SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads

Abdul Quamar
University of Maryland
abdul@cs.umd.edu

K. Ashwin Kumar
University of Maryland
ashwin@cs.umd.edu

Amol Deshpande
University of Maryland
amol@cs.umd.edu

ABSTRACT

In this paper, we address the problem of transparently scaling out transactional (OLTP) workloads on relational databases, to support *database-as-a-service* in cloud computing environment. The primary challenges in supporting such workloads include choosing how to *partition* the data across a large number of machines, minimizing the number of *distributed transactions*, providing high data *availability*, and tolerating *failures* gracefully. Capturing and modeling the transactional workload over a period of time, and then exploiting that information for data placement and replication has been shown to provide significant benefits in performance, both in terms of transaction latencies and overall throughput. However, such workload-aware data placement approaches can incur very high overheads, and further, may perform worse than naive approaches if the workload changes.

In this work, we propose SWORD, a scalable workload-aware data partitioning and placement approach for OLTP workloads, that incorporates a suite of novel techniques to significantly reduce the overheads incurred both during the initial placement, and during query execution at runtime. We model the workload as a hypergraph over the data items, and propose using a *hypergraph compression* technique to reduce the overheads of partitioning. To deal with workload changes, we propose an incremental data repartitioning technique that modifies data placement in small steps without resorting to complete workload repartitioning. We have built a workload-aware *active replication* mechanism in SWORD to increase availability and enable load balancing. We propose the use of *fine-grained quorums* defined at the level of *groups of tuples* to control the cost of distributed updates, improve throughput, and provide adaptability to different workloads. To our knowledge, SWORD is the first system that uses fine-grained quorums in this context. The results of our experimental evaluation on SWORD deployed on an Amazon EC2 cluster show that our techniques result in orders-of-magnitude reductions in the partitioning and bookkeeping overheads, and improve tolerance to failures and workload changes; we also show that choosing quorums based on the query access patterns enables us to better handle query workloads with different read and write access patterns.

EDBT '13, March 18 - 22 2013, Genoa, Italy.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases and Transaction processing*

General Terms

Algorithms, Design, Performance, Experimentation

Keywords

Database-as-a-service, Graph Partitioning, Data Placement, Replication, Quorums

1. INTRODUCTION

Horizontal partitioning (sharding) and replication are routinely used to store, query, and analyze very large datasets, and have become an integral part of any large-scale data management system. The commonly used techniques for sharding include hash-based partitioning [8], round robin partitioning, and range partitioning. A natural consequence of employing sharding and/or replication on transactional workloads is that transactions or queries may need to access data from multiple partitions. This is usually not a problem for analytical workloads where this is, in fact, desired and can be exploited to parallelize the query execution itself. However, to ensure transactional semantics, distributed *transactions* must employ a distributed consensus protocol (e.g., 2-phase commit or Paxos commit [11]), which can result in high and often unacceptable latencies [12]. During the last decade, this has led to the emergence and wide use of key-value stores that do not typically support transactional consistency, or often restrict their attention to simple single-item transactions.

Over the last few years, there has been an increasing realization that the functionality and guarantees offered by key-value stores are not sufficient in many cases, and there are many ongoing attempts to scale out OLTP workloads without compromising the ACID guarantees. H-Store [13] is an attempt to rethink OLTP query processing by using a distributed main memory database, but requires that the transactions be pre-defined in terms of stored procedures and not span multiple partitions. Google's Megastore [3] provides ACID guarantees within data partitions but limited consistency guarantees across them and has a poor write throughput. Moreover, the database features provided by Megastore are limited to the semantics that their partitioning scheme can support.

An alternate schema-independent approach, proposed by Curino et al. [7], is to observe and capture the query and transaction workload over a period of time, and utilize this workload information to achieve a data placement that minimizes the number of distributed transactions. Their approach, called *Schism*, models the transaction workload as a graph over the database tuples, where an edge indicates that the two tuples it connects appear together in a transac-

tion; it then uses a graph partitioning algorithm to find a data placement that minimizes the number of distributed transactions thus increasing throughput significantly over baseline approaches. However, there are several challenges in employing such a fine-grained workload-aware data placement approach: (a) the routing tables that store the tuple-to-partition mappings, required to dispatch the queries or transactions to appropriate partitions, can become very large and expensive to consult; (b) the initial cost of partitioning and the follow-on cost of maintaining the partitions can be very high, and in fact, it is unlikely that the fine-grained partitioning approach can scale to really large data volumes; (c) it is not clear how to handle newly inserted tuples, or tuples that do not appear in the workload; (d) the performance for such an approach can be worse than random partitioning if the workload changes significantly; and (e) random hash-based partitioning schemes often naturally have better load balancing and better tolerance to failures.

In this paper, we present SWORD, a *scalable workload-aware* data partitioning and placement approach for transparently scaling out standard OLTP workloads with full ACID support, to provide *database-as-a-service* in a distributed cloud environment. Like prior work in this area, we model the workload as a *hypergraph*[1], where each *hyperedge* corresponds to a transaction or a query[2], and employ hypergraph partitioning algorithms to guide data placement decisions. Our key contributions in this work are a suite of novel techniques to achieve higher scalability, and to increase tolerance to failures and to workload changes. First, to reduce book-keeping overhead, we propose using a *two-phase approach*, where we first *compress* the hypergraph using either hash partitioning or an analogous simple and easy-to-compute function, and then *partition* the compressed hypergraph. This results in a substantial reduction in the sizes of the mapping tables required for dispatching the transactions to appropriate partitions. As we show, this simple hybrid approach is able to reap most of the benefits of fine-grained partitioning at a much lower cost, resulting in significantly higher throughputs. Our approach also naturally handles both new tuples and tuples that were not accessed in the workload. Further, it is able to deal with changes in workload more gracefully and is more effective at tolerating failures.

Second, we propose an *incremental repartitioning* technique to deal with workload changes, that monitors the workload to identify significant changes, and repartitions the data in small steps to maintain a good overall partitioning. Our approach is based on efficiently identifying *candidate* sets of data items whose migration has the potential to reduce the frequency of distributed transactions the most, and then performing the migrations during periods of low load. Third, we propose using *fine-grained quorums* to alleviate the cost of distributed updates for active replication and to gracefully deal with partition failures. Unlike prior work [22, 10] where the types of quorum are chosen a priori and uniformly for all data items, we choose the type of quorum independently for groups of tuples based on their combined read/write access patterns. This allows us to cater to typical OLTP workloads that have a mix of read and write queries with varying access patterns for different data items.

Fourth, we propose an aggressive replication mechanism that attempts to *disentangle* conflicting transactions through replication, enabling better data placement. Finally, we develop an efficient *query routing* mechanism to identify which partitions to involve in a given transaction. Use of aggressive replication and quorums makes this very challenging, and in fact, the problem of identifying a minimal set of partitions for a given query is a generalization of the *set cover* problem (which is not only NP-Hard but also hard to approximate). We develop a greedy heuristic to solve this problem. We also develop a compact routing mechanism that minimizes memory overheads and improves lookup efficiency.

Our experimental evaluation of SWORD deployed on an Amazon EC2 cluster demonstrates that our hypergraph-based workload representation and use of in-graph replication based on access patterns, lead to a much better quality data placement as compared to other data placement techniques. We show that our scaling techniques result in orders-of-magnitude reductions in the partitioning overheads including the workload partitioning time, cost of distributed transactions, and query routing times for data sets consisting of up to a billion tuples. Our incremental repartitioning technique effectively deals with the performance degradation caused by workload changes using minimal data movement. We also show that our techniques provide graceful tolerance to partition failures compared to other data placement techniques.

Summarizing, the major contributions of our work are:

- *Effective workload modeling and compression* that reduces partitioning and book-keeping overheads, and enables handling of both new tuples and those not represented in the workload.
- *Incremental repartitioning* to mitigate performance degradation due to workload changes without a complete repartitioning.
- *Use of fine-grained quorums* to control the cost of distributed updates, to improve throughput, and to cater to OLTP workloads with a mix of different access patterns.
- *Workload-aware replication* mechanism that attempts to *disentangle* conflicting transactions leading to better data placement.
- *Efficient and scalable routing mechanism* that minimizes the number of partitions to involve for a given query and uses compact routing tables to minimize memory requirements.

The remainder of the paper is organized as follows. Section 2 gives a background on workload modeling, replication, and provides an overview of SWORD's architecture. Section 3 provides the details of our proposed solution for scalable workload-aware data placement. Section 4 discusses the experimental set-up and evaluation of SWORD, and Section 5 explores the related work in the area.

2. OVERVIEW

We begin with providing a high-level overview of SWORD's architecture. We then briefly present the basic workload-aware approach that captures a transaction workload as a hypergraph, and utilizes that workload information to achieve a data placement that minimizes the number of distributed transactions [7].

2.1 System Architecture

The key components of SWORD are shown in Figure 1, and can be functionally divided into three groups: data partitioning and placement, incremental repartitioning, and transaction processing. The data itself is horizontally partitioned (sharded) across a collection of physical database partitions, i.e., machines that run a relational resource manager such as a relational DBMS server (PostgreSQL in our implementation). Data may be replicated (at the granularity of tuples) to improve availability, performance, and fault tolerance. We assume that each tuple is associated with a globally unique **primary key**, which may consist of more than one at-

[1]We chose a hypergraph-based representation because we observed better performance, but we could also use a graph-based representation instead.

[2]Hereafter we use the term *transaction* to denote both an update transaction or a read-only query.

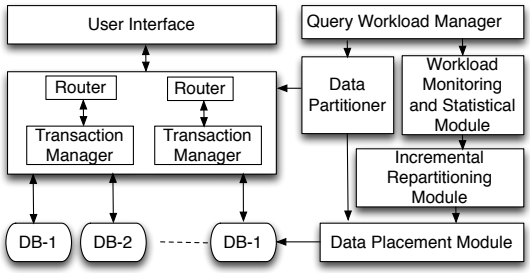


Figure 1: System architecture

tribute. We briefly discuss the key functionality of the different components next.

Data partitioning and placement: These modules are in charge of making the initial workload-aware data placement and replication decisions, and then carrying out those decisions through appropriate data migration and replication. The query workload manager takes the query workload trace (the set of transactions and the tuples they access (Section 4.1)) as input and generates a *compressed* hypergraph representation (Section 3.1) of the query workload. The compressed hypergraph is then fed to the data partitioner which does in-graph replication (Section 3.3), and partitions the resulting hypergraph using the hMetis [1] partitioning tool. The output of the partitioner is a mapping of the tuples to their physical database partitions. These mappings are fed to the data placement sub-module and the router. The data placement sub-module then uses these mappings to partition the database across the machines.

Incremental repartitioning: The workload monitoring and statistical module monitors the workload changes and maintains statistics on the workload access patterns. It provides this input to the incremental repartitioning module (Section 3.2) which identifies when the current partitioning is sub-optimal and triggers data migration to deal with workload changes. The data migration is done in incremental steps through the data placement module during periods of low activity.

Transaction processing: The users submit transactions, and receive their results through an interface provided by the transaction processing module. The user interface sends the transactions to the router which parses the SQL statements in the transactions using an SQL parser that we wrote. The router determines the tuples accessed by the transaction (more specifically, the primary keys of the tuples accessed by the transaction), their replicas, and their location information using the mappings provided by the data partitioner. The router also determines the appropriate number of replicas that need to be accessed for each tuple to satisfy the quorum requirements (Section 3.4). The router then uses a *set-cover based* algorithm to compute the minimum number of partitions that the transaction needs to be executed on (referred to as **query span** in the rest of the paper), to access all the required tuples and replicas (Section 3.5). This information, along with the transaction, is passed on to the transaction manager which executes the transactions in parallel on the required database partitions. The transaction manager uses a *2-phase commit protocol* to provide the ACID guarantees.

2.2 Workload Modeling

We represent the query workload as a hypergraph, $\mathcal{H} = (V, E)$, where each hyperedge $e \in E$ represents a transaction, and the set of nodes $V_e \subseteq V$ spanned by the hyperedge represent the tuples accessed by the transaction. Each hyperedge is associated with an edge weight w_e which represents the frequency of such transactions in the workload.

A *k-way balanced min-cut partitioning* of this hypergraph would give us k balanced partitions of the database (i.e., k partitions of equal size) such that the number of transactions spanning multiple partitions is minimized. This is because every transaction that spans multiple partitions corresponds to a hyperedge that was cut in the partitioning. Instead of looking for partitions of equal size, we may instead assign a *weight* to each node and ask for a partitioning such that the total weights of the partitions are identical (or almost identical). The weights may correspond to the item sizes (in case of heterogeneous data items) or some combination of the item sizes and access frequencies. The latter may be used to achieve balanced loads across the partitions.

The problem of k-way balanced min-cut partitioning generalizes the *graph bisection* problem, and is NP-hard. However, due to the practical importance of this problem, many efficient and effective hypergraph partitioning packages have been developed over the years. We use the hMetis package [1] in our implementation.

Figure 2(a) shows an illustrative example where a transactional query workload is transformed into a hypergraph. The hypergraph consists of a vertex set $V = \{a, b, c, e, f, g\}$ and hyperedge set $E = \{e_1 = \{a, b, c\}, e_2 = \{a, g\}, e_3 = \{g, c\}, e_4 = \{a, e\}, e_5 = \{f, c\}\}$ where e_i represent the transactions. A 2-way min-cut partitioning of this hypergraph gives us 2 distributed transactions, as compared to a naive round-robin partitioning that would have given us 4 distributed transactions.

3. SYSTEM DESIGN

In this section, we first present our proposed techniques for scalable workload-aware data partitioning, and for incremental repartitioning to cater to workload variations. We then discuss our in-graph replication mechanism, and use of fine-grained quorums to improve availability. Finally, we present our query routing mechanism to select partitions to involve in a given query or a transaction.

3.1 Hypergraph Compression for Scaling

The major scalability issues involved with workload-aware hypergraph (or graph) partitioning-based techniques are: (1) the memory and computational requirements of hypergraph storage and processing, which directly impact the partitioning and repartitioning costs, and (2) the large size of the tuple-to-partition mapping produced by the partitioner that needs to be stored at the router for routing the queries to appropriate partitions, that makes the router itself a bottleneck in query processing. Existing hypergraph compression techniques [2] based on *coalescing* help in effectively reducing the size of the hypergraph, and in some cases [26] even minimize the loss of structural information by using additional neighborhood information as input to the coalescing function. However these techniques do not reduce the sizes of mapping tables required for routing queries, and thus are not appropriate for our context.

We propose using a simple two-step **hypergraph compression** technique instead. We first group the nodes of the hypergraph (i.e., database tuples) into a large number of groups using an easy-to-compute function applied to the primary keys of the tuples, and we then collapse each group of nodes into a single *virtual node*. More specifically, in the first step, we map each node $v \in V$ in the original hypergraph to a *virtual node* $v' \in V'$ in the compressed hypergraph by computing $v' = f(pk_v)$, where pk_v represents the node v's primary key. In our current implementation, we use a hash

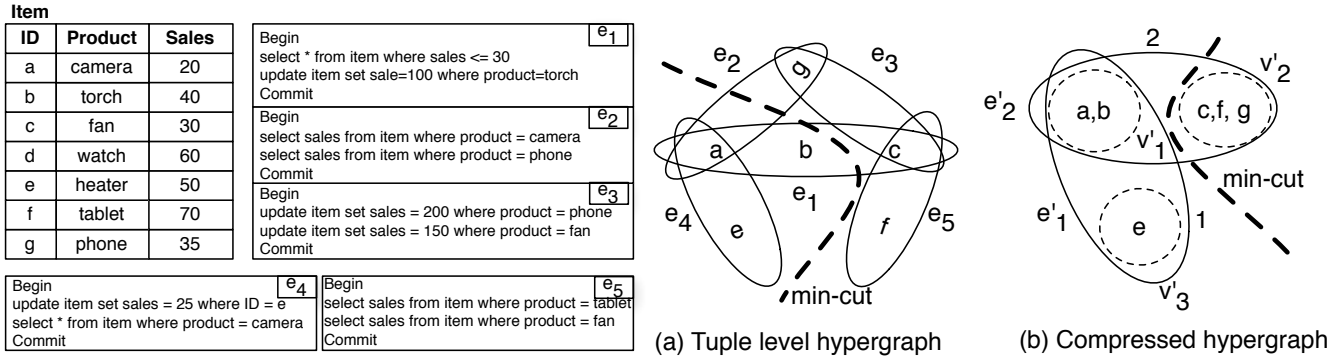


Figure 2: (a) Workload representation using a hypergraph; (b) Hypergraph compression.

function, $HF(pk_v) = hash(pk_v) \bmod N$, where N is the desired number of virtual nodes. However, any inexpensive function (e.g., a range partitioner) could be used instead. Using such a primary key-based coalescing plays a crucial role in developing an efficient and scalable routing mechanism with minimum book-keeping; further details are discussed in Section 3.5.

Let V' denote the resulting set of virtual nodes. For a hyperedge $e \subseteq V$ in the original hypergraph, let $e' \subseteq V'$ denote the set of virtual nodes to which the vertices in e were mapped. If e' contains at least two virtual nodes, then we add e' as a hyperedge to the compressed graph (denoted $\mathcal{H}' = (V', E')$). We define the hypergraph compression ratio (CR) as the ratio of the number of nodes $|V|$ in the original hypergraph to the number of virtual nodes $|V'|$ in the compressed hypergraph, i.e., $CR = \frac{|V|}{|V'|}$. $CR = 1$ indicates no compression, whereas $CR = |V|$ indicates that all the original vertices were mapped onto a single virtual node.

Next, we iterate over each hyperedge $e \in E$ of the original hypergraph and replace each node $v_e \in V_e$ spanned by the hyperedge e with $v'_{e'}$ using the mapping generated in the first step. Each hyperedge $e' \in E'$ so generated in the compressed hypergraph is associated with an edge weight $w_{e'}$ which represents the frequency of the hyperedge. Figure 2(b) provides an illustration of compressed hypergraph generation. The mappings produced by the first step create virtual nodes $v'_1 = \{a, b\}$, $v'_2 = \{c, f, g\}$ and $v'_3 = \{e\}$. The second step generates the hyperedges e'_1 and e'_2 and the edge weights associated with these hyperedges depict the frequency of transactions accessing the corresponding sets of virtual nodes.

This hybrid coarse-grained approach, although simple, is highly effective at reaping the benefits of workload-aware partitioning without incurring the high overhead of the fine-grained approach. First, the hypergraph size is reduced significantly, reducing the overhead of running the partitioning and repartitioning algorithms. Second, it naturally handles new inserted tuples and tuples that were not part of the provided workload. Each such tuple is assigned to a virtual node based on its primary key and placed on the partition assigned to the virtual node. Third, it avoids *over-fitting* the partitioning and replication to the provided workload, resulting in more robust data placement. We also need significantly smaller query workloads as input to make partitioning decisions.

On the other hand, the coarseness introduced by the compression process may result in larger min-cuts (and thus higher number of distributed transactions). However, we empirically show in Section 4 that the orders-of-magnitude gains in terms of the above mentioned benefits far offset the probable increased cost of distributed transactions as compared to a fine-grained approach.

3.2 Incremental Repartitioning

A workload-driven approach is susceptible to performance degradation if the actual workload (in the future) is significantly different from the workload used to make the partitioning and replication decisions. The quantum of performance variance is dependent on the sensitivity of the data placement technique to workload change. As we illustrate through our experimental evaluation (Section 4.4), the coarser representation achieved through hypergraph compression makes our approach less sensitive to workload changes compared to the fine-grained approach. However, significant workload changes will result in the initial placement being sub-optimal over time. In this section, we present an incremental repartitioning technique that performs data migration in incremental steps without resorting to complete repartitioning.

Our proposed incremental repartitioning technique monitors the workload changes at regular intervals, and moves a fixed amount of data items across partitions in incremental steps to mitigate the impact of workload change. The data migration is triggered whenever the percentage increase in the number of distributed transactions ($\triangle_{mincut}$) crosses a certain *threshold*, c, a system parameter which can be set as a percentage of the initial min-cut, depending upon the sensitivity of applications to latency.

At a high level, our algorithm maintains pairs of sets of candidate virtual nodes that can be *swapped* to reduce the size of the min-cut. During lean periods of activity, the algorithm makes a maximum of k such moves in each step to reduce the min-cut of the data placement as per the current workload. It repeats these steps until the min-cut reduces below the threshold value. The algorithm thus provides an incremental approach to adjust the data placement without resorting to complete data migration.

More specifically, let $\mathcal{H}_{cut} = \{e_1, e_2, \ldots, e_t\}$ denote the set of hyperedges that span multiple partitions, i.e., the set of hyperedges in the cut, as per the initial data placement. Let $\mathcal{P}_{cut} = \{P_1, P_2, \ldots, P_t\}$ be the set of *partition sets*, where $P_i \in \mathcal{P}_{cut}$ is the set of partitions spanned by hyperedge $e_i \in \mathcal{H}_{cut}$. Further, let $V_i = \{v_1, v_2, \ldots, v_n\}$ be the set of virtual nodes covered by hyperedge e_i, and let $\mathcal{V}_{cut} = \{\bigcup_{i=1,\ldots,t} V_i\}$, be the union set of nodes covered by all the hyperedges in the cut. This is the first set of our *candidate nodes* for migration. For each virtual node $v_i \in \mathcal{V}_{cut}$, in our first candidate set, we maintain a set of partitions $\mathcal{P}_{v_i}$ that contain the node or its replicas, such that $\{v_i \in p_j, \forall p_j \in \mathcal{P}_{v_i}\}$. Let nh_{ij} be the sum of the weights of hyperedges incident on node v_i in partition p_j. So each vertex v_i is associated with a set $\mathcal{NH}_i$ where $nh_{ij} \in \mathcal{NH}_i$ and $p_j \in \mathcal{P}_{v_i}$.

Let $\mathcal{VS}$ be the set of virtual nodes that are covered only by hyperedges that are not cut. In other words, if we move a virtual node in

$\mathcal{VS}$ to a different partition, it would not change the min-cut value. This set of virtual nodes forms our second set of candidate nodes.

Let the contribution of each hyperedge $e \in \mathcal{H}_{cut}$ towards total number of distributed transactions seen so far be

$$C_e = \frac{ndt_e}{\sum_{i=1,\ldots,t} ndt_{e_i}}$$

and let $\mathcal{C} = \{C_e | e \in H_{cut}\}$. The numerator ndt_e is the weight of the hyperedge e in the cut, whereas the denominator is the sum of the weights of the hyperedges in the cut. We maintain a priority queue, $\mathcal{PQ}$ of hyperedges $e \in \mathcal{H}_{cut}$. Each element in $\mathcal{PQ}$ is ordered by C_e and thus the largest element represents the hyperedge with the highest value of C_e. We choose to consider only those hyperedges that span two partitions which guarantees that a single swap of virtual nodes between two partitions would reduce the min-cut[3].

Swapping gain ($\mathcal{SG}$): Consider a hyperedge $e_i \in \mathcal{H}_{cut}$ spanning two partitions $p_a \in P_i$ and $p_b \in P_i$ where $P_i \in \mathcal{P}_{cut}$. Let $S_a = \{p_a \bigcap V_i\}$, $S_b = \{p_b \bigcap V_i\}$, be the set of virtual nodes covered by e_i in the partitions p_a and p_b respectively. Let $\bar{S}_a = \{\{p_a - V_i\} \bigcap \mathcal{VS}\}$, $\bar{S}_b = \{\{p_b - V_i\} \bigcap \mathcal{VS}\}$. The swapping of all the virtual nodes in S_b with a set of virtual nodes $\{I \subseteq \bar{S}_a \mid I_w \simeq S_{b_w}\}$ where I_w and S_{b_w} is the sum of node weights in I and S_b respectively (to maintain a load balance), would result in two things. Firstly e_i would be removed from $\mathcal{H}_{cut}$ decreasing the min-cut by ndt_e. Secondly, the set of hyperedges other than e_i which are incident on the nodes in S_b might probably become distributed increasing the min-cut by $(\sum_{i \in S_b} nh_{ib} - ndt_e)$ in the worst case. Thus the minimum swapping gain $\mathcal{SG}$ is given by:

$$\mathcal{SG} = ndt_e - (\sum_{i \in S_b} nh_{ib} - ndt_e) = 2 \times ndt_e - \sum_{i \in S_b} nh_{ib}$$

Algorithm 1 Incremental repartitioning algorithm

Require: Initial min-cut M_c, $\mathcal{PQ}$, threshold c, $CN = \emptyset$.
1: **while** $\triangle_{mincut} > c\%$ of M_c **do**
2: **while** $|CN| < k$ **do**
3: $e = \mathcal{PQ}.peek()$
4: $\mathcal{SG}_1 = 2 \times ndt_e - \sum_{i \in S_a} nh_{ia}$
5: $\mathcal{SG}_2 = 2 \times ndt_e - \sum_{i \in S_b} nh_{ib}$
6: **if** $\mathcal{SG}_1 \geq \mathcal{SG}_2$ and $\mathcal{SG}_1 > 0$ **then**
7: Identify $\{I \subseteq S_b \mid I_w \simeq S_{a_w}\}$
8: $CN = CN \leftarrow (I, S_a)$
9: $\mathcal{PQ}.remove(e)$
10: **else if** $\mathcal{SG}_1 \geq \mathcal{SG}_2$ and $\mathcal{SG}_1 > 0$ **then**
11: Identify $\{I \subseteq S_a \mid I_w \simeq S_{b_w}\}$
12: $CN = CN \leftarrow (I, S_b)$
13: $\mathcal{PQ}.remove(e)$
14: **end if**
15: **end while**
16: Swap the k sets of virtual nodes
17: Update $\mathcal{H}_{cut}$, $\triangle_{mincut}$, $\mathcal{PQ}$
18: **end while**

Algorithm 1 provides the details of our proposed incremental partitioning technique. A background process monitors the workload and populates $\mathcal{PQ}$. The algorithm is triggered when $\triangle_{mincut}$ exceeds a given threshold value c. The algorithm (lines 2-9) identifies at most k pairs of sets of virtual nodes for swapping which would maximize the total SG and stores them in $\mathcal{CN}$ as candidates to be swapped. It executes the k swaps (line 10) at a lean period of activity. It then updates $\mathcal{H}_{cut}$, $\triangle_{mincut}$ to reflect the changes caused by the swaps. It repeats the steps until the current min-cut falls below the set threshold value.

[3]A majority of hyperedges in the cut of our compressed hypergraph representing TPC-C, a typical OLTP workload, span two partitions.

3.3 Workload-aware Replication

Active and aggressive replication has the potential to provide better load balancing, improved availability in presence of failures, and a reduction in the number of distributed *read* transactions. However, providing strict transactional semantics with ACID properties becomes a challenge in presence of active replication [10].

We propose an aggressive workload-aware replication technique that provides data availability proportional to the workload requirement. We exploit tuple-level access pattern statistics to ascertain the number of replicas for each data item. We argue that the drawbacks of replicating items that are heavily updated are offset by several considerations: (1) for availability, it is desired that each data item be replicated at least once; (2) items that are heavily updated are typically also heavily read and replicating those items can reduce the total number of read-only distributed transactions; (3) through use of appropriate *quorums*, we can balance the writes across a larger number of partitions. A key feature of our replication technique is the notion of disentangling transactions to afford better min-cuts. We discuss this further below.

Replica generation: We have developed a *statistical module* that uses the transactional logs (Section 4.1) to compute the read and write statistics for each virtual node. Each node v'_i in the compressed hypergraph $\mathcal{H}$ represents a set of tuples T_i. Each tuple $t_{ij} \in T_i$ has a read frequency (fr_{ij}) and a write frequency(fw_{ij}). To compute each node's replication factor we compute the size compensated average of reads and writes per virtual node as follows:

$$Avg(v'_i)_w = \frac{\sum_j fw_{ij}}{logS(v'_i)},\ Avg(v'_i)_r = \frac{\sum_j fr_{ij}}{logS(v'_i)},\ \mathcal{R} = \frac{Avg(v'_i)_w}{Avg(v'_i)_r}$$

where $Avg(v'_i)_w$ and $Avg(v'_i)_r$ are average read and write frequencies of node v'_i, $logS(v'_i)$ is the log of the size of node v'_i. $\mathcal{R}$ is the average write-to-read ratio.

Based on the access pattern statistics generated, if the ratio $\dot{\mathcal{R}} \geq \delta$ (where $0 < \delta < 1$) the virtual node is replicated only once to control the cost of distributed updates. For all other nodes, the number of replicas generated is a linear function of $Avg_r(i)$. δ serves as a threshold value for controlling the number of replicas for heavily written tuples and can be chosen based on the workload requirements and the level of fault tolerance required. We use the log of the sizes of virtual nodes to compensate for the *skew* in the size of the virtual nodes; this helps in limiting the number of replicas created for heavily accessed large virtual nodes.

In-graph replication: Once we have chosen the number of replicas for a virtual node, we modify the compressed hypergraph by adding as many copies of the virtual node as required. One key issue then is assigning these virtual node replicas to the hyperedges in the graph. We observe that by doing this cleverly, we can disentangle some of the transactions that share data items and construct a graph with a better min-cut. Let $R_{v'}$ be the set of replicas for the virtual node v'. The replica assignment algorithm computes a set of distinct hyperedges $E_{v'}$ incident on v' and for each $e' \in E_{v'}$ its associated edge weight $w_{e'}$. There are two possibilities:

Case 1: $|E_{v'}| \geq |R_{v'}|$: We reduce this case to a simple *multiprocessor scheduling* problem. Each replica $r \in R_{v'}$ is associated with a processor b_r. Each hyperedge $e' \in E_{v'}$ is assigned to one

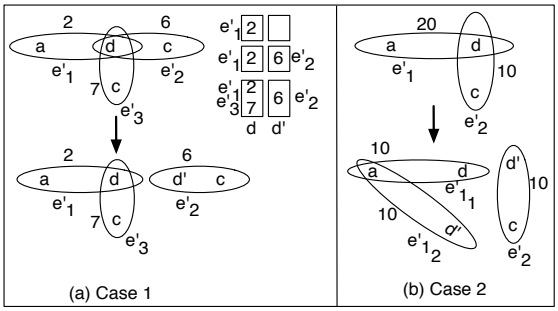


Figure 3: In-graph replication

of these processors, increasing the current load on the processor by the corresponding hyperedge weight $w_{e'}$. Minimizing the maximum load across the processors is equivalent to finding an equitable assignment of the replicas to the hyperedges. Since the scheduling problem is NP-Hard, we use a greedy approach which considers the hyperedges in the decreasing order by weight, and assigns the next hyperedge to the processor with the current minimum load. Finally all the hyperedges assigned to a particular processor b_r are allocated the replica r associated with the processor.

Figure 3(a) gives an example showing the assignment of a virtual node d and its replica d' to the incident hyperedges e'_1, e'_2 and e'_3 with edge weights 2, 6 and 7 respectively. The algorithm creates two processors representing d and its replica $d^{'}$ and assigns hyperedges to these processors greedily.

Case 2: $|E_{v'}| < |R_{v'}|$: The insertion algorithm splits the hyperedge e' with the highest weight $w_{e'_{max}}$ into two hyperedges with weights $\lceil \frac{w_{e'_{max}}}{2} \rceil$, $\lfloor \frac{w_{e'_{max}}}{2} \rfloor$ respectively. It repeats this procedure until the number of hyperedges is equal to $|R_{v'}|$ and then allocates one replica to each hyperedge.

Figure 3(b) gives an example for case 2 showing the assignment of a virtual node d and its replicas d', d'' to the incident hyperedges e'_1 and e'_2 with edge weights $w_{e'_1} = 20$ and $w_{e'_1} = 10$ respectively. The algorithm chooses e'_1 since it has the highest weight and splits it into two hyperedges e'_{1_1} and e'_{1_2} each with a weight of 10.

3.4 Fine-grained Quorums

Aggressive active replication comes at the cost of distributed update transactions which hurt performance. *Quorums* [23] have been extensively used to control the overheads associated with distributed updates for maintaining active replica consistency [22, 10]. In addition to this, quorums also help in improving fault tolerance by gracefully dealing with partition failures.

Let $S = \{S_1, S_2, \cdots, \}$ denote the set of partitions on which a data item is stored. A quorum system Q (for that data item) is defined to be a set of subsets of S with pair-wise non-empty intersections [24]. Each element of Q is called a quorum. A simple example of a quorum system is the *Majority quorum*, where every majority of the partitions forms a quorum. Defining read and write quorums separately, a quorum system is valid if: (a) every read quorum (rq) overlaps with every write quorum (wq), and (b) every two write quorums have an overlap. Another quorum system is ROWA (read-one-write-all), where a read can go to any of the partitions, but a write must go to all the partitions. Quorums allow us to systematically reduce the number of partitions that must be involved in a query, without compromising correctness.

Depending on the nature of the workload, the choice of the quorum system plays a significant role in determining its effectiveness in improving performance. For example, ROWA quorum would perform well for read intensive workloads and Majority quorum would help in controlling the cost of distributed updates for write intensive workloads. However different transactional workloads might have different mixes of read and write queries. Also, different data items in a given workload may have different read-write access patterns. Choosing a *fixed quorum for the all the data items* in the system a priori may significantly hurt the performance.

In this paper, we propose using *fine-grained quorums*, which are defined at the virtual node level (a group of tuples). We focus on two quorum systems, ROWA and Majority. Given a workload, the type of quorum for each virtual node is decided based on its read/write access pattern, as monitored by the statistical module. We compute $\mathcal{R}$, the write-to-read ratio (Section 3.3) for each virtual node. The quorum for each virtual node is then decided based on the value of $\mathcal{R}$. If $\mathcal{R} > \gamma$, where $(0 < \gamma < 1)$, then we chooses Majority quorum else ROWA quorum. The value of γ is a system parameter, which can be adjusted based on the nature of the query workload. We experimented with different values for γ and observed that as γ increases from 0.5 and tends towards 1, the system chooses ROWA for most data items incurring a high penalty for writes thereby reducing performance. On the other hand, as γ decreases from 0.5 and tends towards 0, the system chooses Majority quorum for most data items incurring a higher overhead for reads. Our experiments showed that $\gamma = 0.5$ was able to achieve a fine balance between the benefits of ROWA quorum for reads and Majority quorum for reducing the number of copies to be updated and gave the best performance for the TPC-C benchmark.

Quorums defined at the virtual node level specify the number of copies of each data item that need to be accessed in order to meet the quorum requirement. For each virtual node v' having a set of available copies $\mathcal{C}_{v'}$, a read quorum $|c_r|$, $c_r \subset \mathcal{C}_{v'}$ and a write quorum $|c_w|$, $c_w \subset \mathcal{C}_{v'}$ defines the number of copies of v' required for either a read or write query. These read and write quorums values are defined based on the types of quorum. For example, a majority quorum requires that $|c_r| + |c_w| > |C_{v'}|$ and $2 * |c_w| > |C_{v'}|$, while ROWA requires $|c_r| = 1$ and $|c_w| = |C_{v'}|$.

The choice of quorum at the level of each virtual node makes the system adaptive to a given workload and improves the effectiveness of quorums in reducing the costs of distributed updates significantly. We have conducted extensive experiments to study the use of different quorums for a number of query workloads with different mixes of read and writes. Our results show that fine-grained quorums provide significant benefits in terms of reducing the average query span and improving the transaction throughput for different types of workloads. This feature is especially useful for database-as-a-service in a cloud computing environment.

3.5 Query Routing

The use of graph based partitioning and replication schemes requires that the mappings of tuples to partitions be stored at the router to direct transactions to appropriate partitions. This is a major scalability challenge since the size of these mappings can become very large, and they may not fit fully in the main memory leading to increased lookup times. The problem is further aggravated with tuple-level replication which only adds to the size of these mappings. Existing techniques for dealing with this issue [25] use compute-intensive look-up table compression techniques coupled with a scaled-up router architecture to fit the lookup tables in memory, which may not be cost effective.

We propose a routing mechanism that requires minimum bookkeeping as a natural consequence of our hypergraph compression

technique. The size of the mapping tables is reduced by a factor of CR (the hypergraph compression ratio). Depending on the router's compute and memory capacity, a suitable CR could be chosen to optimize overall performance. In addition to this, we incorporate two additional features to reduce the query span and the cost of distributed updates: fine-grained quorums (as described in Section 3.4) that determine the number of copies of each data item required, and a set-cover algorithm that determines the minimum number of partitions required to satisfy the query and meet the quorum requirements.

Minimum set-cover algorithm: The minimum set-cover problem to minimize the query span can be defined as follows: given a transaction e', a set of virtual nodes $V'_{e'}$ accessed by e' and their replicas $R_{e'}$; a set of partitions $\{P^{e'}_{RV'} \mid V'_{e'} \cup R_{e'} \subseteq P^{e'}_{RV'}\}$; a universe $\mathcal{U}_{e'} = \{v' \rightarrow c \mid v' \in V'_{e'}, c \in C_{e'}\}$ where c is the number of copies required for v' as per the quorum requirement; a set-cover map $\mathcal{S}_{e'} = \{v' \rightarrow c \mid v' \in V'_{e'}, c \in C_{e'}\}$ where the initial count c of each element is set to 0; determine the minimum number of partitions $S \subseteq P^{e'}_{RV'}$ that cover the universe $\mathcal{U}_{e'}$. The minimum set-cover is an NP-Complete problem and we use a greedy heuristic to solve the same. In each iteration the algorithm determines the partition P_i which covers the maximum uncovered elements $UC_{e'}$ in the universe $\mathcal{U}_{e'}$ given by $max(\{\mathcal{U}_{e'} - S_{e'}\} \cap P_i)$, where $\{\mathcal{U}_{e'} - S_{e'}\}$ denotes the operation wherein the counts of the elements in the universe $\mathcal{U}_{e'}$ are decremented by the count of the corresponding elements in $S_{e'}$. The set-cover S is updated with the partition P_i, i.e., $S = S \cup P_i$, $S_{e'} = S_{e'} + P_i$ which increases the count of common elements in the set-cover map by one. The uncovered elements are updated by $UC_{e'} = \mathcal{U}_{e'} - S_{e'}$ which reduces the counts of common elements in $\mathcal{U}_{e'}$ by the counts of the corresponding elements in $S_{e'}$. The algorithm terminates when the counts of all elements in $UC_{e'} = 0$ and outputs S. The algorithm for computing the set-cover is shown in Algorithm 2.

To give an example: consider a transaction $e' = \{2, 3, 5, 9, 12, 14\}$ where the numbers in the set indicate the IDs of the virtual nodes accessed by e', $C_{e'} = \{2, 1, 1, 2, 1, 1\}$ which denotes the number of copies required for corresponding elements in e' to satisfy the quorum requirements. Consider a set of partitions $P_1 = \{2, 9\}$, $P_2 = \{2, 3^c, 5^c, 14\}$, $P_3 = \{9^c, 5^c, 12\}$, and $P_4 = \{12^c, 14^c\}$ where the element $n \in V'_{e'}$ and $n^c \in R_{V'_{e'}}$. In the first iteration the algorithm chooses P_2, updates $S = \{P_2\}$, then computes the uncovered elements $UC_{e'} = e' - P_2$ updating $C_{e'} = \{1, 0, 0, 2, 1, 0\}$. In the second iteration it chooses P_3, updates $S = \{P_2, P_3\}$ and $C_{e'} = \{1, 0, 0, 1, 0, 0\}$. In the third iteration the algorithm chooses P_1, updates $S = \{P_2, P_3, P_1\}$ and $C_{e'} = \{0, 0, 0, 0, 0, 0\}$ and the algorithm terminates as all elements in the universe are covered. S constitutes the minimum number of partitions that the transaction needs to be routed to.

Algorithm 2 Set-cover Algorithm

Require: $\mathcal{H}', e' \in E', P_i \in P_{RV}, \mathcal{U}_{e'} = \{v' \rightarrow c \mid v' \in V_{e'}, c \in C_{e'}\}, S_{e'} = \{v' \rightarrow c \mid v' \in V_{e'}, c \in C_{e'}\}$

```
while UC_e' ≠ 0 do
    p_index = argmax_i({U_e' − S_e'} ∩ P_i)
    S ∪= P_{p_index}
    S_e' += P_{p_index}
    UC_e' = U_e' − S_e'
end while
return S
```

Generation of mapping tables: We use a hash table-based lookup mechanism to determine the virtual nodes accessed by a query. We create a set of hashmaps for each relation $\mathcal{RE}_i$ in the partitioned database. Using the hash function HF_{pk_v} used for graph compression to map tuples to virtual nodes, for each relation $\mathcal{RE}_i$, we create one hashmap $\mathcal{M}_k$ per primary key attribute $k \in \mathcal{K}_i$ where $\mathcal{K}_i$ is the set of attributes which form the primary key for relation $\mathcal{RE}_i$. These hash tables map the distinct values of $k \in \mathcal{K}_i$ to a set of virtual nodes that contain tuples with corresponding values of k. Further, we create union maps $\mathcal{M}_i = \cup\mathcal{M}_k, \forall k \in \mathcal{K}_i$ which essentially contain all virtual nodes containing tuples of relation $\mathcal{RE}_i$. These mapping tables generated at the virtual node level need to be updated or regenerated only at the time of data repartitioning, a relatively infrequent process for stable workloads. New tuples in the database are automatically mapped to existing virtual nodes and hence do not require any updates to the mapping tables at the router.

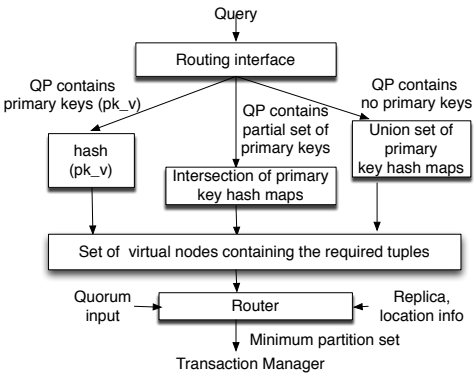


Figure 4: Routing architecture

We see a drastic reduction in memory requirement compared to the fine-grained scheme. This can be attributed to two factors. First, the tables are maintained at the level of virtual nodes and hence provide a reduction in size by a factor of CR. Second, we maintain hash maps per primary key attribute. The number of distinct values per primary key attribute is much smaller than the total number of distinct primary key values[4], making the hash tables very compact. Table 1 shows the effectiveness of our proposed routing mechanism by comparing the size of the router mapping tables for a workload of 1 Billion tuples.

Routing mechanism: Figure 4 illustrates the flow of our routing mechanism. The routing interface provided by the query processing module takes a query as input and parses it to determine the relation $\mathcal{R}_i$ and the set of primary keys $\mathcal{QP}_i$ in the query predicate. It then deals with three cases. First, if $\mathcal{QP}_i = \mathcal{K}_i$, it simply hashes the key values and obtains the virtual node which the query needs to access. This process requires no lookup and is the most efficient case. Second, if $\mathcal{QP}_i \subset \mathcal{K}_i$ the routing module returns $\cap\mathcal{M}_k, \forall k \in \mathcal{QP}_i$ which would give the set of all virtual nodes that contain tuples with the corresponding primary key values. The lookup and intersection operations are quite efficient as the size of these tables is small and the operations can be done in memory. Third, if $\mathcal{QP}_i = null$, i.e., the query predicates do not contain any primary key attributes, the union set $\mathcal{M}_i$ of the corresponding relation $\mathcal{R}_i$ is returned. Here no computation is involved as the pre-computed union set is returned. In the next step, the router determines the virtual node replicas, and their location information using the mappings obtained from the data partitioner. It gives this information as input to the set-cover algorithm which computes the minimum partition set that meets the quorum requirement on which

[4]The Cartesian product of the full set of attributes forming the primary key.

Table 1: Router memory requirements

Scheme	Fine-grained	CR=3	CR=6	CR=11	CR=28
Mappings size	20 GB	8GB	4GB	2.2GB	857MB

the transaction needs to be executed.

4. EXPERIMENTAL EVALUATION

In this section, we present the results of the experimental evaluation of our system. We first provide some details of our system implementation in Section 4.1 followed by our experimental setup in Section 4.2. We then provide an experimental analysis of our hypergraph compression technique, and discuss the effects of our techniques on router efficiency and system throughput. We then evaluate our fine-grained quorum technique and its effect on the end-to-end system performance.

4.1 System Implementation

We have used PostgreSQL 8.4.8 as the relational DBMS system and Java-6-OpenJDK SE platform for developing and testing our framework. For hypergraph generation we follow an approach similar to that of *Schism* [7] wherein we use the PostgreSQL logs to determine the queries run by the benchmark. We have developed a query transformation module that transforms each query into an equivalent SELECT SQL query, from which we can extract the primary keys of the tuples accessed by the query to build the hypergraph. For executing the transactions the transaction manager uses the Java transaction API's (JTA) XAResource interface to interact with the DBMS resource managers running on the individual partitions and executes transactions as per the 2-phase commit protocol.

4.2 Experimental Setup

This section provides the details of our system configuration, workloads, datasets and the baselines used.

4.2.1 System configuration

Our system deployment on Amazon EC2 consists of one router cum transaction manager and 10 database partitions each running an instance of a PostgreSQL 8.4.8 server running with a *read committed* isolation level. The router configuration consists of 7.5 GB memory, 4 EC2 Compute Units, 850 GB storage, and a 64-bit platform with Fedora Core-8. The 10 database partitions are run on separate EC2 instances each with a configuration of 1.7 GB memory, 1 EC2 compute unit, 160 GB instance storage, 32-bit platform with Fedora Core-8.

4.2.2 Workloads and datasets

We have used the TPC-C benchmark for our experimental evaluation which contains a variety of queries: 48% write queries (update and insert), 47% read queries (select), and 5% aggregate queries (sum and count). The database was horizontally partitioned into ten partitions according to different partitioning schemes for the purposes of experimental evaluation. In order to experiment with a variety of different configurations, we used a dataset containing 1.5 Billion tuples, and different transactional workloads consisting of up to 10 million transactions. We have varied the percentage of read and write transactions to simulate different types of transactional workloads. In particular, we created three different mixes from the TPC-C workload: *Mix-1* consisting of 75% of read-only transactions and 25% write transactions, *Mix-2* consisting of 50% each of read and write transactions, and *Mix-3* consisting of 25% read and 75% write transactions.

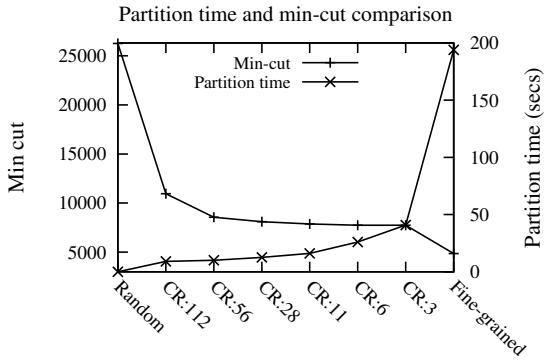


Figure 5: Effect of hypergraph compression on min-cut and partitioning time. (Note that the left y-axis does not start at 0.)

4.2.3 Baselines

In our experiments we compare the performance of our approach of using compressed hypergraphs (referred to as *compressed*) with two partitioning strategies as baselines: *Random (hash-based)* with 3-way replication, and *fine-grained* tuple-level hypergraph partitioning approach.

Random: We use tuple-level random partitioning with 3-way replication as our baseline. This approach is essentially the same as *hash* partitioning. We place the tuples using a hash function (specifically, by overriding the hashcode() function in java to a hash function of choice, MD5 in our case) with range $\{1, ..., P\}$ (P = number of partitions), and the resulting placement is nearly random. The 3-way replication is achieved using 3 different hash functions, with a post-processing step to ensure no two replicas land on the same partition.

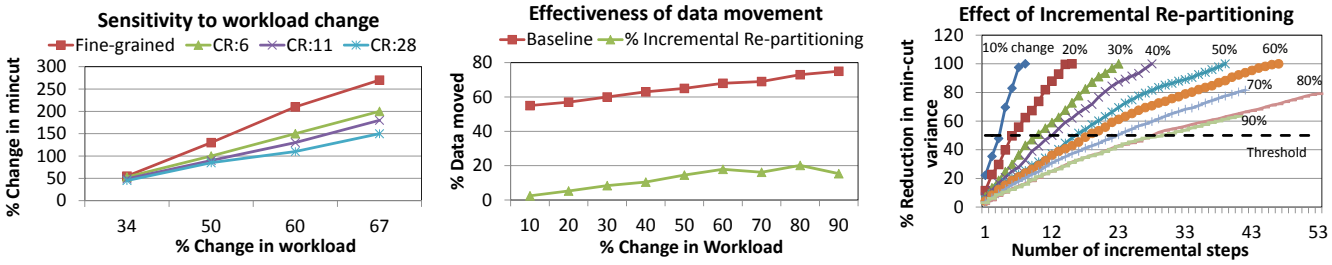
We note here that the TPC-C benchmark includes a large number of queries that access a small number of tuples, and a few queries that access a large number of tuples. In particular, we need to be able to handle queries where only **a portion of the primary key of a table is specified**. An example of such a query is:

select count (c_id) from customer where c_d_id = 3
and c_last = 'OUGTHCALLYPRES' and c_w_id =1.

The predicates of this query specify a partial primary key set and the query accesses more than one tuple. Therefore, in spite of using hash functions to map the tuples and their replicas to physical partitions, we still need tuple-level mapping tables to determine the locations of all tuples that are accessed by such queries.

Fine-grained: Fine-grained partitioning is obtained by first constructing a tuple-level hypergraph where each hyperedge represents a transaction and nodes spanned by the hyperedge represent tuples accessed by the transaction. We use tuple-level read-write access patterns to determine the number to replicas for each tuple and use in-graph replication which approximates an average 3-way replication for each tuple, and partition the hypergraph using hMetis to obtain a fine-grained partitioning.

Compressed: We generated compressed hypergraphs for different compression ratios and selected six different CRs (3, 6, 11, 28, 56, 112) as candidates for comparing our proposed technique with random and fine-grained partitioning schemes for our initial experiments in Section 4.3. Based on the results obtained we use a subset of these CRs (6, 11 and 28) for our experiments in the subsequent sections. Although our workload-aware replication scheme generates different number of replicas for each virtual node based on its access patterns, our scheme approximates an average 3-way repli-



(a) (b) (c)

Figure 6: (a) Fine-grained approach is more sensitive to workload changes than the compressed approach; (b) Our approach needs to move significantly smaller amount of data to maintain an effective partitioning compared to a baseline that does complete repartitioning; (c) Number of iterations required to bring down the increased min-cut under the threshold value.

cation in terms of the total number of replicas produced to provide a fair comparison with the other two techniques.

4.3 Hypergraph Compression Analysis

We explored the trade-off between the partitioning quality (min-cut) and the partitioning time for different compression ratios (CRs) and compared the same with our baselines (Figure 5). The number of distributed transactions is highest for the random partitioning scheme (26244) since it does not take into account the nature of the query workload at all. The min-cut for fine-grained is the minimum (4860) as it accurately represents the query workload at the tuple-level. The min-cuts for the compressed graphs lie in between random and fine-grained, and their magnitudes vary closely in accordance with the compression ratio of the hypergraph, ranging from 10944 for a compression ratio of 112, to 7740 for a compression ratio of 3. On the other hand, the hypergraph partitioning time is highest for the fine-grained and decreases significantly with the increase in the CR of the hypergraph. The partitioning time for random is 0 since it does not involve hypergraph partitioning and places the tuples randomly on different partitions.

There is a clear trade-off between the partitioning time and the min-cut. On one hand, a decrease in min-cut represents a reduction in the number of distributed transactions while a reduction in the partitioning time plays a crucial role in reducing the overall costs associated with partitioning and repartitioning the database. An interesting thing to note here is that there is little variation in the min-cut as the compression ratio is increased from 3 to 56 which gives us the flexibility of compressing the graph without paying too much penalty in terms of the increase in the number of distributed transactions and at the same time making the system more scalable and efficient in terms of handling larger query workloads. Based on these results we have chosen CRs 6, 11, 28, as potential sweet spots which have a reasonable min-cut and a substantially lower partitioning time for our further experiments. We advocate using an analogous analysis phase to choose the CR for other scenarios.

4.4 Effect of Workload Change

To ascertain the sensitivity of our approach to workload change and its impact on system performance, we conducted an experiment (Figure 6(a)) to evaluate the percentage change in the number of distributed transactions against the variation in the workload. For the purpose of the experiment, data was partitioned as per different partitioning strategies (fine-grained, our compressed hypergraph scheme with CRs 6, 11 and 28) for a given workload. Thereafter, the workload was varied by removing some old transactions and adding some new transactions. The variation in the number of distributed transactions was observed against the percentage change in workload. As can be seen, fine-grained partitioning was the most sensitive to workload change and the compressed hypergraph schemes were able to absorb the effect of workload change to a much greater extent with a smaller change in the number of distributed transactions for the same percentage change in workload.

Experimental evaluation of our incremental partitioning module highlights its effectiveness in dealing with the performance variation due to workload change. Figure 6(b) shows the number of data items that need to be moved in terms of percentage of the total number of data items placed as the workload changes. We see that our scheme can handle up to a 90% change in workload by migrating up to a maximum of 20% of data items as compared to the baseline which represents the amount of data required to be migrated when performing a complete repartitioning of the database. Figure 6(c) shows the number of incremental steps required to bring the increased min-cut (due to workload change) to a value below the required threshold value. We see that the number of iterations required to bring the variation (or increase) in min-cut below the threshold value increases as the % change in workload increases. We compute the % change of workload in terms of the fraction of hyperedges (transactions) affected by the workload change. The number of steps would vary with the value of k, the number of swaps that can be done in one iteration. The results shown are for $k = 10$.

4.5 Routing Efficiency and Quality

We measure the router efficiency in terms of the query routing time (comprising of query pre-processing time, and set-cover computation time) and routing quality in terms of the query span. The query pre-processing time includes the time for query parsing, determining the tuples accessed by the query, their replicas and their locations. We study the variation of router efficiency and quality with the variation in the min-cut of the hypergraph. All plots in this section show average quantities per query over 350K TPC-C queries.

Query routing time: Figure 7(a) shows the comparison of query routing times for different partitioning schemes on the log scale. Random and fine-grained partitioning have much higher query pre-processing times since they require lookups into large tuple-level routing tables, whereas, the average query pre-processing time for the compressed hypergraph reduces with the increase in the compression ratio and is substantially smaller than the other two par-

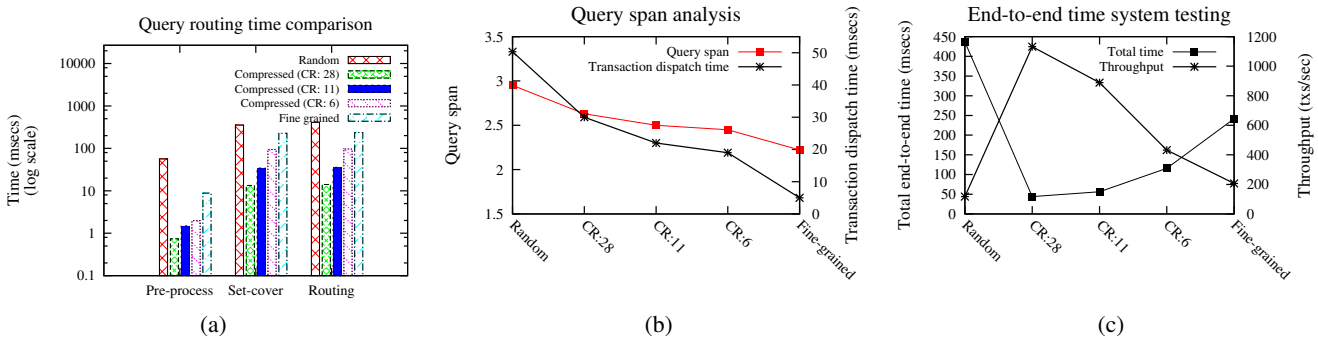


Figure 7: (a) Effect of hypergraph compression in minimizing the query pre-processing time and the set-cover computation time (note that the y-axis is in log scale); (b) The transaction dispatch time is directly dependent on the query span. (c) End-to-end system performance in terms of the end-to-end transaction time and the throughput for the compared schemes.

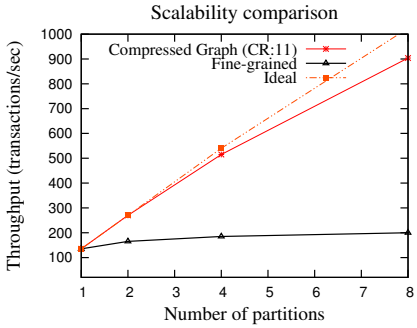


Figure 8: Effect of increasing parallelism on throughput: due to the high query routing costs, the fine-grained approach is not able to effectively utilize the available parallelism.

titioning schemes. This can be attributed to lookups into much smaller hash tables making the system scalable for handling large query workloads.

The set-cover computation time is dependent on the size of the partition-wise list of data items accessed by the query. For tuple-level partitioning schemes, this partition-wise list is in terms of individual tuples and for the compressed graph partitioning schemes it is in terms of virtual nodes . Consequently, random and fine-grained partitioning have a much higher set-cover computation time . The set cover time decreases with increase in CR and compressed graph partitioning with the highest compression ratio (CR:28) having the lowest set-cover computation time.

Query span analysis: Figure 7(b) gives a comparison of the average query span for different partitioning schemes and compares its effect on the transaction dispatch time which is the time taken by the transaction manager for executing transactions on the distributed database partitions[5]. The query span is a measure of the quality of data placement achieved. Random partitioning has the highest query span and the fine-grained partitioning has the lowest , while the query spans of the compressed hypergraphs for different CRs fall between those two. The transaction dispatch time closely follows the distribution of the average query spans, wherein a higher query span results in a higher transaction dispatch time. It is pertinent to note here that the reduction in transaction dispatch time achieved using fine-grained partitioning as compared to compressed hypergraph is not as significant as the orders of magnitude reduction in routing time achieved through graph compression and an efficient routing mechanism.

[5]It does not include the query routing time.

End-to-end system testing: Figure 7(c) shows the comparison of the end-to-end transaction times and throughput measurements on 10 partitions for different partitioning schemes on the log-scale. We see a substantial reduction in the total end-to-end transaction time and a high throughput for the compressed graph partitioning scheme with different CRs as compared to random and fine-grained partitioning. This can be attributed to the substantial reduction in routing time due to hypergraph compression and a reasonably good data placement as compared to fine-grained. Fine-grained does better than random due to better data placement and consequently a reduced transaction dispatch time and improved transaction throughput.

Throughput scalability: In order to test the end-to-end scalability of our system, we partitioned the workload using different partitioning schemes onto 1, 2, 4, and 8 partitions and used the TPC-C workload to ascertain the throughput as compared to an ideal linear speed-up for an embarrassingly parallel system. The results (Figure 8) indicate that the compressed graph scheme starts with a throughput of 165 transactions per second and achieves a throughput of 904 transactions per second for 8 partitions which is substantially higher than that achieved by the fine-grained partitioning scheme which can be attributed to its large query processing time. The non-linear speed-up is due to contention inherent in the TPC-C workload.

4.6 Fine-grained Quorum Evaluation

To ascertain the suitability of different types of quorums for different transactional workloads, we used different proportions of reads and writes to generate different workload mixes. For this set of experiments, we used Mix-1, Mix-2, and Mix-3 data sets, a CR of 11 for the compressed graph, and compared its performance with random and fine-grained for different types of quorums.

We studied the variation of query span and total transaction time for ROWA (read-one-write-all) and Majority quorum for different query workload mixes. Our results (Figure 9) validate that for read-heavy transactional workloads, ROWA gives the minimum query spans and end-to-end transaction times, while the Majority quorum performs better for write heavy loads as they reduce the cost of distributed updates. Thus the experiments demonstrate that the choice of quorum depending on the query workload significantly impacts performance.

Figures 9(d) and 9(e) show the effect of fine-grained quorums on average query span and system throughput respectively. We experimented with different values of γ, the threshold value of $\mathcal{R}$ used by the router for deciding the type of quorum for a given node. We

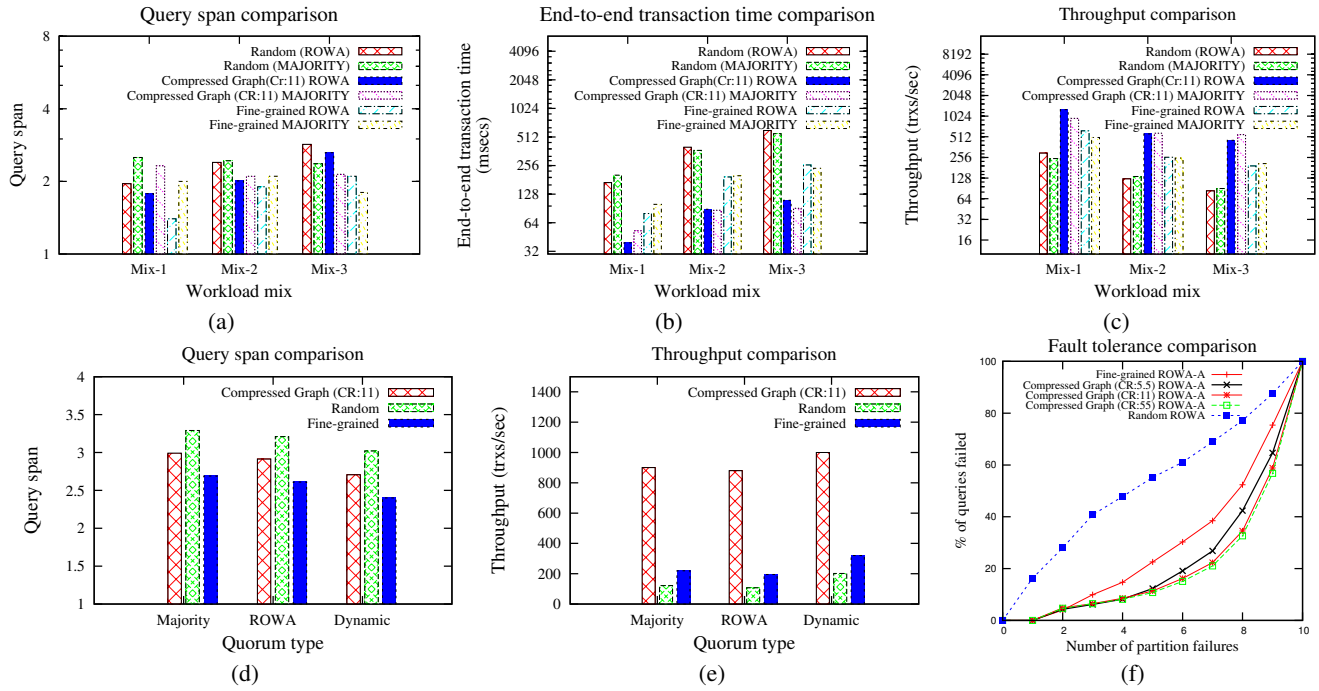


Figure 9: (a)-(c) The impact of the choice of quorum on the performance for different transactional workload mixes. The different query workload mixes shown are: Mix-1 {75% read, 25%write}, Mix-2 {50% read, 50% write}, Mix-3 {25% read, 75%write}. (d)-(e) The impact of fine-grained quorums on query span and system throughput. (f) The effect of data placement on fault tolerance.

show the plots for $\gamma = 0.5$ which provided the best results for the TPC-C workload. Use of fine-grained quorums reduces the average query span and increases throughput for all the partitioning strategies considered as compared to a fixed choice of ROWA or Majority for all data items, making the system adaptable to different workloads.

4.7 Dealing with Failures

We evaluate the effect of quorums and data placement on the ability of the system to deal with failures for the TPC-C workload. Figure 9(f) shows the percentage of query failures as a function of the number of partition failures. We randomly fail a given number of partitions for a given run and see its effect on the query failures. Each point on the plot is an average of 10 runs. The results show the effectiveness of our proposed technique in terms of fault tolerance and indicate that fine-grained partitioning schemes may not perform well in presence of faults, due to a very high degree of data co-location. Our compression scheme does a relatively modest co-location of data, thereby naturally maintaining a balance between minimizing distributed transactions and providing better fault tolerance. Our experimental results show that fault tolerance improves as the CR increases. Random with ROWA provides us with a baseline to see the quantum of improvement for different data placement strategies using read one write all *available* (ROWA-A), which excludes the copies on failed partitions.

5. RELATED WORK

As distributed databases have grown in scale, partitioning and data replication to minimize overheads and improve scalability has received a lot of interest. Nehme et al. [20] have developed a system to automatically partition the database based on the expected workload. Their approach is tightly integrated with the query optimizer which relies on database statistics. However their approach ignores the structural and access correlations between queries that we consider by modeling the query workload as a hypergraph.

The partitioning strategies in cloud/NoSQL systems [18, 6] primarily aim for scalability by compromising the consistency guarantees. Moreover, the partitioning in [18] cannot be changed without reloading the entire dataset. On the other hand, we endeavor to scale OLTP workloads while maintaining transactional ACID properties and provide an incremental repartitioning mechanism to deal with workload changes.

Workload-aware approaches have also been used in the past (e.g., AutoAdmin project [4]) for tuning the physical database design, i.e., identifying the physical design structures such as indexes for a given database and workload. Kumar et al. [17] propose a workload-aware approach for data placement and co-location for read-only analytical workloads. The solutions proposed in that paper focus on optimizing the energy and resource consumption, unlike our work that deals with data placement for minimizing distributed transactions for OLTP workloads. Pavlo et al. [21] propose a workload-aware approach for automatic database partitioning for OLTP applications. However, their approach does not provide an incremental mechanism to deal with workload changes once data is partitioned.

Among workload-aware data placement approaches, Schism [7], is closest to our work, although there are significant differences in handling the critical issues of scalability, availability, fault-tolerance and dealing with workload changes. Schism does not provide any mechanism to deal with workload changes. Another difference between our framework and Schism is the use of aggressive replication. Schism trades-off performance for fault tolerance by not replicating data items with a high write/update frequency. This might compromise the availability of these data items in presence of failure and affect the ability to do load balancing across multiple machines. We instead replicate each tuple at least once, and possibly more times depending on the access frequencies. The replicas are kept strongly consistent. We also empirically show that our ap-

proach is more fault-tolerant than tuple-level fine-grained partitioning techniques. In a recent follow-up work to Schism, Tatarowicz et al. [25] use a powerful router with high memory and computational resources and employ compression to scale up the lookup tables. However, that approach suffers from the same issues as a scaled-up architecture and may not be cost effective because it needs large memory and computation resources. In our work, we minimize the lookup table sizes significantly by using a compressed representation of the workload.

Although there has been a significant amount of work on graph compression techniques, e.g., for community detection [9], graph summarization [19], compressing web graphs [14], finding communities and coalescing nodes in same community [5], these are not appropriate in our settings of scaling out OLTP workloads as they cannot be used to minimize the cost of query execution. Moreover, the use of these techniques that preserve graph structural information would require the router to store additional metadata to route the queries to appropriate partitions, which is an additional overhead and not conducive to scalability.

Replication has been widely used in distributed databases for availability, load balancing and fault tolerance [16, 15, 24]. In this paper, we focus on active replication as it provides increased availability and load balancing. Gray et al. [10] showed that replication in distributed databases can result in performance bottlenecks and can limit their scalability. We address the performance issues related to replication by using a workload-aware replication technique which further minimizes distributed transactions with the use of in-graph replication and control update costs using fine-grained quorums. Although quorums have been used to alleviate the cost of replica updates [22], we have shown that a static choice of quorums for all data items is not sufficient to handle different workloads with varying access patterns.

6. CONCLUSION

In this paper, we presented SWORD, a scalable framework for data placement and replication for supporting OLTP workloads in a cloud-based environment, that exploits available workload information. We presented a suite of techniques to reduce the cost and maintenance overheads of graph partitioning-based data placement techniques and to minimize the number of distributed transactions, while catering for fault tolerance, increased availability, and load balancing using active replication. We proposed an effective incremental repartitioning technique to maintain a good partitioning in presence of workload changes. We also explored the use of fine-grained quorums to reduce the query spans and thus improve throughputs. The use of fine-grained quorums provides our framework with the ability to seamlessly handle different workloads, an essential requirement for cloud-based environments. We have carried out an exhaustive experimental study to investigate the trade-offs between routing efficiency, partitioning time, and the quality of partitioning. Our framework provides a flexible mechanism for determining a sweet spot in terms of the compression ratio of the hypergraph, that gives a reasonably good quality of partition, improves routing efficiency, and reduces partitioning costs substantially. We have shown experimentally that our incremental repartitioning technique mitigates the impact of workload change with minimal data movement and that our proposed data placement scheme naturally improves resilience to failures.

Acknowledgements: This work was supported in part by NSF grant CCF-0937865, an IBM Collaborative Research Award, and an Amazon AWS in Education Research grant.

7. REFERENCES

[1] hMetis: a hypergraph partitioning package, http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview.

[2] C. Ayka, B.Cambazoglu, and U. Bora. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel Distrib. Comput.*, 2008.

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

[4] N. Bruno, S. Chaudhuri, A. C. Konig, V. R. Narasayya, R. Ramamurthy, and M. Syamala. Autoadmin project at Microsoft Research: Lessons learned. *IEEE Data Eng. Bull.*, 2011.

[5] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *WSDM*, 2008.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06*.

[7] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, September 2010.

[8] D. J. Dewitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 1992.

[9] S. Fortunato. Community detection in graphs. *Physics Reports*, 2010.

[10] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.

[11] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 2003.

[12] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, 2010.

[13] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 2008.

[14] C. Karande, K. Chellapilla, and R. Andersen. Speeding up algorithms on compressed web graphs. In *WSDM*, 2009.

[15] B. Kemme and A. Gustavo. Database replication: a tale of research across communities. *PVLDB*, September 2010.

[16] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. *Database Replication*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[17] K. A. Kumar, A. Deshpande, and S. Khuller. Data placement and replica selection for improving colocation in distributed environments. Unpublished manuscript, 2012.

[18] A. Lakshman and P. Malik. Cassandra: Structured storage system on a P2P network. In *PODC '09*.

[19] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, 2008.

[20] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, 2011.

[21] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, 2012.

[22] J. R. Peris, M. P. Martinez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM TODS*, 28(3) 2003.

[23] R. J. Peris and M. P. Martinez. How to select a replication protocol according to scalability, availability and communication overhead. In *SRDS*, 2001.

[24] R. J. Peris, M. P. Martnez, B. Kemme, and G. Alonso. How to select a replication protocol according to scalability, availability, and communication overhead. *IEEE Symposium on RDS*, 2001.

[25] A. L. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *ICDE*, 2011.

[26] X. Wang, A. Smalter, J. Huan, and G. H. Lushington. G-hash: towards fast kernel-based similarity search in large graph databases. In *EDBT*, 2009.