Schism: a Workload-Driven Approach to Database Replication and Partitioning

Carlo Curino
curino@mit.edu

Evan Jones
evanj@mit.edu

Yang Zhang
yang@csail.mit.edu

Sam Madden
madden@csail.mit.edu

ABSTRACT

We present Schism, a novel workload-aware approach for database partitioning and replication designed to improve scalability of shared-nothing distributed databases. Because distributed transactions are expensive in OLTP settings (a fact we demonstrate through a series of experiments), our partitioner attempts to minimize the number of distributed transactions, while producing balanced partitions. Schism consists of two phases: i) a workload-driven, graph-based replication/partitioning phase and ii) an explanation and validation phase. The first phase creates a graph with a node per tuple (or group of tuples) and edges between nodes accessed by the same transaction, and then uses a graph partitioner to split the graph into k balanced partitions that minimize the number of cross-partition transactions. The second phase exploits machine learning techniques to find a predicate-based explanation of the partitioning strategy (i.e., a set of range predicates that represent the same replication/partitioning scheme produced by the partitioner).

The strengths of Schism are: i) independence from the schema layout, ii) effectiveness on n-to-n relations, typical in social network databases, iii) a unified and fine-grained approach to replication and partitioning. We implemented and tested a prototype of Schism on a wide spectrum of test cases, ranging from classical OLTP workloads (e.g., TPC-C and TPC-E), to more complex scenarios derived from social network websites (e.g., Epinions.com), whose schema contains multiple n-to-n relationships, which are known to be hard to partition. Schism consistently outperforms simple partitioning schemes, and in some cases proves superior to the best known manual partitioning, reducing the cost of distributed transactions up to 30%.

1. INTRODUCTION

The primary way in which databases are scaled to run on multiple physical machines is through horizontal partitioning. By placing partitions on different nodes, it is often possible to achieve nearly linear speedup, especially for analytical queries where each node can scan its partitions in parallel. Besides improving scalability, partitioning can also improve availability, by ensuring that when one partition fails the remaining partitions are able to answer some of the transactions, and increase manageability of the database by enabling rolling upgrades and configuration changes to be made on one partition at a time.

Although a number of automatic partitioning schemes have been investigated [1, 23, 18], the mostly widely used approaches are round-robin (send each successive tuple to a different partition), range (divide up tuples according to a set of predicates), or hash-partitioning (assign tuples to partitions by hashing them) [6]. All of these can be effective for analytical queries that scan large datasets.

Unfortunately, for workloads consisting of small transactions that touch a few records, none of these approaches is ideal. If more than one tuple is accessed, then round robin and hash partitioning typically require access to multiple sites. Executing distributed transactions reduces performance compared to running transactions locally. Our results in Section 3 show that using local transactions doubles the throughput. Range partitioning may be able to do a better job, but this requires carefully selecting ranges which may be difficult to do by hand. The partitioning problem gets even harder when transactions touch multiple tables, which need to be divided along transaction boundaries. For example, it is difficult to partition the data for social networking web sites, where schemas are often characterized by many n-to-n relationships.

In this paper, we present Schism, a novel graph-based, data-driven partitioning system for transactional workloads. Schism represents a database and its workload using a graph, where tuples are represented by nodes and transactions are represented by edges connecting the tuples used within the transaction. We then apply graph partitioning algorithms to find balanced partitions that minimize the weight of cut edges, which approximately minimizes the number multi-sited transactions. Schism can be tuned to adjust the degree to which partitions are balanced in terms of workload or data size, and is able to create partitions that contain records from multiple tables.

In addition to this new graph-based approach, Schism makes several additional contributions:

- We show that the throughput of executing small distributed transactions is significantly worse than executing them on a single node.
- We show that Schism is able to replicate records that are infrequently updated. This increases the fraction of transactions that are "single-sited" (go to just one site). Unlike existing partitioning techniques, this approach is able to replicate just a portion of a table.
- We present a scheme based on decision trees for identifying predicates (ranges) that "explain" the partitioning identified by the graph algorithms.
- We show that Schism's partitioning times are reasonable for large datasets. The tool takes on the order of just a few

Proceedings of the VLDB Endowment, Vol. 3, No. 1

minutes, for databases containing millions of tuples. We also propose and evaluate heuristics, including sampling and grouping of tuples, to limit the graph size to further reduce partitioning times.

- Finally, we demonstrate that Schism can find good partitions for several challenging applications, consistently doing as well or better than hash partitioning and manually designed range partitioning. In the most difficult case we tested, consisting of a complex n-to-n social network graph, Schism outperformed the best manual partitioning, reducing distributed transactions by an additional *30%*.

While this paper applies our partitioning approach to disk-based shared-nothing databases, it applies to other settings, including main-memory databases like H-Store [21] that depend on highly partitionable workloads for good performance, and in automating the creation of "sharded" databases where it is important to minimize the number of cross-shard joins.

The remainder of this paper is organized as follows: we provide an overview of our approach in Section 2, discuss the cost of distributed transactions in Section 3, present the core ideas of our work in Section 4, discuss implementation and optimization challenges in Section 5, we show our experimental validation in Section 6, compare with related work in Section 7, and conclude in Section 8.

2. OVERVIEW

The input to our partitioning system is a database, a representative workload (e.g., an SQL trace), and the number of partitions that are desired. The output is a partitioning and replication strategy that balances the size of the partitions while minimizing the overall expected cost of running the workload. As we will see in the next section, in OLTP workloads, the expected cost is directly related to the number of distributed transactions in the workload. The basic approach consists of five steps:

Data pre-processing: The system computes read and write sets for each transaction in the input workload trace.

Creating the graph: A graph representation of the database and workload is created. A node is create for each tuple. Edges represent the usage of tuples within a transaction. An edge connects two tuples if they are accessed by the same transaction. A simple extension of this model allows us to account for replicated tuples, as discussed in Section 4.

Partitioning the graph: A graph partitioning algorithm is used to produce a balanced minimum-cut partitioning of the graph into k partitions. Each tuple is assigned to one partition (i.e., *per-tuple partitioning*), and each partition is assigned to one physical node.

Explaining the partition: The system analyzes the statements in the input trace to collect a list of attributes frequently used in the `WHERE` clauses for each table, which we call a *frequent attribute set*. A decision tree algorithm is used to extract a set of rules that compactly represent the per-tuple partitioning. The rules are predicates on the values of the frequent attribute set that map to partition numbers. We call this *range-predicate partitioning*.

Final validation: The cost of per-tuple partitioning, and range-predicate partitioning are compared against hash-partitioning and full-table replication, using the total number of distributed transactions as the metric. The best strategy is selected, and in the event of a tie the simplest solution is preferred.

The resulting partitioning can be: (i) installed directly into a DBMS that supports partitioning for a distributed shared-nothing architecture, (ii) explicitly encoded in the application, or (iii) implemented in a middleware routing component, such as the one we developed as to test Schism.

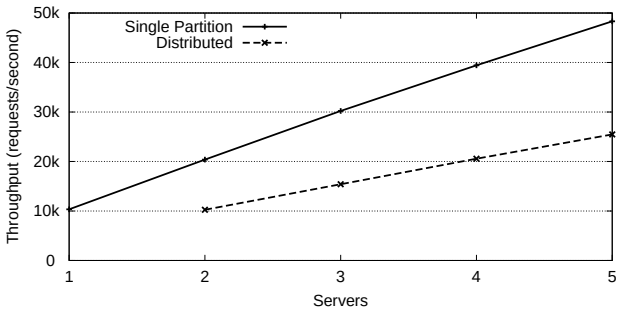


Figure 1: Throughput of distributed transactions

3. THE PRICE OF DISTRIBUTION

Before describing the details of our partitioning approach, we present a series of experiments we conducted to measure the cost of distributed transactions. These results show that distributed transactions are expensive, and that finding a good partitioning is critical for obtaining good performance from a distributed OLTP database.

In a distributed shared-nothing database, transactions that only access data on a single node execute without additional overhead. Statements in the transaction are sent to one database followed by the final commit or abort command. However, if the statements are distributed across multiple nodes, two-phase commit or a similar distributed consensus protocol is required to ensure atomicity and serializability. This adds network messages, decreases throughput, increases latency, and potentially leads to expensive distributed deadlocks. Hence, we wish to avoid distributed transactions.

To quantify the performance impact of distributed transactions, we performed a simple experiment using MySQL. We created a table `simplecount` with two integer columns: *id* and *counter*. Clients read two rows in a single transaction by issuing two statements of the form: `SELECT * FROM simplecount WHERE id = ?`. Each statement returns one tuple. We tested the performance of two partitioning strategies: (i) every transaction is run on a single server, and (ii) every transaction is distributed across multiple machines, using MySQL's support for two-phase commit (XA transactions), with our own distributed transaction coordinator.

Each client selects rows at random according to one of these two strategies, and after getting the response immediately sends the next request. We tested this workload for up to five servers. See Appendix A for a detailed description of our experimental configuration. We used 150 simultaneous clients, which was sufficient to saturate the CPU of all five database servers. The `simplecount` table contains 150k rows (1k for each client), and thus the overall database fits entirely into the buffer pools of our servers (128 MB). This simulates a workload that fits in RAM, which is not uncommon for OLTP.

In an ideal world, the local and distributed transactions in this test would achieve similar performance, since they access the same number of records with the same number of statements. However, the results in Figure 1 show that distributed transactions have a large impact on throughput, reducing it by about a factor of 2. Since a distributed transaction commits across all the participants, latency is also worse. In this experiment, the distributed transactions have approximately double the average latency. For example, for five servers the single-site transaction latency is 3.5 ms, while it is 6.7 ms for distributed transactions. We ran similar experiments for several other scenarios including update transactions, and transactions that accessed more tuples, with similar results.

Real OLTP applications are far more complicated than this simple experiment, potentially introducing: (i) contention, due to trans-

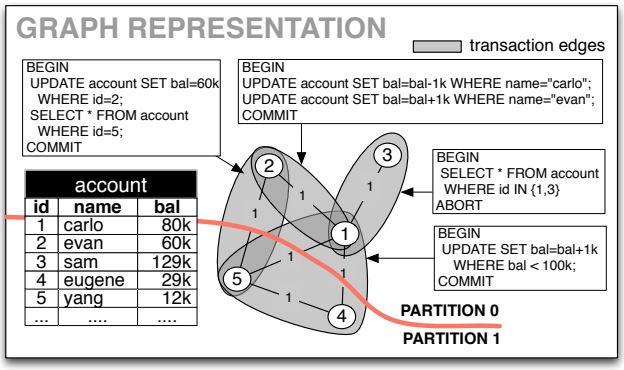


Figure 2: The graph representation

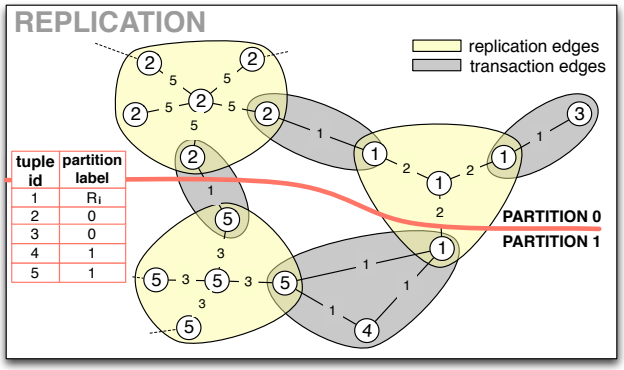


Figure 3: Graph with replication

actions accessing the same rows simultaneously while locks are held—as we found in TPC-C in Section 6.3, (ii) distributed deadlocks, and (iii) complex statements that will need to access data from multiple servers, e.g., distributed joins. All of the above will further reduce the overall throughput of distributed transactions. On the other hand, more expensive transactions will reduce the impact of distribution, as the work performed locally at a partition dwarfs the cost of extra message processing for two-phase commit.

Schism is designed for OLTP or web applications. For these workloads, the conclusion is that *minimizing the number of distributed transactions while balancing the workload across nodes substantially increases transaction throughput*.

4. PARTITIONING AND REPLICATION

Having established the cost of distributed transactions, we present our approach for partitioning and replication, which attempts to distribute data so that transactions access only a single partition.

4.1 Graph Representation

We introduce our graph representation with a simple example. While our example only uses a single table, our approach works with any schema, and is independent of the complexity of the SQL statements in the workload. Suppose we have a bank database containing a single `account` table with five tuples, and a workload of four transactions, as shown in Figure 2. Each tuple is represented as a node in the graph; edges connect tuples that are used within the same transaction. Edge weights account for the number of transactions that co-access a pair of tuples. These are not shown in the figure, as each pair of tuples is accessed by at most one transaction.

In Figure 3, we introduce an extension of the basic graph representation that captures the opportunity for tuple-level replication. Replication is represented by "exploding" the node representing a single tuple into a star-shaped configuration of $n + 1$ nodes where n is the number of transactions that access the tuple.

As an example, consider the tuple `(1, carlo, 80k)` from Figure 2. This tuple is accessed by three transactions and is therefore represented by four nodes in Figure 3. The weights of the replication edges connecting each replica to the central node represent the cost of replicating the tuple. This cost is the number of transactions that update the tuple in the workload (2 for our example tuple). When replicating a tuple, each read can be performed locally, but each update becomes a distributed transaction. This graph structure allows the partitioning algorithm to naturally balance the costs and the benefits of replication.

We experimented with other graph representations, including hypergraphs, but found that they did not perform as well (see Appendix B for further details on our graph representation).

The graph partitioning strategy discussed in the next section heuristically minimizes the cost of a graph cut, while balancing the weight of each partition (more precisely, by satisfying a constraint on the permissible skew). The weight of a partition is computed as the sum of the weights of the nodes assigned to that partition. Therefore, we can balance partitions in different ways by assigning different node weights. We have experimented with two useful metrics for database partitioning: (i) data-size balancing, achieved by setting node weights equal to the tuple size in bytes, and (ii) workload balancing, achieved by setting node weights equals to the number of times the tuple is accessed.

4.2 Graph Partitioning

Schism's graph representation characterizes both the database and the queries over it. Graph partitioning splits this graph into k non-overlapping partitions such that the overall cost of the cut edges is minimized (i.e., find a *min-cut*), while keeping the weight of partitions within a constant factor of perfect balance (where the degree of imbalance is a parameter). This graph operation approximates minimizing the number of distributed transactions while balancing the load or data size evenly across nodes.

Our unified representation of partitioning and replication allows the graph partitioning algorithm to decide for each tuple whether it is better to replicate it across multiple partitions and pay the cost for distributed updates (e.g. tuple 1 in Figure 3), or to place it in a single partition and pay the cost for distributed transactions (e.g. tuple 2 in Figure 3). When all replicas of a tuple are in the same partition, the tuple is not replicated. Otherwise, the partitioner has decided to replicate the tuple. The min-cut algorithm will naturally choose not to replicate tuples that are updated frequently, since there is a high cost for cutting the edges between the replicas of such tuples, representing the cost of distributed updates. Conversely, it is likely to replicate rarely updated tuples, if it reduce the cost of cutting transaction edges. In Section 4.3, we discuss how the tuple-level choices can be generalized into table-wide or range partitioning decisions.

Constrained, k-way graph partitioning is a known to be an *NP-complete* problem. However, given its fundamental role in VLSI and CAD it has been the subject of substantial research and development efforts over the past four decades [12, 10, 13], resulting in sophisticated heuristics and well-optimized, freely-available software libraries. Most graph partitioning algorithms exploit multilevel coarsening techniques and provide distributed parallel implementations to deal with extremely large graphs (hundreds of millions of edges). In Schism, we use *METIS* [11] to partition graphs.

The result of the graph partitioning phase is a fine-grained mapping between individual nodes (tuples) and partition labels. Because of replication, a tuple may be assigned to more partitions.

Fined-grained Partitioning. One way to use the output of our partitioner is to store it in a *lookup table* such as the one shown on

the left-hand side of Figure 3.

In the common case of key-based access of the tuples (i.e., the queries `WHERE` clauses contain equality or range predicate over the the tuple IDs), these tables can be used directly to route queries to the appropriate partition. In our prototype this is achieved by means of a middleware routing component, that parses the queries and compare their `WHERE` clause predicates to the content of the lookup tables. Physically, lookup tables can be stored as indexes, bit-arrays or bloom-filters—see Appendix C for details regarding lookup tables. Assuming a dense set of IDs and up to 256 partitions, a coordinator node with 16GB of RAM can store a lookup table as one byte per ID and record partitioning information for over 15 billion tuples (the coordinator has no other state that places significant demands on RAM). This is more than sufficient for the vast majority of OLTP applications. Furthermore, such lookup tables can stored in a distributed fashion over several machines or in a disk-based index if they exceed available RAM.

With lookup tables, new tuples are initially inserted into a random partition until the partitioning is re-evaluated, at which point they may be moved to the appropriate partition. Because partitioning is fast, we envision running it frequently, which will prevent newly added tuples from causing excessive distributed transactions.

While this approach is effective when a very fine-grained partitioning strategy is required (e.g., in some social-network applications), it is not ideal for very large databases or very extremely insert-heavy workloads. Hence, we developed an analysis tool than can find simpler, predicate-based partitioning of the data that closely approximates the output of the partitioner.

4.3 Explanation Phase

The *explanation phase* attempts to find a compact model that captures the $(tuple, partition)$ mappings produced by the partitioning phase. To perform this task, we use decision trees (a machine learning classifier) since they produce understandable rule-based output that can be translated directly into predicate-based range partitioning. A decision tree classifier takes a collection of $(value, label)$ pairs as input, and outputs a tree of predicates over values leading to leaf nodes with specific labels. Given an unlabeled value, a label can be found by descending the tree, applying predicates at each node until a labeled leaf is reached.

In Schism, the values are database tuples and the labels are the partitions assigned by the graph partitioning algorithm. Replicated tuples are labeled by a special *replication identifier* that indicates the set of partitions that should store the tuple (e.g. the partition set $\{1, 3, 4\}$ can be represented with label R_1).

When successful, the classifier finds a simple set of rules that capture, in a compact form, the essence of the partitioning discovered by the graph partitioning algorithm. For the example in Figures 2 and 3, the decision tree classifier derives the following rules:

$$\begin{aligned} (id = 1) &\rightarrow \ partitions = \{0, 1\} \\ (2 \leq id < 4) &\rightarrow \ partition = 0 \\ (id \geq 4) &\rightarrow \ partition = 1 \end{aligned}$$

It is not always possible to obtain an *explanation* of a partitioning, and not every explanation is useful. An explanation is only useful if (i) it is based on attributes used frequently in the queries (e.g., in our example 50% of the queries use the `id` attribute in their `WHERE` clause)—this is needed to route transactions to a single site and avoid expensive broadcasts, (ii) it does not reduce the partitioning quality too much by misclassifying tuples, and (iii) it produces an explanation that works for additional queries (overfitting can create an explanation that works for the training set, but uses attributes that do not represent the application).

To achieve these goals we: (i) limit the decision tree to operate on attributes used frequently in the queries, (ii) measure the cost in terms of number of distributed transactions (based on Section 3) and discard explanations that degrade the graph solution, and (iii) use aggressive pruning and cross-validation to avoid over-fitting. More details on the implementation of our explanation process are provided in Section 5.2.

4.4 Final Validation

The goal of the validation phase is to compare the solutions obtained in the two previous phases and to select the final partitioning scheme. Specifically, we compare the fine-grained per-tuple partitioning produced by the graph partitioner, the range-predicate partitioning scheme produced in the explanation phase, hash-partitioning on the most frequently used attributes and full-table replication, using the number of distributed transactions as an estimate of the cost of running the workload. We choose the scheme that has the smallest number of distributed transactions, unless several schemes have close to the same number, in which case we choose the one with lowest complexity (e.g., we prefer hash partitioning or replication over predicate-based partitioning, and predicate-based partitioning over lookup tables). The importance of this process is demonstrated in two experiments in Section 6, where the system selects simple hash-partitioning over predicate partitioning for several simple workloads where hash partitioning works well.

5. OPTIMIZATION & IMPLEMENTATION

In this section, we briefly present some of the key engineering challenges and design decisions we faced when building Schism.

5.1 Achieving Scalability

A practical partitioning tool must be capable of handling very large databases. As the size of the database and the number of tuples accessed by each transaction increases, the graph representation grows. If a workload contains many different transactions, a larger workload trace will be needed to capture representative behavior. As the number of partitions increases, more cuts must be found. These factors could impose limits on the size of database our system is capable of handling.

As we show in Section 6, graph partitioners scale well in terms of the number of partitions, but their runtime increases substantially with graph size. Therefore, we focused our effort on reducing the size of the input graph. Intuitively, this decreases the running time of the partitioning algorithm while sacrificing the quality, since a smaller input graph contains less information. However, we developed a number of heuristics that reduce the size of the graph with a limited impact on the quality of the partitioning. Specifically, we implemented the following heuristics:

Transaction-level sampling, which limits the size of the workload trace represented in the graph, reducing the number of edges.

Tuple-level sampling, which reduces the number of tuples (nodes) represented in the graph.

Blanket-statement filtering, which discards occasional statements that scan large portions of a table, since (i) they produce many edges carrying little information, and (ii) parallelize across partitions effectively, since the distributed transaction overhead is less significant than the cost of processing the statement.

Relevance filtering, which removes tuples that are accessed very rarely from the graph, since they carry little information. The extreme case is a tuple that is never accessed.

Star-shaped replication, connecting replica nodes in a star-shaped configuration rather than in a clique, limiting number of edges.

Tuple-coalescing, which represents tuples that are always accessed together as a single node. This is a lossless transformation that reduces the graph size significantly in some cases.

These heuristics have proven to be effective for reducing the graph size for our benchmark suite, while maintaining high quality results. For example, in Section 6.2 we show that a graph covering as little as 0.5% of the original database using a few thousand transactions carries enough information to match the best human-generated partitioning scheme for TPC-C.

5.2 Implementing Explanation

To implement the explanation phase described in Section 4.3 and 4.4, we used the open-source machine-learning library Weka [9]. This process involves the following steps:

Create a training set: Schism extracts queries and the tuples accessed by the workload trace—sampling is applied to reduce running time, without affecting the quality of the result. The tuples are marked with the partition labels produced by the graph partitioning algorithm. Replicated tuples are assigned virtual partition labels that represent the set of destination partitions, as described in Section 4.3. This constitute the training set used by the decision tree classifier.

Attribute Selection: The system parses the SQL statements in the workload and records for each attribute the frequency with which it appears in the `WHERE` clauses. Rarely used attributes are discarded, since they would not be effective for routing queries. As an example, for the TPC-C `stock` table we obtain two frequently used attributes (`s_i_id, s_w_id`), representing item ids and warehouse ids. The candidate attributes are fed into Weka's correlation-based feature selection to select a set of attributes that are correlated with the partition label. For TPC-C, this step discards the `s_i_id` attribute, leaving `s_w_id` as the sole attribute for classification.

A more complex scenario is when tuples of one table are frequently accessed via a join with another table. Our system handles this by preparing the input to the classifier as the join of the two tables. The resulting partitioning will require co-location of the joined tuples in the same partition, and the predicates produced will be join predicates and ranges over the other table. The resulting predicate-based partitioning will only support queries that perform that join. Other queries will need to be sent to all partitions. Although such queries are not very common (none of applications in our benchmark suite include any), this is supported.

Build the classifier: We train a decision tree classifier using J48, a Java implementation of the C4.5 classifier [17]. Partition labels are the classification attribute we wish to learn from the candidate attributes generated from the previous step. The output of the classifier is a set of predicates that approximates the per-tuple partitioning produced by the graph partitioning algorithm. The risk of over-fitting is avoided via cross-validation and by controlling the aggressiveness of the pruning of the decision tree to eliminate rules with little support.

As an example, for TPC-C with two warehouses divided into two partitions we obtain the following rules for the `stock` table:

```
s_w_id <= 1:  partition: 1   (pred. error: 1.49%)
s_w_id >  1:  partition: 2   (pred. error: 0.86%)
```

The output of the classifier for the `item` table is:

```
<empty>:  partition: 0 (pred. error: 24.8%)
```

This indicates that all tuples in the table are replicated. The high prediction error in this example is an artifact of the sampling, since some tuples in the `item` table end up being accessed only by few transactions, thus providing no evidence that replication is needed. However, this does not affect the quality of the solution.

For TPC-C, the overall result is to partition the database by warehouse, while replicating the entire `item` table. This is the same strategy derived by human experts [21]. As we discuss in Section 6, similar partitionings are found for TPC-C with more warehouses and partitions. While TPC-C does not require multiple attributes to define partitions, the classifier can produce rules of this form, where appropriate.

5.3 Obtaining the Traces

To produce the graph representation, we need the set of tuples accessed by each transaction. We developed a tool that takes a log of SQL statements (e.g., the MySQL `general_log`), and extracts the read and write sets. First, SQL statements from the trace are rewritten into `SELECT` statements that retrieve the identifiers (e.g., primary keys) of each tuple accessed. These `SELECT`s are executed, producing a list of $(tuple\_id, transaction)$ pairs used to build the graph. This mechanism can either be used online, extracting the tuple IDs immediately after the original statement, or offline. Extracting tuples long after the original statements were executed still produces good partitioning strategies for our experimental data sets, which suggests our approach is insensitive to the exact tuples that are used. By combining this tolerance to stale data with sampling, we believe that read/write sets can be extracted with negligible performance impact on production systems.

5.4 Data Migration and Query Routing

There are two remaining implementation issues to be considered. The first is how we migrate data from one partition to another. To do this, Schism generates data migration SQL scripts. The current version is designed to partition a single database into partitions. We are extending this to solve the more general problem of repartitioning. Alternatively, the output of our tool can be fed into a partition-aware DBMS such as Oracle or DB2, that can manage the data movement.

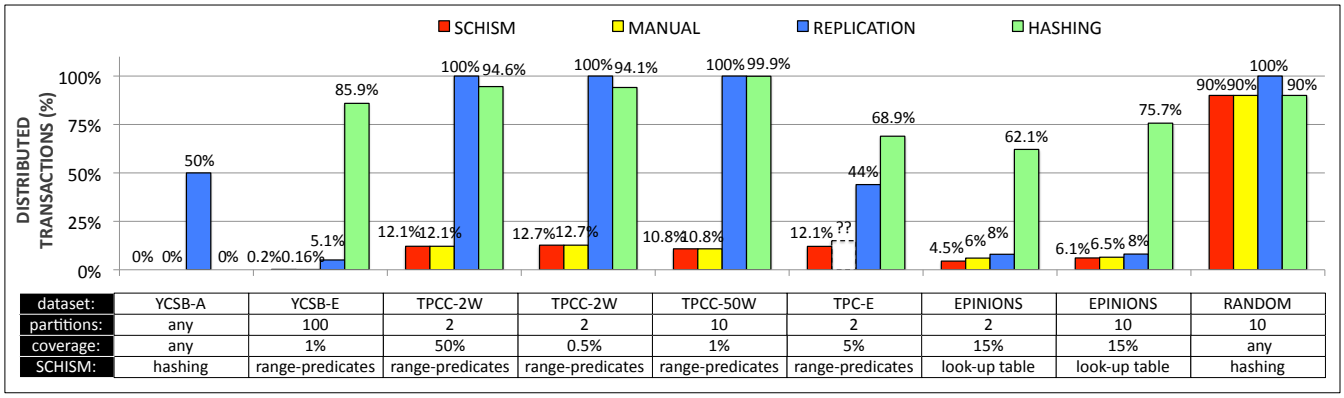
The second important issue is how the replication and partitioning derived by Schism is used at run-time to route queries. We have developed a router and distributed transaction coordinator that enforce the replication and partitioning strategy [5]. It supports hash partitioning, predicate-based partitioning, and lookup tables. The router is a middleware layer that parses the SQL statements and determines which partitions they need to access. For read-only queries over replicated tuples, Schism attempts to choose a replica on a partition that has already been accessed in the same transaction. This strategy reduces the number of distributed transactions. More details about the routing are in Appendix C.

6. EXPERIMENTAL EVALUATION

In this section, we present an experimental validation of Schism.

6.1 Partitioning Algorithm

In order to assess the quality of the partitioning produced by our algorithm we experimented on a variety of workloads derived from synthetic and real applications, described below. We compare the quality of the partitioning in terms of the number of distributed transactions. The results are summarized in Figure 4. The graph at the top compares the number of distributed transactions produced by Schism's graph partitioning algorithm to 1) the best manual partitioning we could devise (*manual*), 2) replication of all tables (*replication*), and 3) hash partitioning on the primary key or tuple id (*hashing*). The table at the bottom indicates the number of partitions used, the fraction of the dataset that was sampled, and the recommendation of Schism's final validation scheme.



dataset:	YCSB-A	YCSB-E	TPCC-2W	TPCC-2W	TPCC-50W	TPC-E	EPINIONS	EPINIONS	RANDOM
partitions:	any	100	2	2	10	2	2	10	10
coverage:	any	1%	50%	0.5%	1%	5%	15%	15%	any
SCHISM:	hashing	range-predicates	range-predicates	range-predicates	range-predicates	range-predicates	look-up table	look-up table	hashing

Figure 4: Schism database partitioning performance.

In the following sections, we describe each of the 9 experiments; all experiments are the result of collecting a large trace of transactions, separating them into training and testing sets, and then applying the sampling and workload reduction techniques described in Section 5. The datasets are discussed in details in Appendix D.

YCSB-A The Yahoo! Cloud Serving Benchmark (YCSB) is a suite of benchmarks designed to be simple yet representative of workloads seen inside Yahoo [4]. Workload A is composed of reads (50%) and updates (50%) of a single tuple selected from a 100k-tuple table randomly with a Zipfian distribution. We use a trace of 10k requests. This shows that Schism can handle an easy scenario where any partitioning will perform well. In particular, we wanted to show that the validation phase detects that simple hash-partitioning is preferable to the more complicated lookup tables and range partitioning, which it does.

YCSB-E [4], a workload of 10k transactions performing either a short scan with a uniform random length of 0-10 tuples (95%), or a one tuple update (5%). Tuples are generated using a Zipfian distribution from a 100k-tuple table. The scans in this workload make hash-partitioning ineffective. Schism's explanation phase is produces a range-partitioning that performs as well as manual partitioning (done via by careful inspection of the workload.)

TPC-C 2W, based on the popular write-intensive, OLTP workload (2 warehouses, 100K transactions in the training set), this experiment shows that Schism can derive a high-quality predicate-based partitioning. The partitioning produced by the tool distributes tuples based on warehouse id, and replicates the `item` table—the same strategy found by human experts [21].

TPC-C 2W, sampling, this experiment is also based on TPC-C, but we stress-test Schism's *robustness to sampling* by partitioning a graph created from only 3% of the tuples and a training set of only 20k transactions. We train the decision tree on a sample of at most 250 tuples per table. This workload accesses slightly less than 0.5% of the entire database's tuples, but Schism still finds the "ideal" partitioning/replication scheme discussed above[1].

TPC-C 50W, in this experiment we scale the size of TPC-C to 50 warehouses (over 25M tuples) to show how Schism scales with the size the database. We also increase the number of partitions to 10. Using a training set of 150K transactions and sampling 1% of the database tuples, we obtain the same partitioning: partition by warehouse id and replicate the `item` table. Both Schism and manual partitioning have fewer distributed transactions in this 50 warehouse/10 partition experiment than the 2 warehouse/2 partition experiment. This is because some TPC-C transactions access multiple warehouses (10.7% of the workload). When partitioning a 2 warehouse database into 2 partitions, every such transaction will access multiple partitions. However, in a 50 warehouse/10 partition configuration, there is a chance that all the warehouses will be co-located in one partition. Thus, this configuration has few multi-partition transactions. This is the largest workload we evaluated, and the total running time (extracting the graph, partitioning, validation, and explanation) was 11 m 38 s.

TPC-E, a read-intensive OLTP workload derived from the popular benchmark, tested with 1k customers using a 100k transaction training set. This experiment tests our system on a more complex OLTP workload. The range-predicate solution provided by the system is better than both baselines (hashing, and full replication); because this benchmark is complex (33 tables, 188 columns, 10 kinds of transactions), we were unable to derive a high quality manual partitioning with which to compare our results. However, the result of 12.1% distributed transactions appears to be quite good.

Epinions.com, 2 partitions, a dataset derived from the eponymous social website [15]. We synthetically generated a set of nine transactions to model the most common functionality of the Epinions.com website—details in Appendix D. To compare against a manual partitioning, we asked two MIT Masters students to derive the best manual partitioning they could find. They found that the multiple n-to-n relations in the schema (capturing user ratings of items and trust relationships between users), combined with multiple access patterns in the transactions made the task challenging. The manual strategy they found, after several hours of analysis of the DB and the workload, uses a mix of replication and partitioning, yielding 6% distributed transactions.

Given the same information, Schism derived a lookup-table-based partitioning that yields just 4.5% distributed transactions. This is possible because the tool is able to inspect the social graph at the tuple level and discover clusters of users and items that are frequently accessed together, but rarely accessed across clusters. Since this was a read-mostly workload, tuples not present in the initial lookup table (i.e., not touched by the training transactions) have been replicated across all partitions. Hash partitioning and range partitioning yield significantly worse results, and thus are not selected by Schism in the final validation phase.

Epinions.com, 10 partitions, the same dataset as the previous test, but increasing the number of partitions to 10. The results (6% distributed transactions) are better than both baseline strategies (75.7% and 8%) and manual partitioning (6.5%).

Random, in this experiment, as in the first experiment, we test the capability of the system to select a simple strategy when more

[1]The small difference between Schism and manual partitioning in the TPC-C experiment are due to sampling of the test-set.

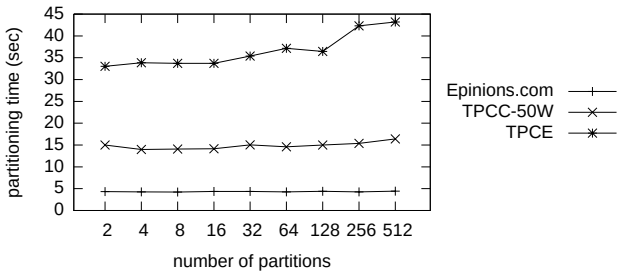


Figure 5: METIS graph partitioning scalability for growing number of partitions and graph size.

Table 1: Graph Sizes

Dataset	Tuples	Transactions	Nodes	Edges
Epinions	2.5M	100k	0.6M	5M
TPCC-50	25.0M	100k	2.5M	65M
TPC-E	2.0M	100k	3.0M	100M

complex solutions do not provide a better partitioning. The workload is composed of a single transaction that updates 2 tuples selected uniformly at random from a table of 1M tuples. In this scenario, no good partitioning exists, and lookup tables, range partitioning, and hash partitioning perform equally poorly. Full replication performs even worse since every transaction is an update. As a result the algorithm chooses to fall back to hash partitioning.

In summary, the Schism approach consistently matches or outperforms simple automatic strategies and manually-generated partitioning schemes on all of our workloads.

6.2 Scalability and Robustness

In order to show the scalability of our approach we tested (i) performance as the number of partitions increases, and (ii) performance as the size and complexity of the database grows.

All the stages of our system, with the exception of the graph partitioning phase, have performance that is essentially independent of the number of partitions, and performance that grows linearly with the size of the database and workload trace. Therefore, to show scalability, we focus on the performance of the graph partitioning phase. For this experiment we use the three graphs of Table 1.

In Figure 5 we show the running times of a serial implementation of the graph partitioning algorithm `kmetis` for an increasing number of partitions (see Appendix A for detailed experimental setup). The cost of partitioning increases slightly with the number of partitions. The size of the graph has a much stronger effect on the running time of the algorithm (approximately linear in the number of edges), justifying our efforts to limit the graph size via the heuristics presented in Section 5.1.

The most important heuristic for reducing the graph size is sampling. It is hard to provide a precise indication of the maximum degree of sampling that can be applied to a graph to still produce a good partitioning. However, we have experimental evidence that suggests that Schism produces good results even when presented with a small sample of the database. For example, for TPC-C with 2 warehouses we were able to sample 1% of the graph while still producing the same result as manual partitioning. A qualitative analysis of our experiments suggests that the minimum required graph size grows with workload complexity, the database size, and the number of partitions. Intuitively, a more complex workload requires a larger number of transactions (and thus edges) to be accurately modeled. A bigger database inherently requires more tuples in the graph. It also requires more edges, and therefore more transactions, so that there are sufficient relationships between tuples to

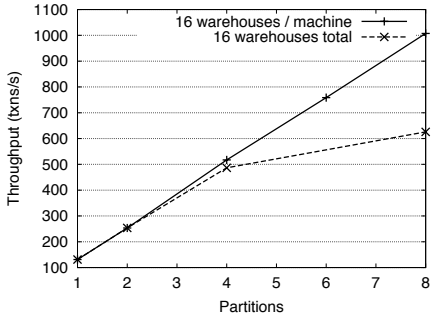


Figure 6: TPC-C Throughput Scaling

adequately capture how the data is accessed. Finally, a larger number of partitions requires a denser graph (more edges). Unfortunately, formalizing this into a quantitative model would require a more comprehensive set of examples, which is beyond the scope of this paper. A simple strategy to select the degree of sampling is to run the tool with increasing sample sizes until the partitioning quality stops increasing. For our simple examples, this seems to produce good results.

6.3 End-to-End Validation

In order to validate our end-to-end approach, and to showcase how Schism produces balanced partitions that maximize throughput, we ran a benchmark based on TPC-C with a cluster of 8 machines (see Appendix A for detailed configuration). We used Schism to divide the database into 1, 2, 4, and 8 partitions. Each partition was then assigned to a separate server. Schism was configured to balance the workload across partitions, which for TPC-C also yields partitions with nearly balanced data sizes. The range partitioning predicates produced are the same as discussed in the previous TPC-C experiments. We used two configurations. In the first, we spread the 16 warehouses across the cluster. This demonstrates scaling out a single application by adding more hardware. In the second, we added warehouses while adding machines, so each machine always had 16 partitions. This demonstrates using Schism to grow an application by adding more hardware. Sufficient TPC-C clients were used in order to saturate the throughput of the system. The throughput is shown in Figure 6.

The results show that a single server is able to achieve approximately 131 transactions per second with 16 warehouses. For the single 16 warehouse scale out configuration, the performance increases approximately linearly up to 4 machines, but with 8 machines it only obtains a 4.7× speedup. This is because the contention that is inherently part of the TPC-C implementation becomes a bottleneck when there are only 2 warehouses per machine. It is not possible to saturate a single machine because nearly all the transactions conflict, limiting the maximum throughput. The configuration which keeps a fixed 16 warehouses per machine avoids this bottleneck, so it shows a scalability that is very close to perfectly linear (7.7× for 8 machines = coefficient of 0.96).

This experiment shows that Schism is capable of producing a high-quality partitioning scheme that allows us to obtain good scalability. Our results suggest that if we used hash partitioning on this workload, we would see 99% distributed transactions, which would lead to a significant throughput penalty.

7. RELATED WORK

Partitioning in databases has been widely studied, for both single system servers (e.g. [1]) and shared-nothing systems (e.g. [23, 18]). These approaches typically use the schema to produce possi-

ble range or hash partitions, which are then evaluated using heuristics and cost models. They offer limited support for OLTP workloads, and are usually not capable of generating sensible partitioning scheme for schemas containing multiple n-to-n relations.

Tsangaris and Naughton's stochastic approach relies on graph partitioning heuristics for clustering in object-oriented databases [22]. However, our system also integrates replication and is designed for the different requirements of distributed databases.

Recently there has been significant interest in "simpler" distributed storage systems, such as BigTable [2] or PNUTS [3]. These systems repartition data continuously to balance data sizes and workload across multiple servers. Despite being more dynamic, these techniques do not handle OLTP transactions on multiple tables.

Scaling social network applications has been widely reported to be challenging due to the highly interconnected nature of the data. One Hop Replication is an approach to scale these applications by replicate the relationships, providing immediate access to all data items within "one hop" of a given record [16]. This approach relies on a partitioning algorithm to assign the initial partitions, which means that Schism could be used with this system.

The hybrid-range partitioning strategy (HRPS) [8] is a hybrid approach to hash- and range-partitioning based on query analysis. The scheme attempts to decluster (run in parallel on several nodes by spreading data across them) long-running queries and localize small range queries. HRPS applies only to single-table range queries over one dimension, and does not consider replication.

At the opposite end of the spectrum with respect to Schism there are pure declustering research efforts [19] and partitioning advisors often included in commercial DBMSs [7, 18] that mainly focus on OLAP workloads and single-system data-on-disk optimizations. Like us, Liu et al. [19] employ graph partitioning, but for the opposite purpose (declustering) and without consideration for replication or predicate-based explanation.

8. CONCLUSIONS

We presented Schism, a system for fine-grained partitioning of OLTP databases. Our approach views records in a database as nodes in a graph, with edges between nodes that are co-accessed by transactions. We then use graph-partitioning algorithms to find partitionings that minimize the number of distributed transactions. We further propose a number of heuristics, including sampling and record grouping, which reduce graph complexity and optimize performance. We also introduce a technique based on decision-tree classifiers to *explain* the partitioning in terms of compact set of predicates that indicate which partition a given tuple belongs to.

Our results show that Schism is highly practical, demonstrating (i) modest runtimes, within few minutes in all our tests, (ii) excellent partitioning performance, finding partitions of a variety of OLTP databases where only a few transactions are multi-partition, often matching or exceeding the best known manual partitionings, and (iii) ease of integration into existing databases, via support for range-based partitioning in parallel databases like DB2, or via our distributed transaction middleware.

9. ACKNOWLEDGEMENTS

This work was supported in part by Quanta Computer as a part of the T-Party Project.

10. REFERENCES

[1] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, 2004.

[2] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[3] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2), 2008.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. *SoCC*, 2010.

[5] C. Curino, E. Jones, Y. Zhang, E. Wu, and S. Madden. Relationalcloud: The case for a database service. *New England Database Summit*, 2010.

[6] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Comm. ACM*, 1992.

[7] R. Freeman. *Oracle Database 11g New Features*. McGraw-Hill, Inc., New York, NY, USA, 2008.

[8] S. Ghandeharizadeh and D. J. DeWitt. Hybrid-range partitioning strategy: a new declustering strategy for multiprocessor databases machines. In *VLDB*, 1990.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11, 2009.

[10] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1), 1998.

[11] G. Karypis and V. Kumar. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 5.0. `http://www.cs.umn.edu/~metis`, 2009.

[12] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal 49*, pages 291–307, 1970.

[13] R. Khandekar, S. Rao, and U. Vazirani. Graph partitioning using single commodity flows. *J. ACM*, 56(4):1–15, 2009.

[14] M. Koyutürk and C. Aykanat. Iterative-improvement-based declustering heuristics for multi-disk databases. *Journal of Information Systems*, 30(1):47–70, 2005.

[15] P. Massa and P. Avesani. Controversial users demand local trust metrics: an experimental study on epinions.com community. In *AAAI'05*, 2005.

[16] J. M. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez. Scaling online social networks without pains. NetDB, 2009.

[17] J. R. Quinlan. C4.5: Programs for machine learning. *Morgan Kaufmann Series in Machine Learning*, 1993.

[18] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, 2002.

[19] D. ren Liu and S. Shekhar. Partitioning similarity graphs: A framework for declustering problems. *ISJ*, 21, 1996.

[20] N. Selvakkumaran and G. Karypis. Multi.objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. In *ICCAD*, 2003.

[21] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007.

[22] M. M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. *SIGMOD Rec.*, 20(2), 1991.

[23] D. C. Zilio. Physical database design decision algorithms and concurrent reorganization for parallel database systems. In *PhD thesis*, 1998.

Table 2: Experimental Systems

Num. Machines	Environment	Description
8	CPU	2 × Intel Xeon 3.2 GHz
	Memory	2 GB
	Hard Disk	1 × 7200 RPM SATA (WD1600JS, 8.9 ms latency)
1	CPUs	2 × Quad-Core Intel Xeon E5520 2.26GHz
	Memory	24 GB
	Hard Disk	6 × 7200 RPM SAS (ST32000444SS, 5 ms latency) Hardware RAID 5 (Dell PERC6/i)
All	OS Distribution	Linux Ubuntu Server 9.10
	OS Kernel	Linux 2.6.31-19 server
	Java	Sun Java 1.6.0_20-b02
	DBMS	MySQL 5.4.3-beta

APPENDIX

A. HW/SW CONFIGURATION

The experiments in Section 3 and Section 6.3 were run on 8 servers running MySQL connected via a single Gigabit Ethernet switch. A more powerful machine was used to generate traffic, and for the run-time performance experiments in Section 6.2. Details about the hardware and software are shown in Table 2.

B. HYPERGRAPHS

Rather than representing transactions as a collection of edges between nodes, we explored representing them as a single hyperedge connecting the nodes accessed by the transaction. This formulation is somewhat more natural, because the cost of the cut hyperedges exactly matches the number of distributed transactions. Additionally, we expected that hypergraph partitioning would produce higher quality, based on previous results in the graph partitioning literature [20] and on previous papers in the database community that have used graph partitioners [14]. However, after an extensive set of tests with the most popular hypergraph partitioning libraries (hMETIS and Zoltan-PHG), we found that graph partitioning was far faster and produced better results. Our assumption is that graph partitioning is more widely used and studied, and thus the tools are more mature.

As a result, we needed to choose to approximate a hyperedge as a set of edges in the graph. This is a known hard task. After many tests, we decided to use a clique to represent transactions (more amenable to sampling) and a star-shaped representation for replication (fewer edges for tuples that are touched very frequently). This combination of representations provided good output and reasonable graph sizes.

C. PARTITIONING AND ROUTING

In this appendix, we discuss two query routing problems: i) how to handle lookup tables, ii) how to use the partitioning scheme (lookup tables, range predicates, hashing) to route queries/updates to the right machines.

C.1 Lookup Tables

As described in Section 4.2, lookup tables are useful when there is some locality in the data that is not obvious from the schema. In order to capture the best partitioning for this type of data, it is necessary to maintain partition information for each tuple individually. At the logical level this is a mapping between tuples and partition ids. Maintaining this mapping for all tuple attributes is infeasible. However, many applications access data primarily by specifying the primary key, which is typically a system-generated, dense set of integers. For this type of application, storing and maintaining lookup tables can be very efficient. This information can be treated as "soft state" because it can be easily recovered by a scanning the database in case of a crash.

At the physical level we experimented with three different implementations of lookup tables: traditional indexes, bit arrays and bloom filters. The most general is a traditional index, which can be used with any type of data in the partitioning column(s), and is efficient and well studied. However, indices require the most of the three solutions. Bit arrays work for (almost) dense integer keys, by using the key as an offset into an array of partition ids. This is a very compact and fast way to store lookup tables, which is a natural fit for the many applications that use integer autoincrement primary keys. With a few gigabytes of RAM, it is possible to store the partition ids for several billion tuples in an array.

Finally, we performed some preliminary testing with using bloom filters, which should provide a more compact representation, with the disadvantage that it will produce some false positives. These false positives decrease performance with no impact on correctness. When routing queries, a false positive means that a partition will be involved in executing a statement, even though it does not need to be. The actual space savings depend on many parameters, such as the number of tuples, the number of partitions, and the false positive rate.

We are currently investigating how to best implement lookup tables for distributed databases, to establish when they are a better choice than traditional hash or range partitioning, and which physical implementation works best for each scenario.

C.2 Routing Queries and Updates

Another important problem to be solved within the middleware layer is how to route a query or an update in order to guarantee the correct execution and minimize the participants involved in executing the statement.

Our system provides a JDBC interface for applications to submit statements. Given an input statement our router component performs the following steps: i) parses the statement, ii) extracts predicates on table attributes from the `WHERE` clause, iii) compares the attributes to the partitioning scheme to obtain a list of destination partitions. A distributed transaction coordinator then manages the transactional execution of the statement across multiple machines.

Statements that access tuples using the partitioning attribute(s) are sent only to the partition(s) that store the tuples. Statements that access a single table using other attributes are handled by broadcasting the statement to all partitions of the table, then taking the union of the result. More complex statements that access multiple tables using attributes that are not the partitioning attributes are not currently handled as they require communicating intermediate results in order to compute the join.

Partitioning is most efficient when most statements use the partitioning attributes in the `WHERE` clause. This is why Schism tries to use the most common `WHERE` clause attributes for the explanation phase.

D. DATASETS

In this section we provide some additional details about the benchmarks used in Section 6.

D.1 Yahoo! Cloud Serving Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) is a collection of simple micro-benchmarks designed to represent data management applications that are simple, but that require high scalability [4]. This benchmark is designed to evaluate distributed key/value storage systems, such as those created by Yahoo, Google and various open source projects. In this work, the benchmark is used to provide insight in some of the capabilities of our tool, e.g., its ability

fall back to cheaper partitioning strategies in case of a tie, on a standardized and simple test case.

From the five core YCSB workloads, we selected workload A and workload E. Workload A consists of a 50/50 mix of reads/writes on a single tuple chosen with a Zipfian distribution. Ignoring the potential challenges to achieve good performance with Internet-scale data, this is a very easy problem to partition. In fact, with the exception of full-replication, every partitioning strategy leads to zero distribution cost because transactions touch only one tuple. Hence, the goal of running this test is to show how our system is capable of selecting a cheaper partitioning strategy (e.g., hash partitioning) when one exists.

Workload E consists of a 95-5 mix of read and writes, where the reads perform a short scan (of length uniformly chosen from 1–100), and writes touch a single record. The starting point of the read scan and the write are chosen at random with Zipfian distribution. This workload shows that hashing fails for range queries, and that our tool can automatically chose range-predicate split points that produce a close-to-optimal partitioning strategy.

D.2 TPC-C

The TPC-C benchmark is designed to simulate the OLTP workload of an order processing system. The implementation we used does not strictly adhere to the requirements of the TPC-C specification. In particular, in order to generate high throughput with small dataset sizes, the client simulators do not use the specified "think times" and instead submits the next transaction as soon as it receives the response from the first. Thus, our results in Section 6.3 are not meant to be compared with other systems, but rather indicate the relative performance of the configurations we tested.

The TPC-C schema contains 9 tables with a total of 92 columns, 8 primary keys, and 9 foreign keys. TPC-C workload contains 5 types of transactions.

D.3 TPC-E

The TPC-E benchmark is an OLTP benchmark that is more complex than TPC-C. It models a brokerage firm that is handling trades on behalf of clients. Again, the implementation we use is not completely faithful to the specification, but it tries to generate the correct data and transactions. The TPC-E schema contains 33 tables with a total of 188 columns, 33 primary keys, 50 foreign keys, and 22 constraints. The workload contains 10 types of transactions.

D.4 Epinions.com

The Epinions.com experiment aims to challenge our system with a scenario that is difficult to partition. It verifies it effectiveness in discovering intrinsic correlations between data items that are not visible at the schema or query level. Our Epinions.com schema contains four relations: `users`, `items`, `reviews` and `trust`. The `reviews` relation represents an n-to-n relationship between users and items (capturing user reviews and ratings of items). The `trust` relation represents a n-to-n relationship between pairs of users indicating a unidirectional "trust" value. The data was obtained by Paolo Massa from the Epinions.com development team [15]. Since no workload is provided, we created requests that approximate the functionality of the website:

Q1. *For a given user and item, retrieve ratings from trusted users*

Q2. *Retrieve the list of users trusted by a given user*

Q3. *Given an item, retrieve the weighted average of all ratings*

Q4. *Given an item, obtain the 10 most popular reviews*

Q5. *List the 10 most popular reviews of a user*

Q6. *Insert/update the user profile*

Q7. *Insert/update the metadata of an item*

Q8. *Insert/update a review*

Q9. *Update the trust relation between two users*

In our comparison against manual partitioning, the above queries and the input database were provided to the students acting as database administrators in charge of partitioning of the database. Finding a good partitioning is non-trivial, since the queries involve multiple n-to-n relations with opposing requirements on how to group tables and tuples. For example, Q1 will access a single partition if the data is partitioned by item, and ratings and trust are stored with items, while Q2 will access a single partition if the data is partitioned by user, and trust is stored with users. The students' proposed solution was to optimize for the most frequent queries in the workload (Q1 and Q4) by partitioning item and review via the same hash function, and replicating users and trust on every node.

D.5 Random

This final synthetic dataset is designed to be an "impossible" to partition workload. Each request writes a pair of tuples selected uniformly at random from the entire table. Here again the goal is to show how, when a tie happens, the algorithm chooses the simplest and more robust partitioning strategy—hash partitioning, in this case.