Improving OCC Performance Through Transaction Batching

ABSTRACT

OLTP systems can often improve throughput by *batching* transactions and processing them as a group. There are well-studied uses of batching for optimizations such as message-packing and group commits, however, there is little research on the benefits of a holistic approach of batching across a transaction's entire life cycle.

In this paper, we design an OLTP system using OCC that incorporates batching at multiple stages of transaction execution: Storage batching enables reordering of transaction reads and writes at the storage layer and validator batching enables reordering of transactions before validation. We formalize the problem of validator transaction reordering, and we propose several algorithms and policies to solve this problem. We carry out an in-depth experimental evaluation of the impact of storage and validator batching, and we show our techniques reduce the number of aborts and increase throughput. We show how different batching algorithms perform, how storage and validator batching interact with each other, and how they improve system-wide and individual transaction performance.

1. INTRODUCTION

Transaction processing is a fundamental aspect of database functionality, and improving OLTP system performance has long been a key research goal in our community. It is well-known that the throughput of OLTP systems can be increased through *batching*-based optimizations, whereby some component buffers a number of transactions or requests as they arrive and processes them as a group.

Batching can improve system performance for several obvious reasons. First, it increases the efficiency of communication by packing messages [24, 15]. Second, it amortizes the cost of system calls by condensing multiple requests into a single one, as in group commit [28, 14]. Third, it reduces the number of requests by discarding duplicate or stale ones, such as writes to the same record [21]. However, all of those are local optimizations based on low-level techniques.

We propose an OLTP system design that embraces batching as a core design principle at multiple stages of a transaction's execution. In particular, we explore the benefits of batching in optimistic concurrency control (OCC) [39] to reduce conflicts, and thus improve both system-wide and individual transaction performance. OCC is a popular concurrency control protocol [7, 8, 4, 46, 48, 12, 19, 6] due to its low overhead in low-contention settings [2]. However, it wastes resources [3] when conflicts are frequent. We show how batching reduces the number of conflicts and extends the applicability of OCC to higher-contention workloads.

Figure 1 shows a distributed OCC-based transaction processing system with centralized validation. The system consists of three components: processors, storage nodes, and a single validator. External clients issue transactions to the system. On arrival into the system, each transaction is assigned to a processor and enters its *read* phase. The processor sends read requests to the storage, executes the transaction, and performs "writes" to a local workspace. After it has processed the transaction, it sends information about the transaction's reads and writes to the validator.

The transaction now enters the *validation* phase. In OCC with *backward validation*, the validator checks whether the transaction conflicts with any previously committed transactions, and makes a "success" or "failure" validation decision. One example of a conflict that would cause validation to fail is a *stale read*. Suppose a transaction t reads an object, and a second transaction t' writes to the same object after t's read. If t' commits before t, t has a conflict, since it should have read the update t' made to the object but it did not. Thus t must fail validation.

If a transaction passes validation, the processor sends its writes to the storage; this is the *write* phase. Otherwise, the processor aborts and restarts the transaction.

The architecture of OCC with backward validation presents unique opportunities for batching because it contains multiple points of asynchronicity. There are three obvious times and locations to apply batching. The first is the processor in the transactions' *read* phase, where transaction requests can be batched before execution. There is recent work in the context of locking-based protocols that batches transactions and serializes them before execution to reduce overheads [55, 20, 44]; these techniques could be adapted and applied in OCC as well.

The second place we can batch is the validator, for transactions in the *validation* phase. If the validator buffers the validation requests as they arrive, it has the flexibility to choose a validation order. This allows the validator to re-

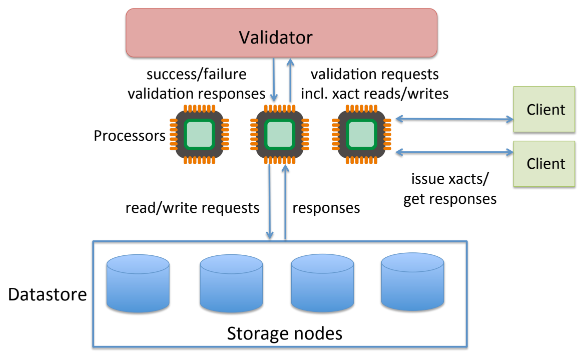

Validator
success/failure validation responses
validation requests incl. xact reads/writes
Processors
Client
Client
issue xacts/ get responses
read/write requests
responses
Datastore
Storage nodes

Figure 1: OCC System Architecture

duce the number of conflicts and aborts. Recall our previous example of a validation failure. A transaction t reads an object, and a second transaction t' writes to the same object after t's read. If t' arrives at the validator before t and commits, t must fail since it should have read the value written by t' but it didn't. Instead, with batching, if transactions t and t' are both in the same batch waiting for validation, we can choose to serialize t before t'. Thus, we avoid conflicts and both transactions can commit.

Third, the system can perform batching at the storage level. This affects already-validated transactions in their *write* phase as well as transactions that are still in their *read* phase. The storage can buffer read and write requests into batches as they arrive. If a batch contains multiple read and write requests for the same object, the system can apply all the writes first, in the serialization order. Next, it can process the reads. Prioritizing writes over reads is always optimal in the sense that it reduces the number of aborts as much as possible. This is because OCC reads come from uncommitted transactions, while writes come from validated transactions that will commit soon. Thus if the storage has both a pending read and a pending write on the same object, but schedules the read before the write, the reading transaction will see a stale value and is guaranteed to fail the validation.

Batching at these three levels in the system reduces aborts due to conflicts, and thus can increase throughput. However, it may also increase latency.

In this paper, we explore in-depth the benefits of transaction batching in OCC with backward validation. Since there is substantial existing work that applies directly to processor batching, we focus on storage and validator batching, which have received little research interest so far.

Our first contribution is an enhanced OCC system architecture that includes batching at the storage and the validator levels. We analyze the reasons for conflicts and aborts at each stage of a transaction's life cycle, and develop techniques to reduce these aborts through batching.

Our algorithmic focus is on validator batching, and more specifically on reordering within a batch at the validator. Optimal transaction reordering within a validator batch is NP-complete, as it can be formulated as the problem of finding a minimal feedback vertex set in a directed graph [35]. Our second contribution consists of two classes of greedy algorithms. These algorithms produce feedback vertex sets that are not minimal but are a close approximation to the optimal solution, and they are fast enough to be used in practice.

As mentioned before, the overhead of validator reordering increases transaction latency. Thus, even as validator reordering reduces the number of conflicts, it may hurt the end-to-end throughput of the system. Our two classes of greedy algorithms explore this trade-off. Additionally, we extend our algorithms to weighted versions that can incorporate features such as transaction priorities.

Our third contribution is a detailed experimental study of the impacts of storage and validator batching in OCC.

The remainder of the paper is organized as follows. In Section 2 we review the fundamentals of OCC with backward validation. In Section 3 we discuss challenges, opportunities and techniques for storage and validator batching. In Section 4, we introduce our algorithms for intra-batch transaction reordering at the validator. In Section 5, we present an extensive experimental study of batching throughout our system. We discuss related work in Section 6 and conclude in Section 7.

2. OCC BACKGROUND

In this paper, we use the term "optimistic concurrency control" to refer to the classical backward validation based protocol introduced in [39]. As explained in the introduction, this is a protocol where every transaction goes through three execution phases. First comes a *read* phase, where the transaction reads data from the storage and executes while making writes to a private "scratch workspace". Then the transaction enters a *validation* phase. If validation is successful, the transaction enters the *write* phase, when its writes are installed in the storage.

OCC validation enforces serializability; we only allow a transaction to pass validation if it can be serialized after all transactions that have already committed. The validator assigns each transaction a number (timestamp) and ensures that transactions are serialized in timestamp order.

To validate a transaction, we need to examine its reads and writes, and compare them to the reads and writes of previously committed transactions in the system. For a given transaction, we call the set of objects it has read its *read set* and the set of objects it has written its *write set*. When a transaction enters validation, its read set and write set are known, as are the read and write sets of all transactions that have previously passed validation and committed. For a transaction with timestamp i, we denote its read set by $RS(i)$ and its write set by $WS(i)$.

When validating a transaction with timestamp j (transaction $T(j)$), the validator needs to check for conflicts with any transaction with timestamp i (transaction $T(i)$) that have already committed (so $i < j$) and that overlapped temporally with $T(j)$, i.e. $T(i)$ hadn't committed when $T(j)$ started. $T(j)$ can be serialized after such a transaction $T(i)$ if one of the following conditions holds:

- $T(i)$ completed its write phase before $T(j)$ started its write phase, and $WS(i) \cap RS(j) = \emptyset$, or
- $T(i)$ completed its read phase before $T(j)$ started its write phase, and $WS(i) \cap RS(j) = \emptyset$ as well as $WS(i) \cap WS(j) = \emptyset$

If $T(i)$ and $T(j)$ overlap temporally, and $WS(i) \cap RS(j) \neq \emptyset$, we say there is a *read-write dependency* from $T(j)$ to $T(i)$. Intuitively, if there is such a dependency, $T(j)$ cannot be serialized after $T(i)$. In addition, we must ensure the writes of the two transactions are installed in the correct order to maintain consistency in the storage. That is, if they write to the same object, the updates must be applied in their serialization order. In the original OCC [39], this is ensured by putting the validation phase and the write phase into a critical section.

The batching techniques we study in this paper are applicable to most OCC systems; however, we make certain assumptions about the implementation of OCC validation, for simplicity and concreteness of our experimental setup. We clarify those assumptions now.

Given that our architecture (Figure 1) has a single validator, we assume the validation phase occurs in a critical section; that is, only one transaction is in the validation phase at a time. We do not require the write phase to be in a critical section; this means that write requests may arrive at the storage out of order. We deal with this using a *versioned* datastore; every item in the datastore is versioned and every write request is tagged with a version number equal to the writing transaction's timestamp. If the datastore receives a write request with version (timestamp) i and a higher-numbered version $j > i$ already exists for the object, the write request is ignored.

The above discussion implies that validation only needs to check, for every transaction $T(j)$ and all transactions $T(i)$ overlapping temporally with $T(j)$, where $i < j$, that $WS(i) \cap RS(j) = \emptyset$. Data versioning provides an easy way to determine whether $T(j)$ overlapped temporally with $T(i)$; every time a transaction performs a read, we tag the read with the version of the item that was read. If $T(j)$'s read set contains an item X and the read saw version k, the validator only needs to check the write sets of all $T(i)$ with $k < i < j$ to see whether they contain X. This use of versioning in validation is identical to the one described in [15].

3. BATCHING OVERVIEW

In this section, we explain how we can perform batching at the storage and the validator in an OCC system.

Batching involves buffering a number of requests as they arrive at some component of the system and processing them as a group. The goal of batching is to reduce the number of conflicts by reordering the operations in each batch, thus increasing the throughput. As explained in Section 2, aborts happen due to read-write dependencies involving a committed transaction $T(i)$ that wrote to some object, and a later transaction $T(j)$ that read the same object but did not see $T(i)$'s update. If $T(i)$ and $T(j)$ are in the same batch at the validator, we call this an *intra-batch* abort of $T(j)$; if they are in a different batch, we call it an *inter-batch* abort.

The server components – the storage and the validator – aim to reduce both intra-batch and inter-batch aborts. The storage batches the read and write requests, and the validator batches the validation requests.

Once a batch is fixed, the number of intra-batch aborts is determined by the amount of data contention among the requests in the batch and by the way we order and interleave the requests in processing. Thus the main optimization strategy is *reordering*. The storage manager reduces aborts by reordering individual read and write operations, while the validator reduces aborts by changing the serialization order of transactions. The overhead of the reordering impacts transaction latency, so the reordering algorithms must be as fast as possible.

To reduce the number of inter-batch aborts, the system can divide the transactions into batches in a careful manner; it can also privilege writes over reads to allow reads to see fresher values. However, at all times in the system, there are multiple batches in-flight. No component can thus eliminate inter-batch aborts completely and unilaterally. The worst case is a slow transaction that writes an object and passes validation, but whose write gets "stuck" in the system and doesn't make it to the storage for a long time. This transaction will conflict with – and cause inter-batch aborts of – all subsequent transactions that read the same item, until the write is actually applied.

We now explain the specific batching strategies that are available at the storage and the validator.

3.1 Storage Batching

At the storage layer, the optimal batching strategy is simple. We buffer a certain number of read and write requests into batches. When a batch is ready, either when there are enough requests or a timeout is reached, for each object we apply the highest-version write request on that object. It is safe to discard all other writes on that object, as explained in Section 2. Next, we handle all the read requests for the same object.

This strategy is optimal for reducing intra-batch aborts, as it ensures that all available writes by committed transactions are applied to all objects before we handle any read requests on these objects. The only way to reduce inter-batch aborts is to increase the batch size.

3.2 Validator Batching

The validator buffers and batches transaction validation requests as they arrive. The optimal strategy for reducing inter-batch aborts is to ensure that transactions accessing the same objects are always batched together, that is, to cluster them based on access patterns. Otherwise, the validator cannot do anything about conflicting transactions that are "spread out" across batches. There has been a lot of research on data clustering either online or offline [47, 18], especially for fine-grained data partitioning. Any of these techniques can be used for validator batch creation. Of course, clustering transactions into batches based on access patterns may increase the number of intra-batch aborts.

Once the batches are fixed, the validator can reduce intra-batch aborts by choosing a good validation (i.e., serialization) order. The validator may not be able to eliminate all intra-batch aborts, since it cannot actually delay the execution of a transaction. Rather, it is presented with read and write sets of transactions that have already run.

Intra-batch transaction reordering can be done with several goals in mind. We can simply minimize intra-batch aborts, i.e. maximize the number of transactions in each batch that commit. However, we may also want to prioritize certain transactions to have a greater chance of committing. For example, if we want to reduce the number of aborts *per transaction*, we can increase a transaction's priority every time it has to abort and restart. Priorities could also be related to external factors (e.g. a transaction's monetary value or its chance of committing). These choices suggest a range of possible policies; we explore them in the next section.

4. VALIDATOR REORDERING

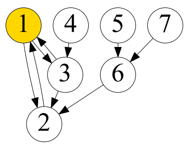


Figure 2: An example of a directed graph; node 1 forms a feedback vertex set.

In this section we present our algorithms for intra-batch validator reordering of transactions (IBVR). We formalize the problem and express it as an instance of the feedback vertex set (FVS) problem (Section 4.1). We give two greedy algorithms for finding feedback vertex sets in Section 4.2 and show an example of executing these algorithms on a graph in Section 4.3. Both algorithms are parameterized on *policies* that allow the validator to privilege certain transactions for committing; we discuss policies in Section 4.4.

4.1 Intra-Batch Validator Reordering (IBVR)

Formally, the problem of intra-batch validator reordering (IBVR) is as follows. A *batch* B is a set of transactions to be validated. We assume all transactions $t \in B$ are *viable*, that is, no $t \in B$ has a read-write dependency to a committed transaction. If there are non-viable transactions in B, they can be removed in preprocessing, as they must always abort due to inter-batch conflicts.

Given B, the goal of IBVR is to find a $B' \subseteq B$ of transactions that must abort due to intra-batch aborts. IBVR must also find a strict (i.e. asymmetric) total order $\prec$ on $B \setminus B'$ that *respects all read-write dependencies*; that is, for $t, t' \in B \setminus B'$, if $t \prec t'$, then there is no read-write dependency from t' to t.

The idea is that the validator processes each batch by running IBVR to identify B' and $\prec$, aborting all the transactions in B' and validating the transaction in $B \setminus B'$ in the order $\prec$. By the constraint we gave on $\prec$ above, and from the discussion in Section 2, $\prec$ is guaranteed to be a valid serialization order that allows all transactions in $B \setminus B'$ to commit.

There is always a trivial solution to any IBVR instance: we can choose B' to be all the elements of B except one, so that $B \setminus B'$ has cardinality one. Such solutions are not useful; therefore, every instance of IBVR is associated with an *objective function* on B', and the goal is to find a B' that maximizes the objective function. An example objective function could be the size of B' (smaller is better), or a more complex function that takes into account transaction weights (e.g. priorities).

4.1.1 IBVR using feedback vertex set

Our algorithms for IBVR are based on a graph representation of the transaction batches. Every batch B of viable transactions induces a *dependency graph* G; this is a directed graph whose nodes are the transactions in B and whose edges are read-write dependencies.

If G is acyclic, then there exists a $\prec$ on G that respects all read-write dependencies; we can construct it as follows. The edges in G provide a strict partial order on B; for two nodes $t, t' \in G$, we order t before t' if and only if t' is reachable from t (they cannot both be reachable from each other by acyclicity). We complete the construction of $\prec$ by choosing any linear extension of this partial order.

We need to show that if $t \prec t'$, then there is no read-write dependency from t' to t. Suppose $t \prec t'$ and consider the nodes t and t' in G. Assume for a contradiction there is a read-write dependency from t' to t. Then by construction, G contains an edge from t' to t, and by construction of $\prec$ we have $t' \prec t$; however, this contradicts the statements that $t \prec t'$ and that $\prec$ is strict.

Our family of algorithms for IBVR is based on constructing the dependency graph G and finding a *feedback vertex set* (FVS) [35] on G. A FVS is a subset of vertices whose removal makes the graph acyclic. For example, consider the graph in Figure 2. The vertex 1 forms a FVS since the graph becomes acyclic after removing vertex 1 and its incoming and outgoing edges.

Finding a minimal-size B' for IBVR is exactly the problem of finding the minimal (smallest-size) feedback vertex set on G. If we have a more complex objective function for IBVR, we can push the objective function into the FVS computation; we assign weights to the nodes to represent the desired transaction priorities, and look for a minimum-weight FVS.

Once we find the FVS B', removing the nodes in B' from G yields an acyclic graph that allows us to determine the desired serialization order $\prec$.

4.1.2 Complexity of FVS

The directed graph FVS (DFVS) problem is well-studied as it has many applications, including deadlock detection, program verification, and Bayesian inference. It was listed among the first set of NP-complete problems [35]. The state-of-the-art study [11] on the hardness of the problem has shown that it is *fixed-parameter tractable*, that is, it can be solved in time $O(f(k)n^c)$ for a function $f(k)$ and a constant c where the number k and the function $f(k)$ are independent of the instance size n. [11] proposes an exact algorithm for finding a minimal DFVS in $O(4^k k! n^{O(1)})$, where n is the number of vertices and k is the size of the minimal solution.

DFVS is also APX-hard [34]. This means there exists a constant $c > 1$ such that the existence of a polynomial-time approximate algorithm that achieves an approximation ratio strictly smaller than c would imply $P = NP$. The best hardness result [17] shows that it is NP-hard to approximate DFVS within a factor of 1.36, by an easy reduction from vertex cover. While it is still open whether there exists a constant factor approximation for DFVS, it is shown that it is NP-hard to approximate DFVS within any constant factor if the Unique Games Conjecture is true [27]. The state-of-the-art algorithm has an approximation factor of $O(\log \tau \log \log \tau)$, where τ is the size of the exact solution, and a factor of $O(\log \tau^* \log \log \tau^*)$ for weighted directed graphs, where τ^* is the weight of the exact solution.

Despite the above hardness results, empirical studies have shown that simple greedy heuristics perform reasonably well when compared to more advanced algorithms [13].

4.2 Our IBVR Algorithms

All our IBVR algorithms begin by constructing the dependency graph G. We start with a set of transactions that have been batched at the validator and construct B by discarding all non-viable transactions. We can identify such transactions by validating each transaction individually.

Next, we create one node per transaction, and one edge per read-write dependency between a pair of transactions. To determine whether a read-write dependency holds from

```
1  Algorithm GreedySccGraph(G, P)
       Input: Directed graph $G$, policy $P$
       Output: $V$, a feedback vertex set for $G$
2      $V \leftarrow \emptyset$
3      $G' \leftarrow trim(G)$
4      $SCC = StronglyConnectedComponents(G')$
5      for $S \in SCC$ do
6          $V \leftarrow V \cup GreedyComponent(S, P)$
7      end
8      return $V$
9  Algorithm GreedyComponent(S, P)
       Input: SCC $S$, policy $P$
       Output: $V'$, a feedback vertex set for $S$
10     if $S.size == 1$ then
11         return $\emptyset$
12     end
13     $V' \leftarrow \emptyset$
14     $v \leftarrow select(S, P)$
15     $V' \leftarrow V' \cup v$
16     $S' \leftarrow delete(S, v)$
17     $V' \leftarrow V' \cup GreedySccGraph(S', P)$
18     return $V'$
```

Algorithm 1: SCC-based greedy algorithm

```
1  Algorithm GreedySortGraph(G, P, k)
       Input: Directed graph $G$, policy $P$, multi factor $k$
       Output: $V$, a feedback vertex set for $G$
2      $V \leftarrow \emptyset$
3      $G \leftarrow trim(G)$
4      while $G \neq \emptyset$ do
5          if $G.size \leq k$ then
6              $w \leftarrow$ a random node from $G$
7              $V \leftarrow V \cup (G \setminus \{w\})$
8              break
9          end
10         $Q \leftarrow sort(G, P)$
11         for $i = 1; i \leq k; ++i$ do
12             $V \leftarrow V \cup Q[i]$
13             $G \leftarrow remove(G, Q[i])$
14         end
15         $G \leftarrow trim(G)$
16     end
17     return $V$
```

Algorithm 2: Sort-based greedy algorithm

transaction t' to t, we check whether $WS(t) \cap RS(t') \neq \emptyset$. If so, we add an edge from t' to t. We can implement this by creating a hash table from the write sets and probing it with the read sets, or using bitvector intersection computations. The time complexity of building G is $O(|B| + |E|)$, where $|B|$ is the number of nodes in the graph, i.e., the size of B, and $|E|$ is the number of edges in the graph, i.e. the number of read-write dependencies.

We now process G to find a feedback vertex set. We need fast FVS algorithms to avoid increasing both transaction latencies and the number of aborts. A brute-force search for a FVS is prohibitively slow in practice, although it can serve as an experimental baseline.

We propose two greedy algorithms for finding feedback vertex sets – one based on the graph's strongly connected components (SCCs) and the other based on sort. We also introduce a hybrid algorithm that is slower but more precise than either greedy algorithm.

Both before and during the execution of our FVS algorithms, we *trim* the graph to remove all nodes which have no incoming edges and/or no outgoing edges; such nodes cannot participate in any cycles and are unnecessary to include in any FVS.

4.2.1 SCC-based greedy algorithm

The intuition behind our first algorithm is that each cycle is contained in a strongly connected component of the graph, so we can process the components in parallel. After preprocessing, we partition the graph into SCCs. For a graph with V vertices and E edges, we can do this in time $O(|V| + |E|)$ using Tarjan's SCC algorithm [53].

Any SCCs that consist of a single node do not need to be considered further, as the one vertex involved cannot be on a cycle. If the SCC contains multiple vertices, we choose a vertex to remove according to some *policy*. The policy is a ranking function on vertices and we choose the top-ranked vertex to remove. We now recurse on the remaining graph.

Algorithm 1 shows this in more detail. We begin by trimming and partitioning the graph into SCCs (lines 3-4). We process each SCC S using $GreedyComponent(S, P)$ (lines 5-7). This subroutine starts by eliminating SCCs of size one (lines 10-12). Next, it chooses a vertex v from S, i.e. the top-ranked vertex under P (line 14). It includes v in the FVS of S (line 15), removes it from S (line 16), and recursively calls *GreedySccGraph* on the remaining graph (line 17). Finally, it returns the union of all the FVSs obtained in processing S (line 18). When the top-level procedure $GreedyComponent(G, P)$ has processed all the SCCs of G, it returns the union of the FVSs obtained (line 8).

The policy P is the heart of the algorithm and affects both its accuracy and running time. Fortunately, there is no trade-off between accuracy and running time; a more accurate policy will lead to a smaller FVS and faster termination. We discuss possible policies in Section 4.4.

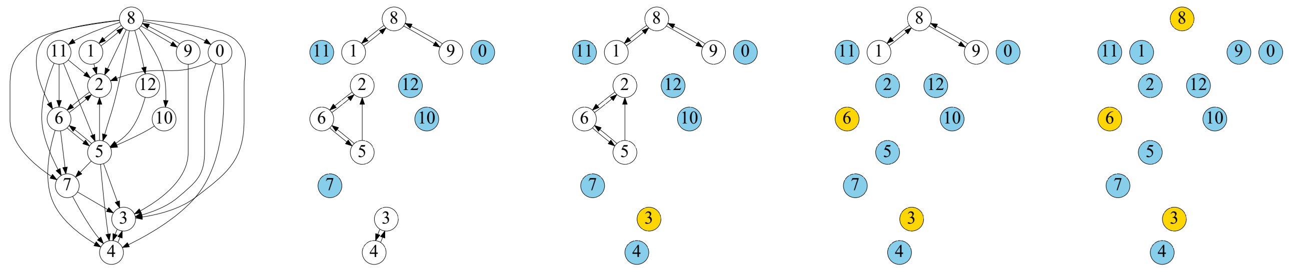
4.2.2 Sort-based greedy algorithm

Our first algorithm relies on a SCC partitioning routine that takes linear time in the size of the graph. As this routine is called several times during the course of the algorithm, it can cause a non-trivial overhead. Thus, we propose a second, faster greedy algorithm which uses sort-based heuristics to remove nodes.

Through empirical tests of the SCC-based greedy algorithm we found that nodes with certain properties are very likely to be included in a FVS. Such nodes generally have high in-degree and/or out-degree, for example. Our second algorithm is based on this observation; it sorts the nodes according to some ranking function (i.e., policy) P, and includes the top-ranked k nodes in the FVS. We call k the *multi factor* of the algorithm. The algorithm removes these nodes and iterates on the remaining graph.

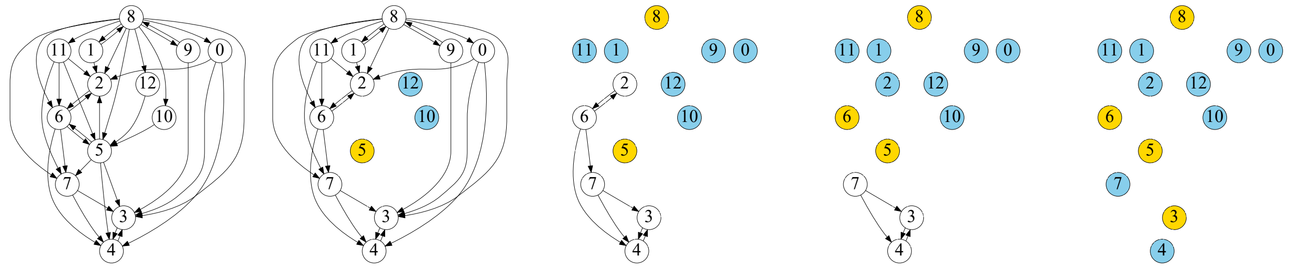
Algorithm 2 shows this in more detail. We trim the graph (line 3); if the graph has no more than k nodes, we include all but one (random) node in the FVS (line 5-9). Otherwise, we sort the nodes into a queue Q using P, and include the top-ranked k nodes in V (line 10-14). After removing the selected nodes from G, we trim the remaining the graph again (line 15). We repeat until the graph is empty (line 4).

This algorithm is faster but less accurate than the SCC-



(a) original (b) partition into SCCs (c) add 3 to FVS (d) add 6 to FVS (e) add 8 to FVS

Figure 3: An example of the SCC-based greedy algorithm using `prod-degree` policy



(a) original (b) add 5 to FVS (c) add 8 to FVS (d) add 6 to FVS (e) add 3 to FVS

Figure 4: An example of the sort-based greedy algorithm using `prod-degree` policy and multi factor 1

based greedy algorithm. However, our experiments in Section 5 show that it produces comparable results with the SCC-based algorithm in practice.

4.2.3 Hybrid algorithm

We can combine the SCC-based greedy algorithm and the precise brute-force FVS search into a *hybrid algorithm*. This is similar to the SCC-based greedy algorithm but runs a precise, brute-force FVS search in certain cases. Instead of processing all SCCs via the $GreedyComponent$ subroutine (lines 5-7 of Algorithm 1), it runs the precise search when processing SCCs that are smaller than a certain *threshold*, and the greedy subroutine $GreedyComponent$ on SCCs that are larger than the threshold. Adjusting the threshold allows us to trade off precision versus running time.

4.3 Example

Figures 3 and 4 show executions of the two greedy algorithms on the same graph. Both algorithms use a policy where the ranking of a node is the product of its in-degree and its out-degree, i.e. `prod-degree`.

Figure 3 shows the SCC-based greedy algorithm. The graph cannot be trimmed, so we partition it into SCCs. We remove all SCCs of size 1 – namely nodes 0, 7, 10, 11, and 12 (Figure 3b). There are three remaining SCCs. We first look at the component containing nodes 3 and 4. Since nodes 3 and 4 have the same product of in-degree and out-degree, we can add either one of them to the FVS. We choose node 3. Now node 4 has neither incoming nor outgoing edges, so it is trimmed (Figure 3c). We repeat the process with the other components. For the SCC containing nodes 2, 5, and 6, we add node 6 to the FVS, as it has the largest product of in-degree and out-degree among the three nodes in this SCC. We now trim nodes 2 and 5 (Figure 3d). Finally, we remove node 8 from the last component, and trim nodes 1 and 9 (Figure 3e). This leaves us with a final FVS consisting of nodes 3, 6, and 8.

Figure 4 shows the same example using the sort-based greedy algorithm, with $k = 1$. After the first sort, we add node 5 to the FVS since it has the highest product of in-degree and out-degree. After eliminating node 5, nodes 10 and 12 have only incoming edges, and get trimmed (Figure 4b). We sort the remaining nodes. This time, we add node 8 to the FVS, and trim nodes 0, 1, 9, 11 (Figure 4c). We repeat this process with the remaining nodes until the graph is empty. This yields a FVS consisting of node 3, 5, 6, and 8 (Figure 4e), which contains one more vertex than the FVS we obtained with the SCC-based algorithm.

4.4 Algorithm Policies

We now discuss policies for our greedy algorithms. As mentioned, a policy is a ranking function on vertices of the graph, and a good policy is one that ranks highly vertices which are likely to be in a desirable FVS.

We first discuss the policies that aim at minimizing the number of conflicts, i.e. the size of the FVS. The simplest policy is `random` that assigns all nodes random rankings. Alternatively, we can rank nodes using degree-based heuristics, which use the intuition that the removal of a node will break many cycles if the node is high in some measurement of its graph degree. Such heuristics have been shown to work well for FVS computation [13]. For example, the policy `max-degree` chooses the node with the largest degree (either in-degree or out-degree), `sum-degree` chooses the node with the largest total degree (in-degree plus out-degree), and `prod-degree` chooses the node with the largest product of in-degree and out-degree.

More sophisticated policies are possible if the system is optimizing for something beyond maximizing the number of commits. For example, we may want to bound the transactions' tail latency; we can do that by incorporating latency information in our policies. We can rank transactions based on how many times they have been aborted and restarted, breaking ties using degree-based heuristics; thus, transactions that have been restarted many times do not enter the FVS and are allowed to commit. While this policy reduces the transaction tail latencies, it can increase the non-tail latencies, due to its suboptimality in finding the feedback vertex set. To alleviate this problem, we can devise policies that combine information about a transaction's number of restarts and its graph degree. For example, we can compute the ranking of a vertex as the product of its in-degree and out-degree divided by an exponential function of the number of restarts of the corresponding transaction.

Finally, we can devise policies to handle *hard transactions*. In many workloads there are transactions that are naturally more prone to conflicts, such as transactions that access many items and/or that access "hot" items. If we use graph degree based policies, such transactions are likely to be included in the FVS and aborted repeatedly. To avoid starvation, we can adjust our policies to increase the probability that these transactions can commit. For example, we can approximate the "hardness" of a transaction by the size of its read and write sets and include that as a weighting factor in the policy.

5. EVALUATION

In our experimental evaluation, we set out to study the effect of batching at the storage and the validator, the performance of our validator reordering algorithms and policies, and the impact of system configuration parameters. In particular, we asked the following questions:

1. How well do our validator reordering algorithms from Section 4.2 perform? How does batching and reordering at the validator using these algorithms affect the end-to-end system performance?
2. How does the batch size affect performance? Using larger batches should give the validator and the storage more opportunities for reordering but it should also increase transaction latency, leading to more conflicts.
3. How does storage and validator batching affect the system throughput, abort rate, and latency?
4. How do the different algorithm policies presented in Section 4.4 affect the system performance?
5. What is the performance impact of creating validator batches based on access patterns (Section 3.2)?

5.1 Implementation and Experimental Setup

Our system consists of four components: a transaction generator, a processor thread, a storage thread, and a validator thread. The threads communicate through queues of requests; that is, there is a generator queue, a processor queue, a storage queue, and a validator queue.

The transaction generator continuously produces new transactions until the system reaches the maximum permitted concurrency level.

The processor multiplexes transactions, receives transaction requests from the transaction generator, sends read / write requests to the storage, sends validation requests to the validator, and replies to the transaction generator. It also restarts aborted transactions; thus, it only communicates commit decisions to the transaction generator.

The storage thread continuously processes read and write requests. When batching is enabled, it buffers a batch of requests. When it processes a batch, it first executes all the write requests in the batch (discarding a write if a newer version exists in the storage), and then all the read requests.

The validator performs backward validation. It caches the write sets of committed transactions for future validations until these writes are overwritten by later committed transactions. When batching is enabled, the validator collects the requests into a batch as they arrive, and runs one of the algorithms described in Section 4 to determine a serialization order. Every transaction that passes validation is assigned an integer *commit timestamp*, which corresponds to the version number of the updates it will install in the storage.

When *clustering* is enabled, the validator creates transaction batches by grouping together transactions based on their access patterns. It processes each batch when either there are enough transactions in the batch, or a timeout for that batch is reached.

The system is implemented in Java. All the experiments run on a multicore machine, with Intel Core i7-5600U CPU @2.60GHz and 8GB RAM. We use a key-value model for the storage, which we implement as an in-memory hash table. As a default, we populate the database with 100K items, each with an 8-byte key. The values are left null as they are not relevant to our evaluation. We generate a transactional workload where each transaction reads 5 items and writes to 5 items. The reads and writes are drawn from a Zipfian distribution, which is implemented based on [26]. We limit the concurrency level to 600, i.e. at any time there are at most 600 live transactions in the system. The default batch size is 150 for both storage and validator.

The validator uses the sort-based greedy algorithm with the `prod-degree` policy and multi factor 2 unless otherwise specified. The *baseline* configuration represents the system running with both storage and validator batching turned off.

All our experimental figures show the averages of three runs, each lasting for 30 seconds in between a 5-second warm-up and a 5-second cool-down time. The deviation was not significant in any of the experiments, so we omit the error bars for clarity.

5.2 Validator Reordering Algorithms

In our first experiment, we investigate the performance of the feedback vertex set algorithms from Section 4.2.

We begin with a comparison of the algorithms with respect to their raw performance - i.e., their accuracy and running time. We run the algorithms on graphs constructed as described in Section 4.1, using a variety of workloads generated from a Zipfian distribution.

We first test the algorithms offline on dependency graphs constructed at the validator when running the system. Each dependency graph is constructed from a batch of transactions at the validator, excluding non-viable transactions, i.e. transactions that have inter-batch conflicts. We run the system for 30 seconds (aside from a 5-second warm-up and a 5-second cool-down time) to get a trace of the dependency graphs. We compare the averages of the size of the feedback vertex set and the running time per dependency graph on each trace.

We test the SCC-based greedy algorithm with the `max-degree`, `sum-degree` and `prod-degree` policies; we refer to

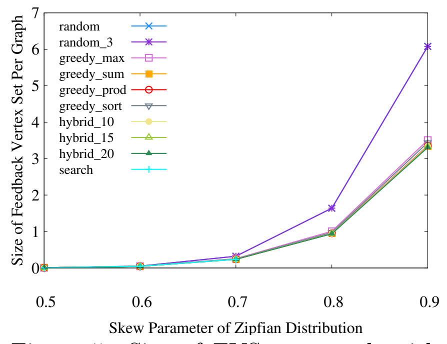


Figure 5: Size of FVS per graph with different algorithms

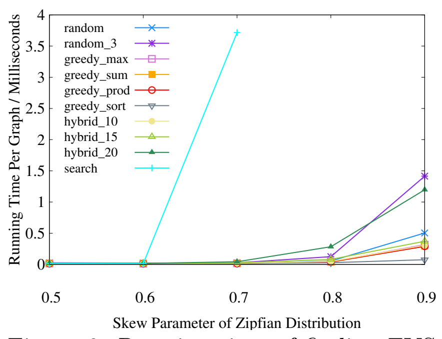


Figure 6: Running time of finding FVS with different algorithms

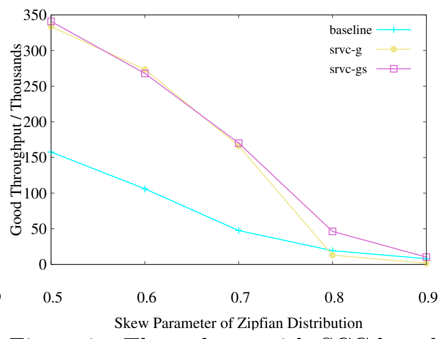


Figure 7: Throughput with SCC-based and sort-based greedy algorithms

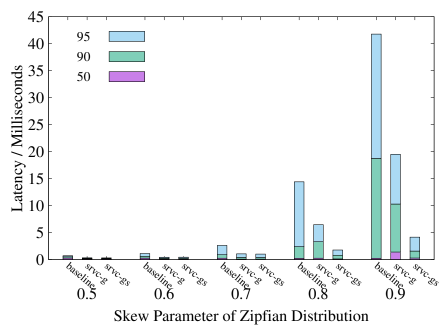


Figure 8: Percentile latency for greedy algorithms

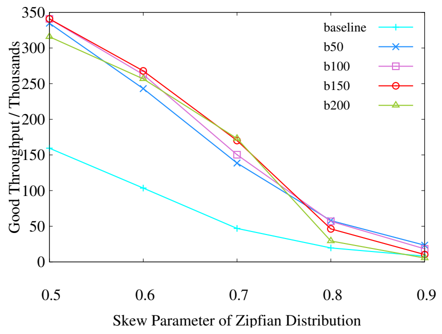


Figure 9: Throughput with various batch sizes

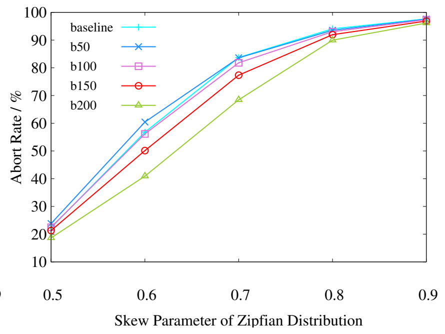


Figure 10: Abort rate with various batch sizes

those implementations as *greedy_max*, *greedy_sum*, and *greedy_prod*. We also test the sort-based greedy algorithm *greedy_sort* (using the `prod-degree` policy for sorting and multi factor 2), and the hybrid algorithm *hybrid_m*. The hybrid algorithm uses *greedy_prod* as a subroutine when the size of the SCC is larger than m, and switches to the brute force search when the size is under m. Thus, by increasing the threshold, we can progressively approximate the optimal solution.

We test these algorithms against several baselines. *search* is an accurate, brute force FVS search algorithm. *random* is the SCC-based greedy algorithm using the `random` policy, i.e., it removes vertices at random from the SCCs of the graph. Our final baseline runs the random algorithm multiple times (3 times for *random_3*) on each graph, and returns the smallest FVS across all the runs. This mitigates the worst-case impact of bad random choices.

Figure 5 shows the average size of the feedback vertex set found by each algorithm. The brute force search algorithm is so slow that it cannot produce results once the skew factor increases beyond 0.7 as the graph's SCCs become too large.

The figure shows that the *random* baseline computes a FVS whose size is almost twice as large as the greedy and the hybrid algorithms. Running the random algorithm multiple times produces similar results. This confirms the theoretical results which show that finding good feedback vertex sets is hard. The greedy algorithms, on the other hand, produce very accurate results. The average size of the feedback vertex set is almost identical to that of the brute force search when the skew factor is no larger than 0.7, and is very close to the best hybrid algorithm (*hybrid_20*, i.e. one that uses the brute force search when the size of the SCC is no larger than 20). Among the greedy algorithms, *greedy_prod* is consistently better than other greedy algorithms, although the difference is small.

Figure 6 shows the running time of the algorithms. The running time of the hybrid algorithm depends on the threshold for switching to brute force search. Thus, *hybrid_20* and *hybrid_15* have a longer running time than other algorithms, while the running time of *hybrid_10* is comparable to the SCC-based algorithms. Each of the SCC-based algorithms (*greedy_max*, *greedy_sum*, *greedy_prod*, *random*) has a similar running time. The random algorithm takes longer than the greedy algorithms because it removes more nodes and thus requires more computation. The running time of *random_3* is three times that of *random*, since it runs the random algorithm three times. The sort-based greedy algorithm (*greedy_sort*), while slightly less accurate than the SCC-based greedy algorithms, requires less than 26% of the running time of these algorithms.

We compare the best SCC-based algorithm (*greedy_prod*) against the sort-based greedy algorithm in terms of the end-to-end performance of the system. Figure 7 shows the good throughput of the system with *greedy_prod* (*srvc-g*) and *greedy_sort* (*srvc-gs*). In both cases, storage batching is enabled, and the greedy algorithm policy is set to minimizing the number of conflicts, i.e. the size of the feedback vertex set. The *baseline* line shows the throughput with both storage and validator batching disabled.

The two greedy algorithms have similar throughput when the skew is moderate. However, *greedy_prod* degrades significantly with high data contention (skew factor 0.8 and 0.9). This is because while *greedy_prod* is slightly more accurate, it takes much longer to run. This increases transaction latency and leads to more aborts, especially when the chance of conflicts is high. *greedy_sort* consistently gives the

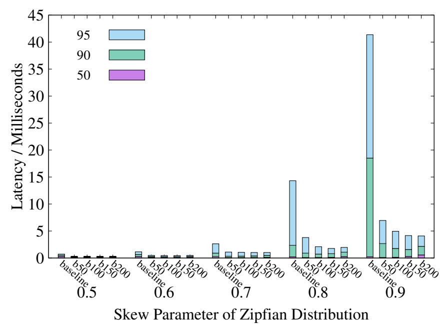


Figure 11: Percentile latency with various batch sizes

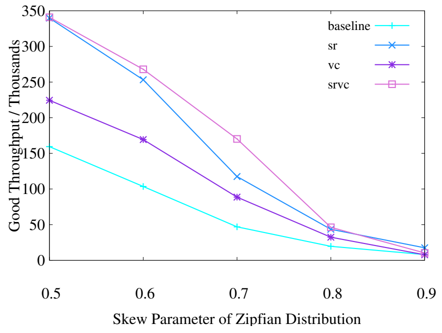


Figure 12: Throughput under workloads of Zipfian distribution

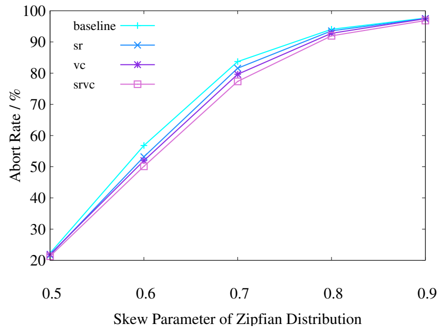


Figure 13: Abort rate under workloads of Zipfian distribution

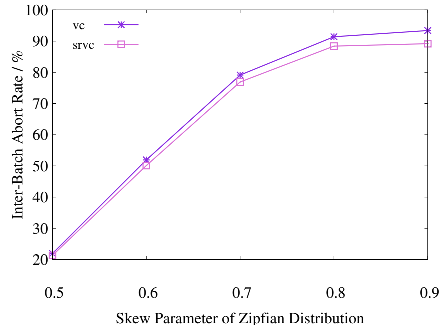


Figure 14: Inter-batch aborts under workloads of Zipfian distribution

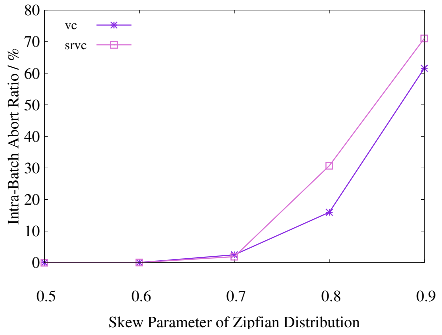


Figure 15: Intra-batch aborts under workloads of Zipfian distribution

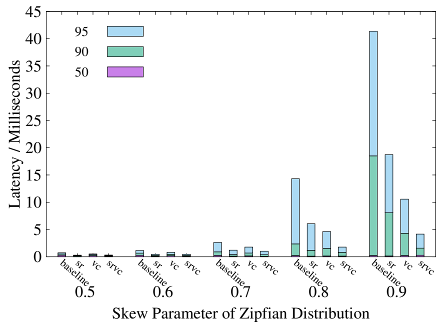


Figure 16: Percentile latency under workloads of Zipfian distribution

highest throughput over all the workloads, due to its high accuracy and low running time.

Figure 8 shows transaction latencies by percentile. The percentile latency of *greedy_sort* is much lower than that of the baseline and *greedy_prod*. This is consistent with the throughput data we have seen.

In summary, the sort-based greedy algorithm is much faster than the "smarter" algorithms and only slightly worse in terms of accuracy. This means it gives the best end-to-end system performance. For this reason, all subsequent experiments use the sort-based greedy algorithm with a `prod-degree` policy unless otherwise specified.

5.3 Batch Size

In this experiment, we explore how the batch size affects system performance. Smaller batch sizes should give lower latencies but they offer fewer opportunities for reordering, leading to more aborts.

We configure the system to perform both storage and validator batching with batch sizes from 50 to 200 ($b50$, $b100$, $b150$, $b200$), using the same batch size at the storage and the validator. As before, *baseline* is the system with both kinds of batching turned off.

As expected, increasing the batch size always reduces the number of aborts due to more opportunities for reordering; this is seen in Figure 10.

Figure 9 shows the throughput of the system with different batch sizes as data skew increases. As expected, the throughput first rises as we increase the size of the batch, and then degrades when the batch becomes too large. Thus, the throughput with batch size 150 is higher than with batch size 200. The percentile latency displays a similar pattern, as shown in Figure 11. Again the best batch size is 150; however, using batching always gives higher throughput and a better latency profile than the baseline.

Given the above results, a batch size of 150 appears optimal for our configuration; we use this batch size in the remainder of our experiments.

5.4 Storage and Validator Batching

Next, we perform a detailed analysis of the effects of storage and validator batching. We configure the system in several different modes: no batching (*baseline*), storage batching (*sr*), validator batching with the `prod-degree` policy that maximizes the number of commits (*vc*), and both storage and validator batching (*srvc*).

As explained in Section 3, batching and reordering affect the abort rate by reducing inter-batch and intra-batch aborts. The number of inter-batch aborts is affected by system-wide transaction latency, the freshness of the transactions' reads, and their access patterns. Validator reordering reduces the number of intra-batch aborts; however, storage reordering can increase the number of such aborts because it reduces inter-batch aborts (and thus more viable transactions end up in validator batches rather than aborting due to inter-batch conflicts). The overall good throughput of the system is a function of both the transaction latency and the abort rate.

Figure 12, Figure 13, and Figure 16 show the good throughput, the abort rate, and the percentile latency of different system modes under a variety of data skew parameters. Figure 14 shows the number of inter-batch aborts, while Figure 15 shows the intra-batch abort ratio, i.e. the number of transactions that commit divided by the number of viable transactions in a batch.

Overall, using batching at the storage and/or the validator

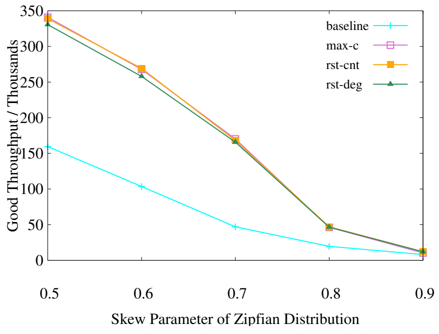


Figure 17: Throughput with tail latency optimized policies

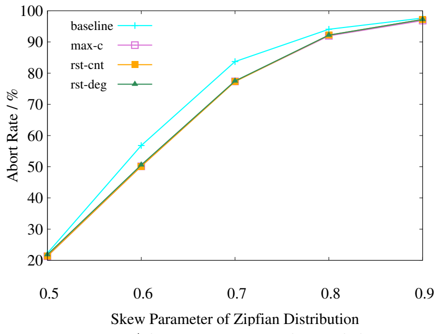


Figure 18: Abort rate with tail latency optimized policies

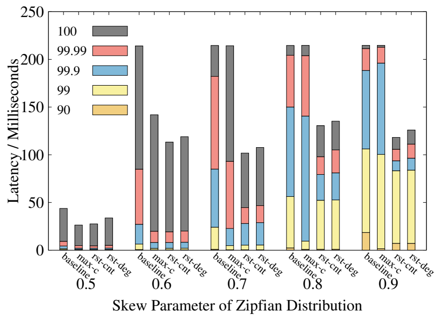


Figure 19: Percentile latency with tail latency optimized policies

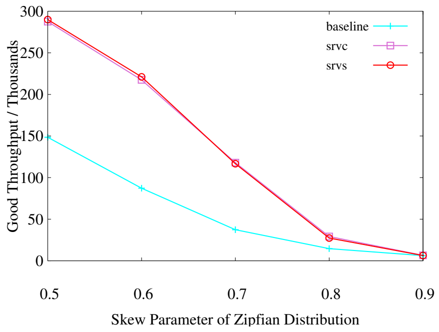


Figure 20: Throughput with heterogenuous workloads

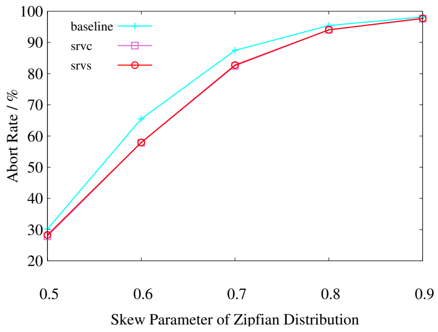


Figure 21: Abort rate with heterogenuous workloads

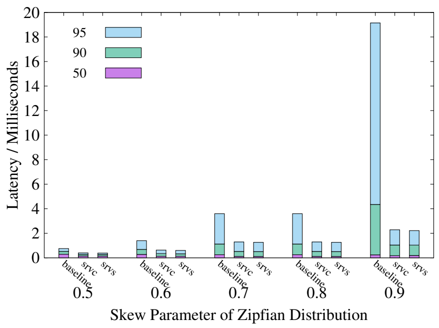


Figure 22: Percentile latency with heterogenuous workloads

consistently leads to significant improvements in throughput, abort rate, and latency profiles over the baseline. When batching is enabled (*sr* and *srvc*), the throughput is 2.1-3.7 times that of the baseline (*baseline*). In addition, using batching at the validator always reduces the abort rate, as shown in Figure 13, and gives a better latency profile (Figure 16).

When the data contention is very low, i.e. when the skew factor is 0.5, the abort rate is low (Figure 13). In addition, the abort rate is dominated by inter-batch conflicts (Figure 14) and intra-batch conflicts are rare (Figure 15). Thus, the storage-batching-only mode (*sr*) gives similar throughput as with *srvc*. As the data skew increases, so does the number of intra-batch conflicts and aborts; the overhead of validator batching starts to pay off. In a medium-contention setting, using both validator and storage batching (*srvc*) gives the best throughput. When the data contention becomes extremely high, i.e. the skew factor reaches 0.9, the number of intra-batch conflicts that cannot be resolved by validator reordering increases. The validator reordering takes more time due to denser graphs, while bringing less benefit. Thus, the best throughput is achieved by using storage batching only (*sr*).

To summarize, it is always beneficial to enable storage batching since this technique reduces inter-batch aborts at a minimal cost. While validator batching consistently gives a better abort rate and percentile latency, it is most effective in mid-contention settings, when the reduction of intra-batch conflicts that it brings is sufficient to justify its cost.

5.5 More Policies for Validator Reordering

In the next set of experiments, we explore validator reordering with the more complex policies presented in Section 4.4; specifically, we look at policies that aim to reduce the transaction tail latency, and at policies that help "hard transactions" with many dependencies commit.

5.5.1 *Reducing tail latencies*

We explore the possibility of reducing transaction tail latency with latency-specific policies. Our baselines are the `prod-degree` policy that maximizes the number of commits (*max-c*) as well as the *baseline* with no batching.

Our first tail-latency aware policy (*rst-cnt*) favors transactions that have already been aborted and restarted. When choosing a node to include in the feedback vertex set, it chooses the one with the smallest number of restarts, breaking ties using `prod-degree`.

The second latency-aware policy combines the number of restarts and a degree-based measurement of a transaction into a single value. It computes the weight of a node as the product of in-degree and out-degree over the exponential of the number of restarts with base 2. When choosing a node to include in the feedback vertex set, the nodes are sorted in descending order by their weights. Thus, a node with a high degree product can have its weight reduced if the corresponding transaction has restarted many times.

Figure 17 and Figure 18 show the transaction throughput and abort rate. As expected, the impact of tail-latency aware policies on overall transaction policies is slightly worse than the `prod-degree` policy due to its suboptimality in finding the minimal feedback vertex set.

Figure 19 shows the tail latency from 90% to 100%, i.e. the latency threshold for up to 90% and up to 100% of the transactions. While maximizing the number of commits, the `prod-degree` policy can produce worse tail latencies than the baseline. This is because it is unaware of the restart times

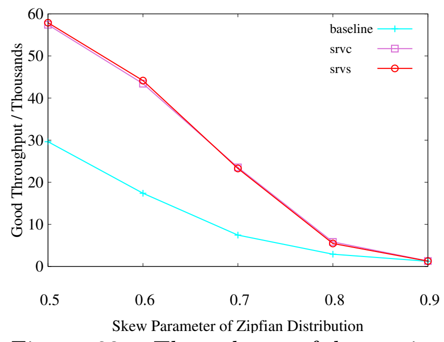


Figure 23: Throughput of large size transactions

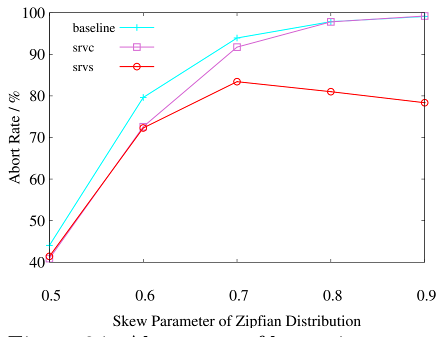


Figure 24: Abort rate of large size transactions

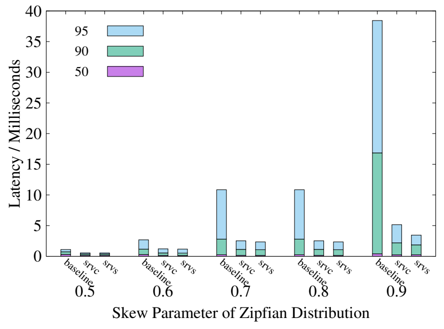


Figure 25: Percentile latency of large size transactions

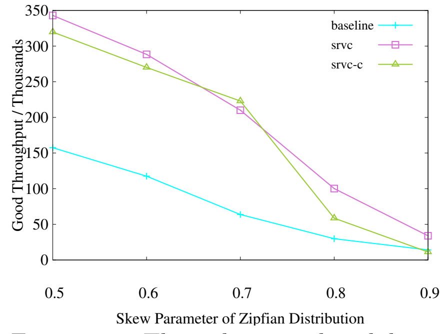


Figure 26: Throughput with validator clustering

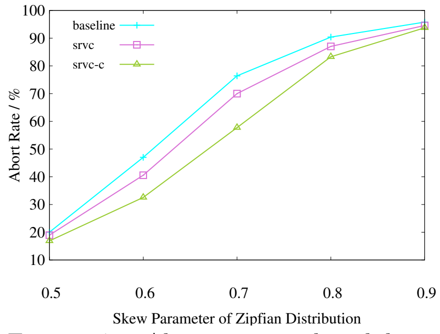


Figure 27: Abort rate with validator clustering

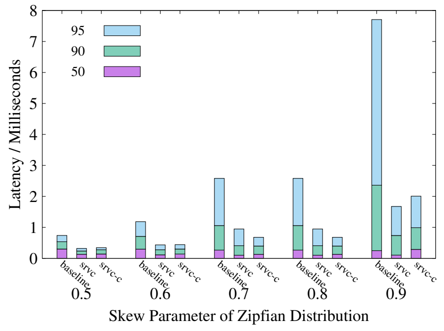


Figure 28: Percentile latency with validator clustering

of transactions, and it can discriminate against transactions that are inherently hard to commit, e.g. because they access many hot items.

In contrast, the latency-aware policies *rst-cnt* and *rst-deg* consistently perform significantly better than the *baseline* and the *max-c* policy, especially for tail latencies from 99.9% to 100%. The two latency-aware policies have similar overall performance profiles, although *rst-deg* consistently performs slightly better than *rst-cnt*.

In summary, the latency-aware policies decrease tail latencies significantly while maintaining a comparable overall performance to the policy that maximizes the number of commits. Moreover, a simple latency-aware policy that privileges the number of restarts over the graph degree is as effective as a more sophisticated policy that combines the graph degree and the number of restarts together.

5.5.2 Helping hard transactions commit

We simulate a heterogeneous workload by including 80% of normal transactions with 5 reads and 5 writes, and 20% of "hard transactions" with 10 reads and 10 writes. All the data accesses are drawn from the same Zipfian distribution. The larger transactions are more likely to conflict with others and less likely to commit; thus, we assign them higher priorities in the validator reordering. We compare the system performance with unweighted (*srvc*) validator reordering and a weighted (*srvs*) validator reordering policy that privileges the larger transactions. In both cases, storage batching is enabled. As before, the *baseline* configuration uses no batching.

Figures 20, 21, and 22 show the overall throughput, the abort rate, and the percentile latency. Since the ratio of large transactions is small, the overall performance under weighted and unweighted reordering policies is similar.

Figures 23, 24, and 25 show the throughput, the abort rate, and the percentile latency of the larger transactions alone. While the throughput of unweighted and weighted validator reordering is still similar, weighted validator reordering gives a much better transaction profile as far as the abort rate and the percentile latency are concerned.

5.6 Validator Clustering

In our final experiment, we explore the performance impact of clustering by access patterns for validator batch creation as described in Section 3.2. Since we do not introduce new clustering techniques in the paper, we hardcode a "clustering" strategy by partitioning the data into 2 equal partitions beforehand, and generating transactions based on the data partitions. Each transaction randomly chooses a partition as its home data partition. For each item it accesses, there is a 90% chance that the item is from its home partition, and a 10% chance that the item is from the other remote data partition. The transaction follows a Zipfian distribution when choosing items within a data partition.

We compare the system performance with validator clustering (*srvc-c*) and without validator clustering (*srvc*) when both storage and validator batching are enabled, as well as the baseline (*baseline*). Figure 26 shows the system throughput. Validator clustering can reduce throughput due to higher latencies, especially when the skew is high. On the other hand, as shown in Figure 27 and Figure 28, it gives a much better abort rate and percentile latency, since it reduces both inter- and intra-batch conflicts.

6. RELATED WORK

We discussed OCC and its applications in Sections 1; we also presented related work for the feedback vertex set problem in Section 4.1.2. Here, we discuss related work on concurrency control under high data contention, transaction scheduling, batching, and data clustering techniques.

High contention concurrency control. The performance of concurrency control protocols suffers when concurrency level and data contention are high [22]; this has particular impacts on OCC [3]. Hybrid approaches combine OCC and locking to limit the number of transaction restarts [54, 59]. The problem can also be addressed by adjusting the concurrency level adaptively, limiting the number of arriving transactions and/or using an exponential backoff for aborted transactions [31]. Transaction chopping [52] reduces contention [44, 58] by partitioning transactions into smaller pieces and executing dependent pieces in a chained manner. It is also possible to reduce conflicts by executing transactions at heterogeneous isolation levels [57, 58], using eventual consistency for non-critical transactions. While we also address the problem of reducing conflicts under data contention, our batching and reordering techniques are different from previous work.

Transaction scheduling. The dynamic timestamp assignment technique [5] assigns each transaction a timestamp interval and flexibly picks the commit timestamp from the interval. A similar technique can be used to optimize read-only transactions in distributed asynchronous OCC [15]. This approach can be extended to dynamically update the timestamp intervals of live transactions while committing a different transaction [9]. Dynamic timestamp assignment is compatible with our batching.

Also relevant is prior research on dynamic real-time transaction scheduling. Transactions in a real-time database system are associated with a soft or hard deadline, where completing after the deadline is either useless or undesirable. Thus, urgent or high value [29] transactions must be prioritized. OCC with forward validation [30, 40, 41] is a good technique in this scenario, since the validator can choose to abort or defer a transaction if it would cause live transactions with higher priority to abort. There are also systems that use locking and preemption [1], as well as hybrid optimistic/pessimistic methods [43, 32]. These approaches can be viewed as a simplified version of our validator reordering; moreover, none of the systems cited use batching.

Calvin [55] uses a deterministic locking-based approach, which fixes a partial order for transactions prior to execution by analyzing transaction dependencies at a centralized scheduler. ROCOCO [44] proposes a variation of Calvin where the scheduler can be distributed. Deterministic scheduling has also been extended to multi-version concurrency control [20]. These are complementary to our work. Our approach is more flexible as it allows reordering at multiple stages in a transaction's life cycle.

Conflict resolution for replicated systems may increase throughput by allowing replicas to diverge. Consistency can be achieved by reconciliation and merging [49]. Re-establishing consistency may not be necessary if transaction operations commute [42, 50], or the updates are not read by later transactions [42, 21]. Serializable snapshot isolation [33] achieves serializability by running transactions at snapshot isolation and aborting transactions that can potentially induce write skew. Warp [19] is a distributed OLTP system based on OCC. It organizes servers in a chain. At validation, it passes each transaction around the chain of servers that it accessed, builds the dependency graph and decides a serialization ordering along the way, and finally transmits the ordering back along the chain.

Our work has different goals and uses different techniques from all the above systems.

Batching. Batching to amortize costs and condense work is a common optimization technique. One major application is to pack networking and logging messages [10, 25, 24, 15]. Batching is also widely applied to application requests to improve performance, including group commits [28, 14], condensing IO requests [14, 21], and Paxos [51]. Since the use of batching is often associated with a throughput/latency tradeoff, there is work on adaptive batching [23, 45]. Those uses of batching are low-level and are not aware of the overall system infrastructure or the application semantics. Our work embraces batching as a core design principle at multiple stages of transaction execution. In addition, we focus on the use of batching for reordering, which is not explored in the work we cited.

Data clustering. As mentioned in Section 3.2, it is desirable to create validator batches that cluster transactions by access patterns. Clustering items that are likely to be accessed together in a transaction in an OLTP workload has been modeled as graph partitioning [56], which is NP complete; however, there are many sophisticated heuristics that can help [38, 37, 16]. Free and highly-optimized software libraries [36] are also available. This problem has gained recent attention due to its application in data-partitioned OLTP systems [47]. All the techniques mentioned can be incorporated into intelligent batch creation at the validator.

7. CONCLUSIONS AND FUTURE WORK

We have shown how to improve transaction performance in an OCC system by integrating batching into the system architecture at the storage and the validator levels. Batching allows the reordering of requests and reduces the number of conflicts and aborts. We have formulated validator reordering as the problem of finding the minimal unweighted (or weighted) feedback vertex set in a directed graph, and we have proposed two greedy algorithms for finding a FVS that are flexible and that perform well in practice. We have carried out an extensive experimental study of storage and validator batching in a main memory transaction processing prototype. Our experiments show that there is a sweet spot for the optimal batch size, which is consistent with the intuition that the batch size should strike a balance between latency and the flexibility of reordering. We have also investigated the impact of storage and validator batching on system performance. While both kinds of batching improve the performance significantly, our experiments revealed that storage batching is always beneficial, while validator batching is most useful in moderate contention settings where its benefits outweigh the cost of running the reordering algorithm. We show two examples of more complex policies for validator reordering that aim to reduce tail latencies and help hard transactions. We finally show that intelligent batch creation by access patterns at the validator gives better abort rate and latency.

In summary, batching improves OCC performance significantly in all settings. In future work, we plan to explore more sophisticated batch creation techniques, as well as investigating adaptive batching to intelligently enable batching and adjust batch sizes for best system performance.

8. REFERENCES

[1] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems (TODS)*, 17(3):513–560, 1992.

[2] A. Adya and B. Liskov. Lazy consistency using loosely synchronized clocks. In *PODC*, pages 73–82, 1997.

[3] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems (TODS)*, 12(4):609–654, 1987.

[4] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[5] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser. Dynamic timestamp allocation for transactions in database systems. In *DDB*, pages 9–20, 1982.

[6] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *SIGMOD*, pages 1295–1309, 2015.

[7] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - A transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[8] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *PVLDB*, 4(11):944–955, 2011.

[9] C. Boksenbaum, M. Cart, J. Ferrié, and J.-F. Pons. Concurrent certifications by intervals of timestamps in distributed database systems. *Software Engineering, IEEE Transactions on*, SE-13(4):409–419, 1987.

[10] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[11] J. Chen, Y. Liu, S. Lu, B. O'Sullivan, and I. Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. *Journal of the ACM (JACM)*, 55(5):21, 2008.

[12] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, pages 261–264, 2012.

[13] V. Cutello and F. Pappalardo. Targeting the minimum vertex set problem with an enhanced genetic algorithm improved with local search strategies. In *Intelligent Computing Theories and Methodologies*, pages 177–188. Springer, 2015.

[14] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment*, 6(14):1942–1953, 2013.

[15] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: elastic, high performance optimistic concurrency control by watermarking. In *SOCC*, pages 262–275, 2015.

[16] C. H. Ding, X. He, H. Zha, M. Gu, and H. D. Simon. A min-max cut algorithm for graph partitioning and data clustering. In *ICDM*, pages 107–114, 2001.

[17] I. Dinur and S. Safra. On the hardness of approximating minimum vertex cover. *Annals of mathematics*, pages 439–485, 2005.

[18] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *SIGMOD*, pages 299–313, 2015.

[19] R. Escriva, B. Wong, and E. G. Sirer. Warp: Lightweight multi-key transactions for key-value stores. Technical report, Cornell University, Ithaca, NY, USA, 2013.

[20] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11), 2015.

[21] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, pages 15–26, 2014.

[22] P. Franaszek and J. T. Robinson. Limitations of concurrency in transaction processing. *ACM Transactions on Database Systems (TODS)*, 10(1):1–28, 1985.

[23] R. Friedman and E. Hadad. Adaptive batching for replicated servers. In *SRDS*, pages 311–320, Oct 2006.

[24] R. Friedman and R. Van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *HPDC*, pages 233–242, 1997.

[25] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *SOSP*, pages 15–28, 2011.

[26] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *ACM SIGMOD Record*, volume 23, pages 243–252. ACM, 1994.

[27] V. Guruswami, R. Manokaran, and P. Raghavendra. Beating the random ordering is hard: Inapproximability of maximum acyclic subgraph. In *FOCS*, pages 573–582. IEEE, 2008.

[28] R. Hagmann. Reimplementing the cedar file system using logging and group commit. *SIGOPS Oper. Syst. Rev.*, 21(5):155–162, Nov. 1987.

[29] J. R. Haritsa, M. J. Canrey, and M. Livny. Value-based scheduling in real-time database systems. *The VLDB Journal*, 2(2):117–152, 1993.

[30] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *RTSS*, pages 94–103, 1990.

[31] A. Helal, T.-H. Ku, R. Elmasri, and S. Mukherjee. Adaptive transaction scheduling. In *CIKM*, pages 704–713, 1993.

[32] J. Huang, J. A. Stankovic, K. Ramamritham, and D. F. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *VLDB*, volume 91, pages 35–46, 1991.

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

[34] V. Kann. *On the approximability of NP-complete optimization problems.* PhD thesis, Royal Institute of Technology Stockholm, 1992.

[35] R. M. Karp. *Reducibility among combinatorial problems.* Springer, 1972.

[36] G. Karypis and V. Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, University of Minnesota, 1995.

[37] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[38] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[39] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[40] K.-W. Lam, K.-Y. Lam, and S.-L. Hung. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *RTSS*, pages 174–179, 1995.

[41] J. Lee and S. H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *RTSS*, pages 66–75, 1993.

[42] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, pages 265–278, 2012.

[43] Y. Lin and S. H. Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. In *RTSS*, pages 104–112, 1990.

[44] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, 2014.

[45] J. Nagle. Congestion control in IP/TCP internetworks. *SIGCOMM Comput. Commun. Rev.*, 14(4):11–17, Oct. 1984.

[46] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*, 5(11):1459–1470, 2012.

[47] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *SIGMOD*, pages 61–72, 2012.

[48] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, pages 251–264, 2010.

[49] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. *SIGOPS Oper. Syst. Rev.*, 31(5):288–301, Oct. 1997.

[50] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, pages 1311–1326, 2015.

[51] N. Santos and A. Schiper. Tuning paxos for high-throughput with batching and pipelining. In *Distributed Computing and Networking*, pages 153–167. Springer, 2012.

[52] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, 1995.

[53] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[54] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *TKDE*, 10(1):173–189, 1998.

[55] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[56] M. M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. *SIGMOD Rec.*, 20(2):12–21, Apr. 1991.

[57] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining ACID and BASE in a distributed database. In *OSDI*, volume 14, pages 495–509, 2014.

[58] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance ACID via modular concurrency control. In *SOSP*, pages 279–294, 2015.

[59] P. S. Yu and D. M. Dias. Analysis of hybrid concurrency control schemes for a high data contention environment. *Software Engineering, IEEE Transactions on*, 18(2):118–129, 1992.