin and the destination nodes must be updated. If those two nodes are stored on the same server, then the edge is said to be 'local'. A local update is assumed to be instantaneous. An edge connecting two nodes stored on different servers is said to be 'distributed'. A 'write' operation for a distributed edge is carried out first on one of its servers and then, after a small but non-zero delay, on the other. That is, even though the edge is distributed, it should be considered as a single logical entity with respect to write operations.

This implementation of distributed writes leaves open the possibility of introducing faults in edge records. Consider an edge e, spanning two servers, S_1 and S_2. A query Q_1, containing a write operation for e, arrives in S_1 at time t and is performed in S_2 at time $t + \delta$. At some point between t and $t + \delta$, another query, Q_2, also writing e, arrives in S_2 and is performed in S_1 some time later. The result of this occurrence, which will be referred to as a 'conflict', is that the S_1 entry for e is written in the order Q_1, Q_2, while the S_2 entry is written in the order Q_2, Q_1. Both these entries cannot be considered correct; if one of them is feasible, then the other is not, and vice-versa. Such distributed edges are said to be 'half-corrupted'.

The mechanism of possible conflict is illustrated in Figure 1, where time is shown flowing downwards.

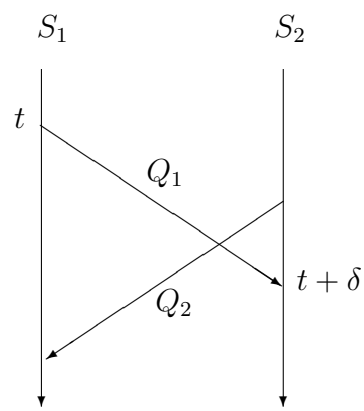

Figure 1. Conflict between Q_1 and Q_2.

A subsequent query which happens to read the correct entry of a half-corrupted edge, and completes a write operation for it without a conflict, will repair the fault and make the edge record clean again. However, if it reads the incorrect entry and writes any edge, it causes the target to become 'semantically corrupted', or simply 'corrupted'. It is assumed that the correct and incorrect reads of a half-corrupted edge are equally likely, i.e. each of them occurs with probability 0.5.

Any edge can become corrupted by being written on the basis of reading incorrect information. Corrupted edges cannot be repaired, since there is no post-facto solution to the

graph repair problem in the general case. This is because graphs, as a semi-structured data model, impose very few constraints on valid states. Thus, most corrupted states would appear as valid to any subsequent recovery scheme.

Queries that update edges arrive in a Poisson stream, at the rate of λ per second. We assume that each such query contains a random number of read operations, K, followed by one write operation. This is a conservative assumption, since more than one writes per query would increase the rate of corruption. The variable K can have an arbitrary distribution, with probabilities $P(K = k) = r_k$. Let $g(z)$ be the corresponding generating function:

$$g(z) = \sum_{k=0}^{\infty} r_k z^k \ . \tag{1}$$

A possible assumption for the distribution of K is that it is geometric with parameter r, and also satisfies $K \geq s$ for some integer s (in some applications there may be a minimum number of reads. That is, $r_k = 0$ for $k < s$; $r_s = r$, $r_{s+1} = (1-r)r$, etc. Then the generating function has the form

$$g(z) = \frac{z^s r}{1 - z(1 - r)} \ . \tag{2}$$

The K edges read, and the one written by the query are assumed to be independent of each other (but note below that they are not equally likely).

The edges in the database are divided into T types, numbered 1, 2, ..., T in reverse order of popularity. The probability that a read or a write operation accesses an edge of type i is p_i, with $p_1 > p_2 > \ldots > p_T$. The number of edges of type i is N_i, and typically $N_1 < N_2 < \ldots < N_T$. The total number of edges is N. For every type, a fraction f of the edges are distributed and the rest are local. The probability of accessing a particular edge of type i, for either reading or writing, is p_i/N_i.

At time 0, all edges are clean (free from corruption). When a certain fraction, γ (e.g., $\gamma = 0.1$), of all edges become corrupted, the database itself is said to be corrupted for practical purposes. The object of the analysis is to provide an accurate estimate of the length of time that it takes for this to happen.

At any moment in time, an edge belongs to one of the following four categories.

Category 0: Local and clean.

Category 1: Distributed and clean.

Category 2: Half-corrupted.

Category 3: Corrupted.

Only distributed edges can be in category 2, but any edge, including local ones, can be in category 3.

Denote by $n_{i,j}(t)$ the number of type i edges that are in category j at time t. The vector $\mathbf{n}_i(t) = [n_{i,0}(t), n_{i,1}(t), n_{i,2}(t), n_{i,3}(t)]$, defines the state of the type i edges at time

t ($i = 1, 2, \ldots, T$). The state of the entire database is defined by the vector $\mathbf{n}(t) = [\mathbf{n}_1(t), \mathbf{n}_2(t), \ldots, \mathbf{n}_T(t)]$.

At all times, the elements of vector $\mathbf{n}_i$ add up to N_i. Any state $\mathbf{n}(t)$ such that

$$\sum_{i=1}^{T} n_{i,3}(t) \geq \gamma N \ , \tag{3}$$

will be referred to as an 'absorbing state'. The absorbing states correspond to a corrupted database.

The value of interest is U, the average first passage time from the initial state where $\mathbf{n}_i(0) = [(1-f)N_i, fN_i, 0, 0]$ (i.e., a clean database), to an absorbing state.

The above assumptions and definitions imply that a read operation performed at time t would return a correct answer with probability $\alpha(t)$, given by

$$\alpha(t) = \sum_{i=1}^{T} \frac{p_i}{N_i}[n_{i,0}(t) + n_{i,1}(t) + \frac{1}{2}n_{i,2}(t)] \ . \tag{4}$$

The probability, $\beta(t)$, that all the read operations in a query arriving at time t return correct answers, is equal to

$$\beta(t) = \sum_{k=0}^{\infty} r_k \alpha^k(t) = g[\alpha(t)] \ . \tag{5}$$

To simplify the notation, we have written $\alpha(t)$ and $\beta(t)$ as functions of t. However, it is important to remember that they are really functions of the current system state, $\mathbf{n}(t)$.

Consider now the probability, q_i, that a query of type i arriving at time t and taking a time δ to complete a write operation, will be involved in a conflict. That is the probability that another query of type i arrives between t and $t+\delta$ and writes the same edge, but starting at its other end. This can be expressed as

$$q_i = 1 - e^{-\frac{1}{2}\lambda p_i \delta / N_i} \ \ ; \ \ i = 1, 2, \ldots, T \ . \tag{6}$$

It should be pointed out that in practice δ is dominated by network delays. The actual processing time associated with a write operation is negligible.

If the time to complete a distributed write is not constant, but is distributed exponentially with mean δ, then the expected conflict probability would be

$$q_i = \frac{\lambda p_i \delta}{2N_i + \lambda p_i \delta} \ \ ; \ \ i = 1, 2, \ldots, T \ . \tag{7}$$

When δ is small, there is very little difference between these two expressions.

An incoming query that is involved in a conflict would change the category of a distributed edge from 1 to 2, provided that all read operations of both queries return correct

results. Hence, the instantaneous transition rate, $a_{i,1,2}$, from state $[n_{i,0},n_{i,1},n_{i,2},n_{i,3}]$ to state $[n_{i,0},n_{i,1}-1,n_{i,2}+1,n_{i,3}]$, can be written as

$$a_{i,1,2} = \frac{\lambda p_i n_{i,1}}{N_i} q_i \beta^2 \ , \tag{8}$$

where β is given by (4) and (5).

Conversely, an incoming query writing a category 2 edge can change it to a category 1 edge, provided that all its read operations return correct results and it is not involved in a conflict. Hence, the instantaneous transition rate, $a_{i,2,1}$, from state $[n_{i,0},n_{i,1},n_{i,2},n_{i,3}]$ to state $[n_{i,0},n_{i,1}+1,n_{i,2}-1,n_{i,3}]$, is given by

$$a_{i,2,1} = \frac{\lambda p_i n_{i,2}}{N_i} (1-q_i)\beta \ . \tag{9}$$

The other possible transitions convert an edge of category 0, 1 or 2 into an edge of category 3. This happens when a query writes after receiving an incorrect answer to at least one of its reads. Denoting the corresponding instantaneous transition rates by $a_{i,j,3}$, for $j = 0, 1, 2$, we have

$$a_{i,j,3} = \frac{\lambda p_i n_{i,j}}{N_i} (1-\beta) \ . \tag{10}$$

Using these transition rates, one can simulate the process of corrupting the database and obtain both point estimates and confidence intervals for the average time to corruption, U. However, the systems of practical interest tend to be large, and such simulations take a very long time to run. It is therefore desirable to develop an analytical solution that is both efficient to implement and provides accurate estimates for U. That is our next task.

3. Fluid Approximation

Instead of describing the system state by integer-valued functions specifying numbers of edges of various types and categories, it is convenient to use continuous fluids of those types and categories. So now $n_{i,j}(t)$ is a real-valued function indicating the amount of fluid present at time t in a 'bucket' of type i and category j ($i = 1, 2, \ldots, T$; $j = 0, 1, 2, 3$). The total amounts of different types, and the initial states, are the same as before.

Fluids flow out of, and into buckets, at rates consistent with the transition rates described in the previous section. Thus, the bucket labeled $(i, 0)$ (local of type i and clean) has an outflow at the rate given by (10), and no inflow. This can be expressed by writing

$$n'_{i,0}(t) = -\frac{\lambda p_i [1-\beta(t)]}{N_i} n_{i,0}(t) \ , \tag{11}$$

where $\beta(t)$ is given by (4) and (5).

The bucket labeled $(i, 1)$ (distributed of type i and clean) has two outflows, at rates given by (8) and (10) respectively, and an inflow at rate given by (9). The corresponding equation is,

$$n'_{i,1}(t) = -\frac{\lambda p_i[q_i\beta^2(t) + 1 - \beta(t)]}{N_i} n_{i,1}(t) + \frac{\lambda p_i(1-q_i)\beta(t)}{N_i} n_{i,2}(t) \ . \tag{12}$$

Similarly, bucket $(i, 2)$ (half-corrupted of type i) has two outflows, at rates given by (9) and (10) respectively, and an inflow at rate given by (8). This implies

$$n'_{i,2}(t) = -\frac{\lambda p_i[1 - q_i\beta(t)]}{N_i} n_{i,2}(t) + \frac{\lambda p_i q_i \beta^2(t)}{N_i} n_{i,1}(t) \ . \tag{13}$$

Finally, bucket $(i, 3)$ (corrupted of type i) has three inflows, at rates given by (10), and no outflows. Hence,

$$n'_{i,3}(t) = \frac{\lambda p_i[1 - \beta(t)]}{N_i} [n_{i,0}(t) + n_{i,1}(t) + n_{i,2}(t)] \ . \tag{14}$$

The object is to determine the value U such that

$$\sum_{i=1}^{T} n_{i,3}(U) = \gamma N \ . \tag{15}$$

Unfortunately, the above differential equations are coupled in a complicated way. Not only do the unknown functions appear in each others equations, but they also depend on $\beta(t)$, which in turn depends on $\alpha(t)$, which depends on all the unknown functions. Moreover, that dependency is non-linear. Consequently, an exact solution for this set of equations does not appear to be feasible. We need another level of approximation.

Denote by $\bar{n}_{i,j}$ the average value of the function $n_{i,j}(t)$ over the interval $(0, U)$:

$$\bar{n}_{i,j} = \frac{1}{U} \int_0^U n_{i,j}(t) dt \ . \tag{16}$$

Replacing, in the right-hand side of (4), all functions by their average values, allows us to treat the probability $\alpha(t)$ as a constant, $\bar{\alpha}$:

$$\bar{\alpha} = \sum_{i=1}^{T} \frac{p_i}{N_i} [\bar{n}_{i,0} + \bar{n}_{i,1} + \frac{1}{2}\bar{n}_{i,2}] \ . \tag{17}$$

Then the probability $\beta(t)$ will also be a constant, $\bar{\beta}$, defined by (5). Also, where one unknown function appears in the differential equation of another, replace the former by its average value. The resulting differential equations are linear, with constant coefficients, and are easily solvable. The solution of (11), which involves only $n_{i,0}(t)$, becomes

$$n_{i,0}(t) = (1 - f) N_i e^{-a_{i,0} t} \ , \tag{18}$$

where $a_{i,0} = \lambda p_i(1-\bar{\beta})/N_i$.

In equation (12), $n_{i,2}(t)$ is replaced by $\bar{n}_{i,2}$. The solution is then

$$n_{i,1}(t) = \frac{b_{i,1}\bar{n}_{i,2}}{a_{i,1}}[1 - e^{-a_{i,1}t}] + fN_i e^{-a_{i,1}t} \ , \tag{19}$$

where $a_{i,1} = \lambda p_i(q_i\bar{\beta}^2 + 1 - \bar{\beta})/N_i$ and $b_{i,1} = \lambda p_i(1-q_i)\bar{\beta}/N_i$.

Similarly, in equation (13), $n_{i,1}(t)$ is replaced by $\bar{n}_{i,1}$. This yields

$$n_{i,2}(t) = \frac{b_{i,2}\bar{n}_{i,1}}{a_{i,2}}[1 - e^{-a_{i,2}t}] \ , \tag{20}$$

where $a_{i,2} = \lambda p_i[1 - q_i\bar{\beta}]/N_i$ and $b_{i,2} = \lambda p_i q_i \bar{\beta}^2/N_i$.

Replacing $n_{i,j}(t)$ by $\bar{n}_{i,j}$ in equation (14) ($j = 0, 1, 2$), makes the right-hand side constant and therefore

$$n_{i,3}(t) = t\frac{\lambda p_i(1-\bar{\beta})}{N_i}(\bar{n}_{i,0} + \bar{n}_{i,1} + \bar{n}_{i,2}) \ . \tag{21}$$

Hence, according to (15), the time to corruption U can be estimated as

$$U = \gamma N \left[\sum_{i=1}^{T} \frac{\lambda p_i(1-\bar{\beta})}{N_i}(\bar{n}_{i,0} + \bar{n}_{i,1} + \bar{n}_{i,2})\right]^{-1} \ . \tag{22}$$

Integrating (18), (19) and (20) over the interval $(0, U)$ and dividing by U, we obtain the following expressions:

$$\bar{n}_{i,0} = \frac{(1-f)N_i}{a_{i,0}U}[1 - e^{-a_{i,0}U}] \ ; \tag{23}$$

$$\bar{n}_{i,1} = \frac{b_{i,1}\bar{n}_{i,2}}{a_{i,1}} + (fN_i - \frac{b_{i,1}\bar{n}_{i,2}}{a_{i,1}})\frac{1}{a_{i,1}U}[1 - e^{-a_{i,1}U}] \ ; \tag{24}$$

$$\bar{n}_{i,2} = \frac{b_{i,2}\bar{n}_{i,1}}{a_{i,2}}[1 - \frac{1}{a_{i,2}U}(1 - e^{-a_{i,2}U})] \ . \tag{25}$$

This is a set of non-linear simultaneous equations for the averages $\bar{n}_{i,0}$, $\bar{n}_{i,1}$ and $\bar{n}_{i,2}$. They can be solved by consecutive iterations.

Start with some initial estimates for $\bar{n}_{i,j}$; call them $\bar{n}^{(0)}_{i,j}$. Using (17), get an initial estimate for $\bar{\alpha}$ and hence for $\bar{\beta}$; call those $\bar{\alpha}^{(0)}$ and $\bar{\beta}^{(0)}$. Then (22) provides an initial estimate for U, called $U^{(0)}$.

Substituting the initial estimates into the right-hand sides of (23), (24) and (25), yields new values for the averages $\bar{n}_{i,j}$; call them $\bar{n}^{(1)}_{i,j}$. They in turn provide new values, $\bar{\alpha}^{(1)}$ and $\bar{\beta}^{(1)}$, and a new estimate, $U^{(1)}$.

In step s of this procedure, the values $\bar{n}^{(s-1)}_{i,j}$, $\bar{\beta}^{(s-1)}$ and $U^{(s-1)}$ are used to compute $\bar{\alpha}^{(s)}$, $\bar{\beta}^{(s)}$, $\bar{n}^{(s)}_{i,j}$ and $U^{(m)}$. The process terminates when the results of two consecutive iterations are sufficiently close to each other.

The above solution can be described more concisely by treating the right-hand sides of equations (22) – (25), together with (17) and (5), as a mapping, $\mathbf{f}$, from one triple $(\bar{\mathbf{n}}, \bar{\beta}, U)$ to another. Then those equations can be re-written in the form of a single fixed-point equation:

$$(\bar{\mathbf{n}}, \bar{\beta}, U) = \mathbf{f}(\bar{\mathbf{n}}, \bar{\beta}, U) \ . \tag{26}$$

A fixed point with respect to $\bar{\mathbf{n}}$ exists by Brouwer's theorem [3], because the numbers of edges of all types and categories are bounded and the mapping $\mathbf{f}$ is continuous. Moreover, since $\bar{\beta}$ and U are determined by $\bar{\mathbf{n}}$, a fixed point exists for them too.

The iterative solution is of the form

$$(\bar{\mathbf{n}}^{(s)}, \bar{\beta}^{(s)}, U^{(s)}) = \mathbf{f}(\bar{\mathbf{n}}^{(s-1)}, \bar{\beta}^{(s-1)}, U^{(s-1)}) \ .$$

This solution will be referred to as the 'Fixed-Point approximation', or the 'F-P approximation'.

In the following section, the Fixed-Point approximation is applied to the study of reasonably realistic sample databases. The quality of the approximation is evaluated by comparisons with simulations. We shall see that, although on the whole the approximation is very accurate, there are limits to its applicability. To overcome those limitations, a more elaborate approximation will be developed in section 5.

4. Numerical and Simulation Results

Consider an example database containing five types of edges. Their numbers are: $N_1 = 10^4$, $N_2 = 10^5$, $N_3 = 10^6$, $N_4 = 10^7$ and $N_5 = 10^8$. The corresponding probabilities of access are $p_1 = 0.5$, $p_2 = 0.26$, $p_3 = 0.13$, $p_4 = 0.07$ and $p_5 = 0.04$. The number of read operations per query is distributed geometrically, starting at 2: $r_k = (1-r)^{k-2}r$ $(k = 2, 3, \ldots)$, with $r = 0.07$. Thus, on the average, there are about 15 reads per query. For a given probability, α, that a read operation returns a correct result, the probability that all reads are correct is given by

$$\beta = \frac{\alpha^2 r}{1 - \alpha(1-r)} \ . \tag{27}$$

The time to complete a distributed write operation is assumed constant, equal to 0.005 seconds.

In all types, a fraction 0.3 of the edges are distributed and the rest are local (for an argument in support of this fraction, see [Huang and Abadi 2016]). The database starts clean at time 0 and is considered to be corrupted when a fraction $\gamma = 0.1$ of all edges are corrupted.

In Figure 2, the average period until corruption is plotted against the arrival rate of queries, λ. The latter is varied in the range (100,5000) queries per second. The time U is measured in hours.

We observe that U decreases with λ. This was of course to be expected, since a higher arrival rate leads both to higher probability of conflicts, and faster spread of incorrect information. In this database, type 1 forms a relatively small nucleus of edges that are quite likely to be accessed; once they become involved in conflicts, corruption spreads rapidly.

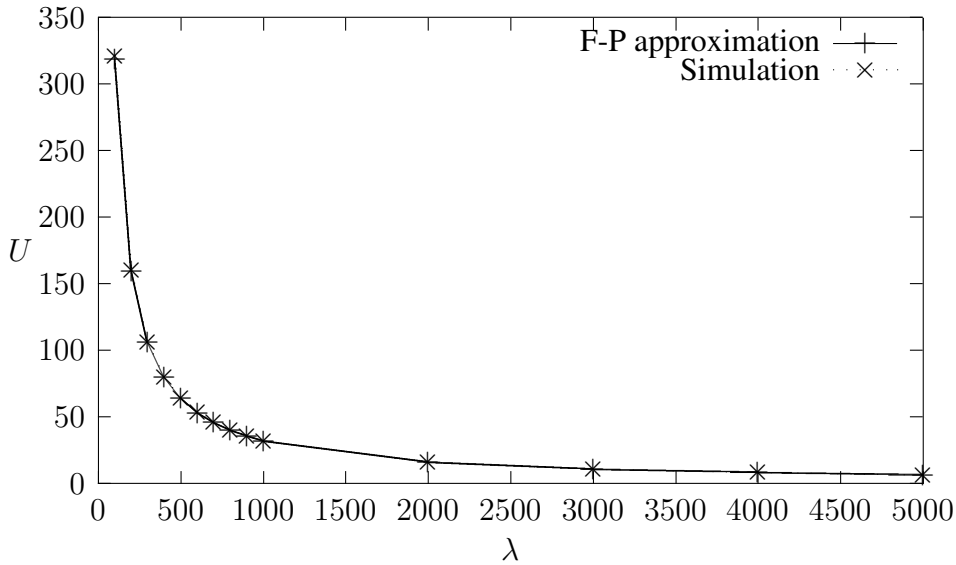

Figure 2. Corruption time in hours vs. arrival rate/sec.

The figure also aims to compare the Fixed-Point approximation results with those obtained by simulation. That is, the transition steps governed by the rates (8), (9) and (10) were simulated until an absorption state was reached.

The two plots are practically indistinguishable; the relative differences that exist are smaller than 1%. On the other hand, the Fixed-Point approximation plot took a fraction of a second to compute (each point required fewer than 10 iterations), whereas the simulated one took more than half an hour.

The next experiment examines the effect of the average number of read operations per query, $E(K)$, on the time to corruption. The arrival rate is fixed at $\lambda = 500$ queries per second. The other parameters are as in Figure 2. In this example, the random variable K has the standard geometric distribution with parameter r: $r_k = (1-r)^{k-1}r$.

In Figure 3, r decreases from 0.99 to 0.02, which means that $E(K) = 1/r$ increases from 1.01 to 50. The time to corruption, U, is again measured in hours.

As expected, the more edges are read by queries, the higher is the probability of reading a corrupted edge, and hence the shorter is the corruption time. Less obvious, however, is the observation that the resulting decrease in U is highly non-linear. Indeed, increasing $E(K)$ beyond 10 almost ceases to make a difference. We see roughly the same U, whether there are 10 or 50 reads per query.

The accuracy of the Fixed-Point approximation is again very good over the entire range of $E(K)$.

It may also be of interest to examine the effect of the fraction of distributed edges, f, on the interval U. In Figure 4, that fraction is varied between $f = 0.1$ and $f = 1$. The arrival rate is fixed at $\lambda = 100$ (in order to prolong the time to corruption), and the number of reads per query is distributed geometrically starting with 2, with parameter $r = 0.07$ and mean just over 15.

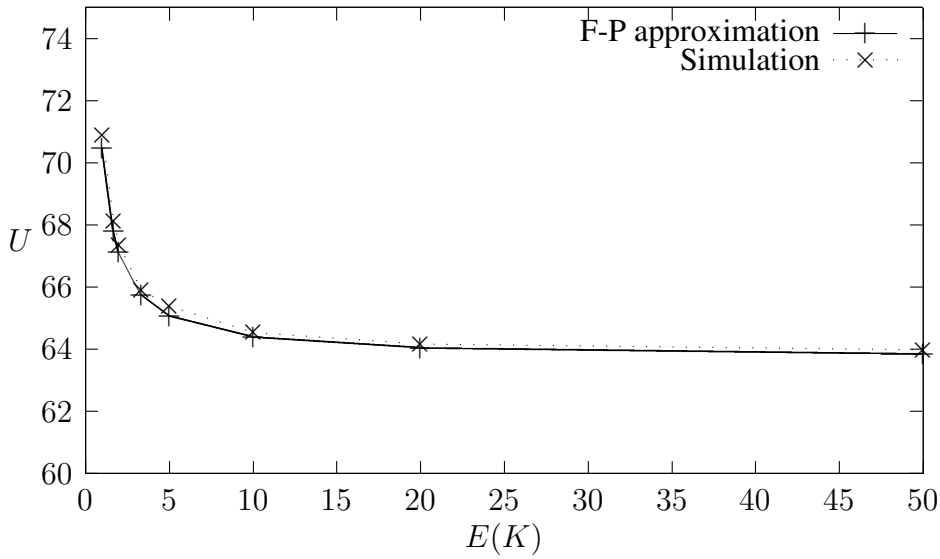

Figure 3. Corruption time in hours vs. average number of reads.

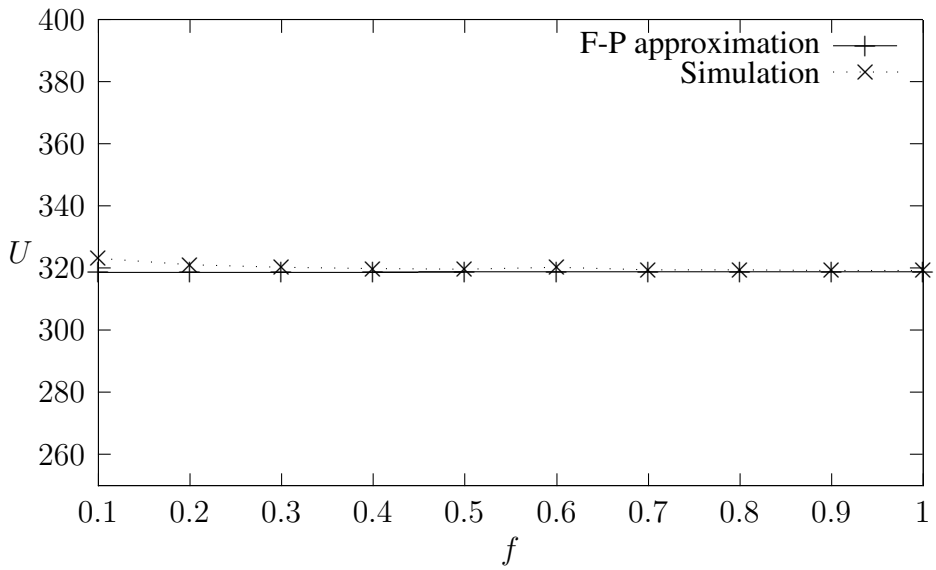

Figure 4. Varying fraction of distributed edges.

The Fixed-Point approximation plot is flat. This is not entirely surprising, since the expression for the probability α involves only the sums $\bar{n}_{i,0} + \bar{n}_{i,1}$, and not the individual averages. Moreover, both local and distributed edges are corrupted by being updated as a result of an incorrect read.

The simulation agrees with the approximation for most of the range, but begins to diverge from it when $f = 0.1$. It seems that when the fraction of distributed edges is very small, the accuracy of the Fixed-Point approximation diminishes slightly.

In the next experiment, the parameter that is varied is the fraction, γ, of edges that should become corrupted before the database is considered to be corrupted. The arrival rate is fixed at $\lambda = 500$, and all other parameters are as in Figure 2.

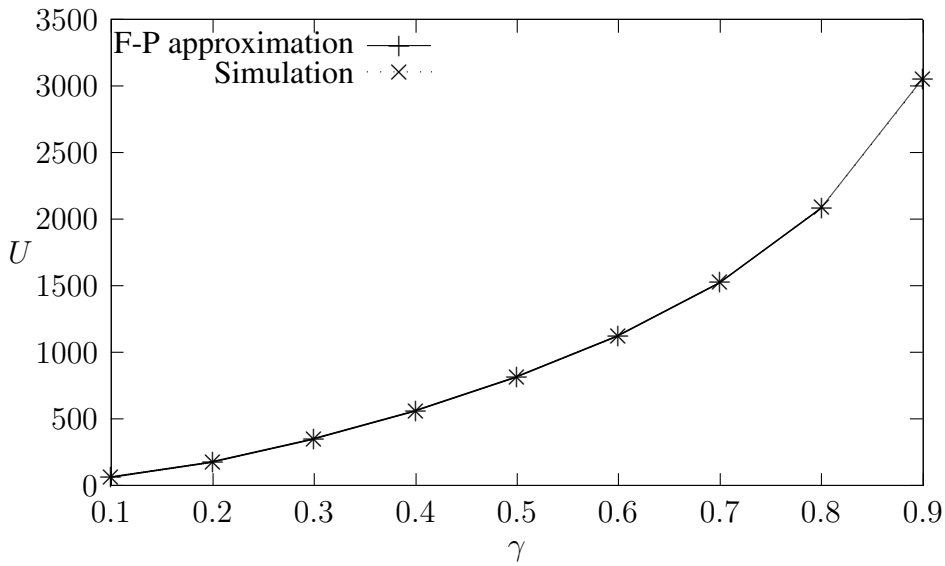

Figure 5. Varying fraction of corrupted edges.

Figure 5 shows how the time to corruption grows when the definition of corruption becomes more demanding. The plot is a convex curve, which is slightly counter-intuitive. One might guess that the more edges are corrupted, the faster even more edges would be corrupted. That would produce a concave curve. In fact the opposite is observed. The likely explanation is that the fewer the remaining clean edges, the longer it takes for a random access to hit one and corrupt it. This is particularly true when most of the edges – in this example the ones of type 5 – are accessed with a low probability.

The approximate estimates are almost indistinguishable from the simulation ones, while being several orders of magnitude faster to compute.

In order to explore the limits of applicability of the Fixed-Point approximation, we scaled up the size of the database by a factor of about 100, and also removed the smaller subsets of more popular edges. The resulting example is a database containing $N = 10^{10}$ edges, with $T = 1$ and $p_1 = 1$ (i.e., each edge is equally likely to be accessed). The fraction of distributed edges is still $f = 0.3$. This example will be referred to as the 'Large Uniform Database'.

The results are now considerably less accurate. For example, when the arrival rate is $\lambda = 1000$ and the average number of read operations per query is 2.5, the time to corruption predicted by the Fixed-Point approximation is roughly $U = 103$ days. On the other hand, a simulation run (lasting several hours) indicated a time to corruption in the region of 880 days.

The problem here appears to be the time-varying rate at which corruption spreads, determined by the time-varying probability $1-\beta(t)$. When the interval U becomes very large, an approximation that replaces $\beta(t)$ by a constant over the entire period ceases to be adequate.

The solution we propose is to break up the observation period into a number of relatively small intervals. The approximate solutions for those intervals would then be pieced together to model the evolution of the database over a long period of time.

5. Piecewise-Linear Fixed-Point Approximation

In order to simplify the presentation, it is convenient to rewrite the differential equations (11), (12) and (13), for $i = 1, 2, \ldots, T$, as a single equation in terms of the system state vector $\mathbf{n}(t)$:

$$\mathbf{n}'(t) = \mathbf{f}[\mathbf{n}(t), \beta(t)] \; . \tag{28}$$

Here $\mathbf{f}(\cdot,\cdot)$ represents all the right-hand sides of the equations, making explicit the dependance on $\beta(t)$. In fact, we know that at any moment t, $\beta(t)$ is determined by $\mathbf{n}(t)$ through (4) and (5). This can be expressed by writing $\beta(t)$ as a function of the current system state: $\beta(t) = \beta[\mathbf{n}(t)]$.

Let h be some (relatively small) time increment. Consider the sequence of instants t_0, t_1, ..., where $t_0 = 0$ and $t_m = t_{m-1} + h$ ($m = 1, 2, \ldots$). Assume that $\mathbf{n}(t)$ varies linearly on each of the intervals (t_{m-1}, t_m). Moreover, assume that on (t_{m-1}, t_m), $\beta(t)$ is a constant, denoted by $\bar{\beta}_m$. Then, applying (28) to the interval (t_{m-1}, t_m), we can write

$$\frac{\mathbf{n}(t_m) - \mathbf{n}(t_{m-1})}{h} = \mathbf{f}[\mathbf{n}(t_{m-1}), \bar{\beta}_m] \; . \tag{29}$$

Thus, if we know the value of $\bar{\beta}_m$ and the state vector at the beginning of the interval, $\mathbf{n}(t_{m-1})$, we can find the state vector at the end of the interval:

$$\mathbf{n}(t_m) = \mathbf{n}(t_{m-1}) + h\mathbf{f}[\mathbf{n}(t_{m-1}), \bar{\beta}_m] \; . \tag{30}$$

The linearity of $\mathbf{n}(t)$ on (t_{m-1}, t_m) implies that its average value on that interval is $\bar{\mathbf{n}}_m = [\mathbf{n}(t_{m-1}) + \mathbf{n}(t_m)]/2$. Now, $\beta(t)$ is not necessarily linear on (t_{m-1}, t_m), but we can approximate its average value by computing $\bar{\beta}_m = \beta(\bar{\mathbf{n}}_m)$. We thus have the elements of a new fixed-point solution of the differential equations on the interval (t_{m-1}, t_m).

Start with an initial estimate for $\bar{\beta}_m$, call it $\bar{\beta}_m^{(0)}$. A good choice for that estimate is the value determined by the initial state vector: $\bar{\beta}_m^{(0)} = \beta[\mathbf{n}(t_{m-1})]$. Use (30) to compute a first approximation for the end state, $\mathbf{n}^{(1)}(t_m)$ and hence a first approximation for the average, $\bar{\mathbf{n}}_m^{(1)}$. This provides the next estimate, $\bar{\beta}_m^{(1)} = \beta(\bar{\mathbf{n}}_m^{(1)})$. That, in turn, yields second approximations of the end state, $\mathbf{n}^{(2)}(t_m)$, average state, $\bar{\mathbf{n}}_m^{(2)}$, and average β, $\bar{\beta}_m^{(2)} = \beta(\bar{\mathbf{n}}_m^{(2)})$.

Continue these iterations until two consecutive estimates of $\bar{\beta}_m$ are sufficiently close to each other. Substituting the last value of $\bar{\beta}_m$ into (30) gives the state vector at the end of the interval, $\mathbf{n}(t_m)$, and hence the value of β at the end of the interval, $\beta(t_m) = \beta[\mathbf{n}(t_m)]$.

The Piecewise-Linear fixed-point approximation, or the 'P-L approximation', consists in carrying out the above procedure for each of the consecutive intervals $(0, h)$, $(h, 2h)$, $(2h, 3h)$, ... The end values in interval m, $\mathbf{n}(t_m)$ and $\beta(t_m)$, are the initial conditions for interval $m + 1$. For the first interval, the initial state is the clean database and the initial value of β is 1. The computation stops at the first m such that $\mathbf{n}(t_m)$ is an absorbing state; the returned value of the time to absorption is $U = t_m$.

The Piecewise-Linear fixed-point approximation was applied to the Large Uniform Database introduced at the end of the last section. A time increment of one hour was used: $h = 3600$ seconds. In Figure 6, the time to corruption, measured in *days*, is plotted against the arrival rate, which increases from $\lambda = 1000$ to $\lambda = 10000$ queries per second. The results are compared with those obtained by simulation, and also with the estimates provided by the earlier F-P approximation which computes a single fixed point over the entire period.

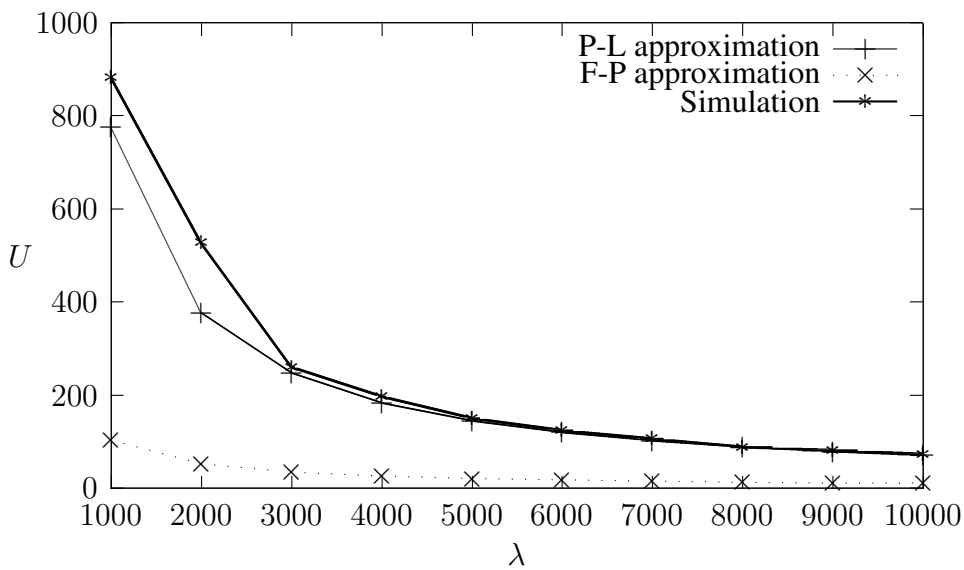

Figure 6. Large Uniform Database: time to corruption in days vs. arrival rate.

We observe a good agreement between the simulation and the Piecewise-Linear approximation. Each simulated point was the result of a single simulation run, terminated when the fraction of corrupted edges reached γ. The ten points in the graph took more than 20 hours of computer time to produce. Since we are dealing with transient, rather than steady-state behaviour, the only way to obtain samples of observations and hence confidence intervals, would be to repeat the runs multiple times. That effort was not thought to be justified.

The run time of the Piecewise-Linear approximation depends on the value of h. It tends to be larger than that of the single step solution, but not excessively so. In the above example, even if h is set to 1 minute, the difference between the P-L run time and the F-P run time is about one order of magnitude (seconds for P-L, as opposed to fractions of a second for F-P). That is a small price to pay for the much higher accuracy achieved.

Which approximation should be used if a large database has structure, rather than being

uniform? To address that question, we have increased the size of the database in figures 2 – 5 by about a factor of 100, while maintaining and extending the structure. The new database has 7 edge types, with sizes 10^4, 10^5, 10^6, 10^7, 10^8, 10^9 and 10^{10}, respectively. The corresponding access probabilities are $p = (0.5, 0.25, 0.12, 0.06, 0.04, 0.02, 0.01)$. This will be referred to as the 'Large Structured Database'.

In an experiment similar to the one in Figure 3, the value of U for the Large Structured Database is plotted against the average number of reads. The arrival rate is fixed at $\lambda = 1000$. Each point is evaluated by the F-P approximation, the P-L approximation with an increment of $h = 1$ hour, and by simulation. The results are shown in Figure 7.

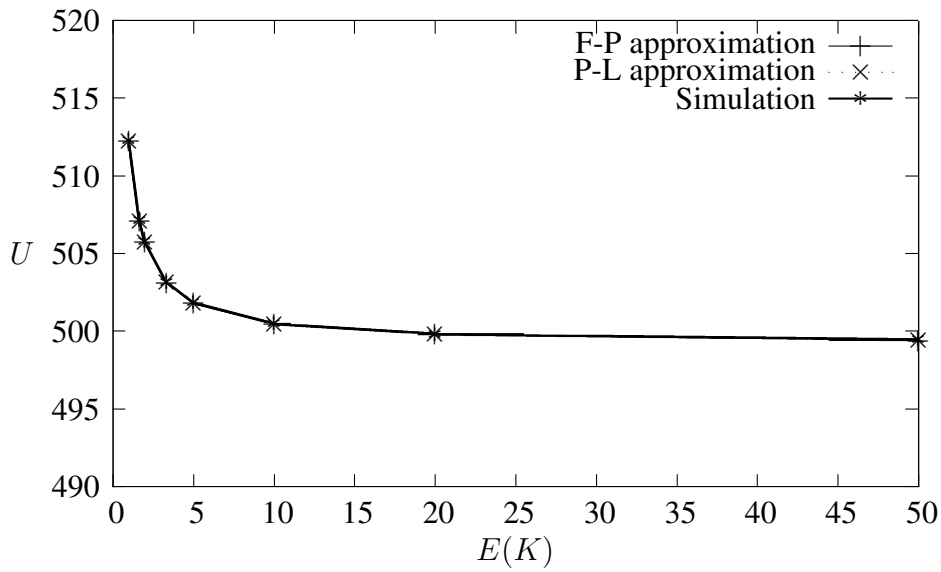

Figure 7. Large Structured Database: corruption time in days vs. average number of reads.

As expected, this large database takes much longer to corrupt: more than 500 *days*, compared to less than 72 hours. Moreover, it takes much longer to simulate: the total run time for the eight simulated points was about 18 hours. The two approximated plots took less that 10 minutes to compute. On the other hand, all three evaluation methods produced almost identical results.

It was slightly surprising to see that, for this system, the accuracy of the F-P approximation is as good as that of the P-L approximation. The likely explanation is in the structure of the database. The presence of a small and popular class of nodes is key to the propagation process. Once that class becomes corrupted, which happens relatively quickly, corruption is spread to other classes by incorrect reads, while the probabilities α and β do not change very quickly.

This observation is confirmed by the last experiment, illustrated in Figure 8, where the corruption time for the Large Structured Database is plotted against the fraction γ of edges to be corrupted. The number of reads per query is assumed to be geometrically distributed with parameter $r = 0.07$. Again, Each point is evaluated by the F-P approximation, the P-L

approximation with an increment of $h = 1$ hour, and by simulation.

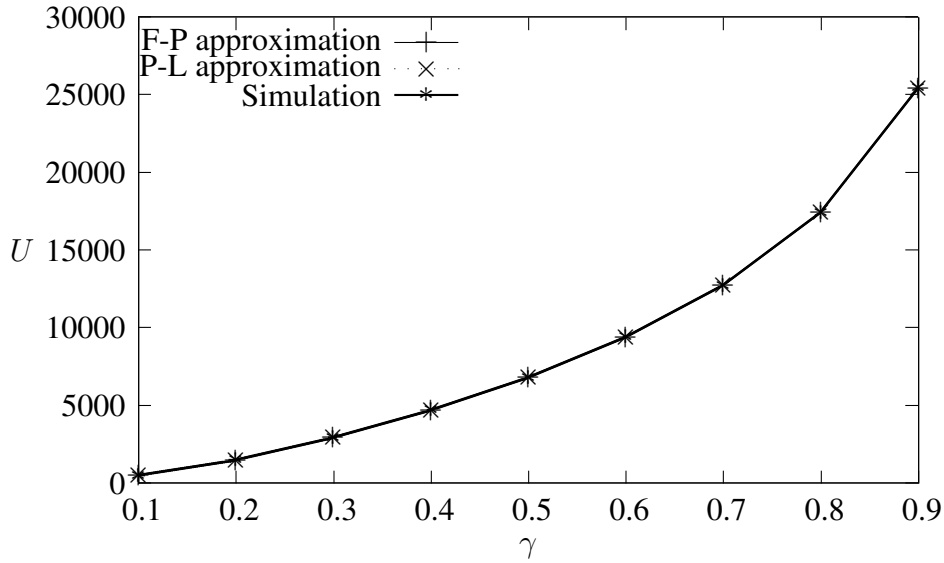

Figure 8. Large structured database: corruption time in days vs. fraction of corrupted edges.

The values of U now vary from about 500 days to more than 25000 days. The three plots are almost indistinguishable over the entire range. The nine simulated points took more than 95 hours of computer time.

The last three experiments suggest that structure helps the F-P approximation, regardless of the size of the database. The most difficult databases to model are the large and structureless ones. In those cases, the F-P approximation may fail, while the P-L approximation works well.

One might be tempted to dispense with the fixed-point iterations in each interval (t_{m-1}, t_m). Instead of determining $\bar{\beta}_m$ for that interval, it would be possible to use just the initial values $\mathbf{n}(t_{m-1})$ and $\beta(t_{m-1})$ to compute

$$\mathbf{n}(t_m) = \mathbf{n}(t_{m-1}) + h\mathbf{f}[\mathbf{n}(t_{m-1}), \beta(t_{m-1})] \,, \tag{31}$$

which would determine $\beta(t_m)$ However, that shortcut would ignore the fact that, as corruption spreads during the interval (t_{m-1}, t_m), $\beta(t)$ decreases. Consequently, the rate of spreading would be under-estimated, and hence the period U would be over-estimated. For the example in Figure 6, we found that the relative error of the shortcut, compared to the P-L approximation, was about 20%.

6. Conclusions

The problem that we have addressed – to construct and solve a quantitative model of database deterioration – is of considerable practical importance. The fluid approximation that has been developed is fast and provides accurate estimates of the time to corruption.

Two solution methods were developed: the Fixed-Point approximation which handles the entire period to corruption in a single step, and the Piecewise-Linear approximation which divides that period into a number of smaller intervals. The former is faster and is recommended for databases that are not too large, or have hierarchical structure. The latter is slower, but can be applied to any database, including large and structureless ones.

The model examined here may be extended in several directions. For example, it may be possible to assume that corrupted edges can be repaired by overwriting them with clean information. That would necessitate distinguishing between corrupted local edges and corrupted distributed edges. The time to absorption would be replaced by a first passage time to a certain subset of states. Also, one could assume that an edge that is the subject of a conflict does not immediately become half-corrupted, but remains in a special 'conflicted' state until it is next read. Only at that point does a conflicted edge become half-corrupted.

Both the above modifications can be handled quite easily by the methods presented here. Indeed, we have run some experiments which suggest that they do not materially affect the lifetime of the database.

A more substantial generalization would be to introduce a process of creation of new edges, and possibly of removal of existing ones. A fluid approximation of such a database should be possible, and would be a suitable topic for future research.

References

[1] Bailis, P., & Ghodsi, A. (2013). Eventual Consistency Today: Limitations, Extensions, and Beyond. *Acmqueue*, 11(3), 20–32.

[2] Brewer, E.A. (2000). Towards robust distributed systems. *Procs. 19th Annual ACM Symposium on Principles of Distributed Computing, Portland*, 16—19.

[3] Brouwer, L.E.J. (1911). Uber Abbildungen von Mannigfaltigkeiten. *Mathematische Annalen*, 71, 97–115.

[4] Buluc, A., Meyerhenke, H., Safro, I., Sanders, P. & Schulz, C. (2016). Recent Advances in Graph Partitioning. *Algorithm Engineering*, LNCS, 9220, 117–158.

[5] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati,G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., & Vogels, W. (2007). Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Operating Systems Review*, 41(6), 205–220.

[6] Ezhilchelvan, P., Mitrani, I., & Webber, J. (2018). On the degradation of distributed graph databases with eventual consistency. *15th European Performance Engineering Workshop (EPEW 2018), Paris*.

[7] Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *ACM SIGACT News*, 33(2), 51–59.

[8] Herlihi, M.P., & Wing, J.M. (1990). Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS*, 12(3), 463–492.

[9] Huang, J. & Abadi, D.J. (2016). Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings VLDB Endowment*, 9(7), 540–551.

[10] Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly media, ISBN 978-1-449-37332-0.

[11] Robinson, I., Webber, J., & Eifrem, E. (2015). Graph Databases, New Opportunities for Connected Data. O'Reilly Media, ISBN 978-1491930892.

[12] Schloegel, K., Karypis, G., & Kumar, V. (2003). Graph partitioning for high-performance scientific simulations. *Sourcebook of parallel computing*, 491–541.

[13] Shang, Z., & Yu, J.X. (2013). Catch the Wind: Graph workload balancing on cloud. *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, 553–564.

[14] Stanton, I., & Kliot, G. (2012). Streaming Graph Partitioning for Large Distributed Graphs. *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1222–1230.

[15] Vogels, W. (2009). Eventually Consistent. *Communications of the ACM*, 52(1), 40–44.