Clay: Fine-Grained Adaptive Partitioning for General Database Schemas

Marco Serafini△, Rebecca Taft♦, Aaron J. Elmore♣,
Andrew Pavlo♠, Ashraf Aboulnaga△, Michael Stonebraker♦
△Qatar Computing Research Institute - HBKU, ♦Massachusetts Institute of Technology,
♣University of Chicago, ♠Carnegie Mellon University

mserafini@qf.org.qa, rytaft@mit.edu, aelmore@cs.uchicago.edu,
pavlo@cs.cmu.edu, aaboulnaga@qf.org.qa, stonebraker@csail.mit.edu

ABSTRACT

Transaction processing database management systems (DBMSs) are critical for today's data-intensive applications because they enable an organization to quickly ingest and query new information. Many of these applications exceed the capabilities of a single server, and thus their database has to be deployed in a distributed DBMS. The key factor affecting such a system's performance is how the database is partitioned. If the database is partitioned incorrectly, the number of distributed transactions can be high. These transactions have to synchronize their operations over the network, which is considerably slower and leads to poor performance. Previous work on elastic database repartitioning has focused on a certain class of applications whose database schema can be represented in a hierarchical tree structure. But many applications cannot be partitioned in this manner, and thus are subject to distributed transactions that impede their performance and scalability.

In this paper, we present a new on-line partitioning approach, called Clay, that supports both tree-based schemas and more complex "general" schemas with arbitrary foreign key relationships. Clay dynamically creates blocks of tuples to migrate among servers during repartitioning, placing no constraints on the schema but taking care to balance load and reduce the amount of data migrated. Clay achieves this goal by including in each block a set of hot tuples and other tuples co-accessed with these hot tuples. To evaluate our approach, we integrate Clay in a distributed, main-memory DBMS and show that it can generate partitioning schemes that enable the system to achieve up to 15× better throughput and 99% lower latency than existing approaches.

1. INTRODUCTION

Shared-nothing, distributed DBMSs are the core component for modern on-line transaction processing (OLTP) applications in many diverse domains. These systems partition the database across multiple nodes (i.e., servers) and route transactions to the appropriate nodes based on the data that these transactions touch. The key to achieving good performance is to use a partitioning scheme (i.e., a mapping of tuples to nodes) that (1) balances load and (2) avoids

expensive multi-node transactions [5, 23]. Since the load on the DBMS fluctuates, it is desirable to have an *elastic* system that automatically changes the database's partitioning and number of nodes dynamically depending on load intensity and without having to stop the system.

The ability to change the partitioning scheme without disrupting the database is important because OLTP systems incur fluctuating loads. Additionally, many workloads are seasonal or diurnal, while other applications are subject to dynamic fluctuations in their workload. For example, the trading volume on the NYSE is an order of magnitude higher at the beginning and end of the trading day, and transaction volume spikes when there is relevant breaking news. Further complicating this problem is the presence of *hotspots* that can change over time. These occur because the access pattern of transactions in the application's workload is skewed such that a small portion of the database receives most of the activity. For example, half of the NYSE trades are on just 1% of the securities.

One could deal with these fluctuations by provisioning for expected peak load. But this requires deploying a cluster that is over-provisioned by at least an order of magnitude [27]. Furthermore, if the performance bottleneck is due to distributed transactions causing nodes to wait for other nodes, then adding servers will be of little or no benefit. Thus, over-provisioning is not a good alternative to effective on-line reconfiguration.

Previous work has developed techniques to automate DBMS reconfiguration for unpredictable OLTP workloads. For example, Accordion [26], ElasTras [6], and E-Store [28] all study this problem. These systems assume that the database is partitioned a priori into a set of *static blocks*, and all tuples of a block are moved together at once. This does not work well if transactions access tuples in multiple blocks and these blocks are not colocated on the same server. One study showed that a DBMS's throughput drops by half from its peak performance with only 10% of transactions distributed [23]. This implies that minimizing distributed transactions is just as important as balancing load when finding an optimal partitioning plan. To achieve this goal, blocks should be defined dynamically so that tuples that are frequently accessed together are grouped in the same block; co-accesses within a block never generate distributed transactions, regardless of where blocks are placed.

Another problem with the prior approaches is that they only work for tree schemas. This excludes many applications with schemas that cannot be transposed into a tree and where defining static blocks is impossible. For example, consider the Products-Parts-Suppliers schema shown in Figure 1. This schema contains three tables that have many-to-many relationships between them. A product uses many parts, and a supplier sells many parts. If we apply prior approaches and assume that either Products or Suppliers is the root
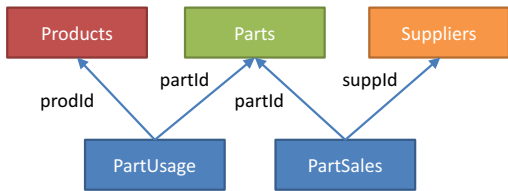
Figure 1: Products-Parts-Suppliers database schema. Arrows represent child-parent foreign key relationships.

of a tree, we get an inferior data placement. If we assume Products is the root, then we will colocate Parts and Suppliers tuples with their corresponding Products tuples. But this is also bad because Parts are shared across multiple Products, and Suppliers may supply many Parts. Hence, there is no good partitioning scheme that can be identified by solely looking at the database schema, and a more general approach is required for such "bird's nest" schemas.

There is also previous work on off-line database partitioning for general (i.e., non-tree) schemas with the goal of minimizing distributed transactions. Schism is a prominent representative of this line of work [5]. The basic idea is to model the database as a graph where each vertex represents a tuple, and an edge connects two vertices if their tuples are accessed together in a transaction. An edge's weight corresponds to the number of transactions accessing the two tuples together. Partitions are defined using a *MinCut* algorithm to split the graph in a way that minimizes the weight on inter-partition edges (such edges represent distributed transactions). Schism is an off-line approach, which means that it needs to be re-run each time a reconfiguration is required. As we will explain later, the dual goals of balancing load and minimizing distributed transactions are difficult to express in a MinCut problem formulation. Furthermore, Schism does not take into account the current database configuration, and thus it cannot minimize data movement.

To overcome all of the above limitations, we present **Clay**, an elasticity algorithm that makes no assumptions about the schema, and is able to simultaneously balance load and minimize distributed transactions. Unlike existing work on on-line reconfiguration, which migrates tuples in static blocks, Clay uses dynamic blocks, called *clumps*, that are created on-the-fly by monitoring the workload when a reconfiguration is required. The formation of a clump starts from a hot tuple that the DBMS wants to migrate away from an overloaded partition. After identifying such a hot tuple, Clay enlarges the clump around that tuple by adding its frequently co-accessed tuples. This avoids generating a large number of distributed transactions when moving the clump. Another advantage of Clay is that it is incremental, thereby minimizing the cost of data migration. In our experiments, Clay outperforms another on-line approach based on the Metis graph partitioning algorithm [16] by 1.7–15× in terms of throughput and reduces latency by 41–99%. Overall, the performance of Clay depends on how skewed the workload is: the higher the skew, the better the gain to be expected by using Clay.

2. OVERVIEW

We first illustrate the main idea of our clump migration technique using the example of the Products-Parts-Suppliers database from Figure 1. For simplicity, we examine an instance running on three servers, each with one partition. Assume that partition P3 becomes overloaded because it hosts too many hot tuples. When overload is detected, Clay monitors all transactions executed in the system for a few seconds. Based on this sample, it builds a *heat graph* like the one depicted in Figure 2, where vertices are tuples and edges represent co-accesses among tuples. The heat graph includes only
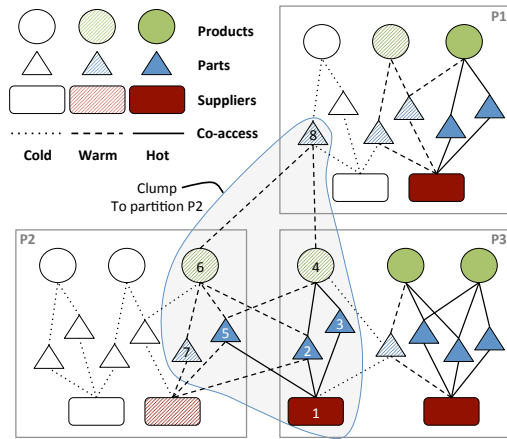

Figure 2: Heat graph example for a Products-Parts-Suppliers database in partitions P1, P2, and P3. For simplicity, we only consider three degrees of hotness. P3 is initially overloaded. Clay creates a clump and moves it to P2. Vertex IDs indicate the order that Clay adds them to the clump.

the tuples whose activity has been observed during the monitoring interval. Some tuples and edges may be hotter than others. This is modeled using vertex and edge weights.

Clay builds clumps based on the heat graph. Initially, it creates a clump consisting of the hottest tuple of the most overloaded partition – the Suppliers tuple corresponding to vertex #1 in Figure 2. It then evaluates the effect of moving the clump to another partition. To minimize distributed transactions, Clay looks for the partition whose tuples are most frequently accessed with the tuples in the clump – partition P2 in our example. The move (only vertex #1 at this point) generates too many distributed transactions because there is a large number of edges between partitions P2 and P3. As a result, P2 would become overloaded, and P3 would have no benefit from the move due to an increased number of distributed transactions. Therefore, Clay extends the clump with the vertex that is most frequently co-accessed with the clump, which is vertex #2 in the example. The process repeats, and the clump is extended with vertices #3–8. Note that vertices #4 and #6–8 are not co-accessed with the initial tuple, but are still added to the clump due to the transitivity of the co-access relation. Note also that vertices #5–8 reside on a different partition from the initial tuple. Clay ignores the current partitioning when building a clump, focusing exclusively on the co-access patterns and adding affine tuples from any partition. The process continues until Clay finds a clump that can be moved to a partition without overloading it. If the clump cannot be extended anymore or it reaches a maximum size, Clay scales out the system by adding a new partition and restarts the clump-finding process.

To build the heat graph, it is necessary to collect detailed information about co-accesses among tuples in the same transaction. Clay performs this on-line monitoring efficiently and only for a short interval of time (~20 seconds). Although the heat graph can become large, with up to billions of vertices and edges in our experiments, it is still small enough to fit in main memory; our reconfiguration algorithm always used less than 4 GB.

3. RELATED WORK

A significant amount of research exists on partitioning strategies for analytic workloads (OLAP), typically balancing locality with declustering data to maximize parallelism [20, 33]. Some of that work explicitly considers on-line partitioning algorithms for analytics [14] or large graphs [32]. We limit our discussion to partition-

ing of OLTP databases, since the goals and techniques are different from partitioning for OLAP applications. Most notably, these approaches do not combine scaling to tuple-level granularity, mixing load-balancing with minimizing cross partition transactions, and building incremental solutions to update the partitioning.

As discussed earlier, Schism [5] is an off-line algorithm that analyzes a transaction log and has no performance monitoring or live reconfiguration components. It builds an access graph similar to our heat graph and uses Metis [16] to find a partitioning that minimizes the edge cuts. But since Metis cannot support large graphs, the DBA must pre-process the traces by sampling transactions and tuples, filtering by access frequency, and aggregating tuples that are always accessed together in a single vertex. Since keeping an explicit mapping of every tuple to a partition would result in a huge routing table, Schism creates a decision tree that simplifies the individual mapping of tuples into a set of range partitions. Finally, in the final validation step, Schism compares different solutions obtained in the previous steps and selects the one having the lowest rate of distributed transactions. Clay's clump migration heuristic is incremental, so it minimizes data migration, and it outperforms Metis when applied to the heat graph. In addition, Clay's two-tiered routing creates small sets of hot tuples that minimize the size of the routing tables, so it does not require Schism's extra steps.

Sword [25] is another off-line partitioning tool that models the database as a hypergraph and uses an incremental heuristic to approximate constrained n-way graph partitioning. It uses a one-tier routing scheme that divides the database into coarse-grained chunks. Sword performs incremental partitioning adjustments by periodically evaluating the effect of swapping pairs of chunks. Our experiments show that Clay outperforms state-of-the-art algorithms that compute constrained n-way graph partitioning from scratch. Furthermore, Clay adopts a two-tiered approach that supports fine-grained mapping for single tuples.

Like Schism and Sword, JECB [30] provides a partitioning strategy to handle complex schemas, but the focus is on scalable partitioning for large clusters. JECB examines a workload, database schema, and source code to derive a new partitioning plan using a divide-and-conquer strategy. The work does not explicitly consider hot and cold partitions (or tuples) that arise from workload skew.

PLP is a technique to address partitioning in a single-server, shared-memory system to minimize bottlenecks that arise from contention [22]. The approach recursively splits a tree amongst dedicated executors. PLP focuses on workload skew, and does not explicitly consider co-accesses between tuples or scaling out across multiple machines. ATraPos improves on PLP by minimizing accesses to centralized data structures [24]. It considers a certain number of sub-partitions (similar to algorithms using static blocks) and assigns them to processor sockets in a way that balances load and minimizes the inter-process synchronization overhead.

None of the aforementioned papers discuss elasticity (i.e., adding and removing nodes), but there are several systems that enable elastic scaling through limiting the scope of transactions. Megastore [2] uses entity groups to identify a set of tuples that are semantically related, and limit multi-object transactions to within the group. Others have presented a technique to identify entity groups given a schema and workload trace [17]. This approach is similar to Clay in that it greedily builds sets of related items, but it focuses on breaking a schema into groups, and load-balancing and tuple-to-partition mapping are not factors in the grouping. Similarly, Horticulture [23] identifies the ideal attributes to partition each table but does not address related tuple placement. Beyond small entity groups, ElasTras [6], NuoDB [21], and Microsoft's cloud-based SQL Server [3] achieve elastic scaling on complex structures by limiting transactions to a single partition. Although ElasTras does support elastic scaling, the system does not specify how to split and merge partitions to balance workload skew and tuple affinity. Many key-value stores support intelligent data placement for load-balancing and elastic scaling [31, 11, 9], but provide weaker transaction guarantees than a relational DBMS.

Accordion [26] provides coarse-grained elastic partitioning: the database is pre-partitioned into a relatively small number of data *chunks* (or virtual partitions), each potentially comprising a large number of tuples. The limitation on the number of chunks is given by Accordion's Mixed Integer Linear Programming (MILP) solver to find an optimal plan. The problem with a coarse-grained approach is that it cannot deal with skewed workloads where multiple hot tuples may be concentrated in one data chunk [28]. Accordion learns the shape of the capacity function for each configuration. With few chunks there are only relatively few configurations, but if we consider each tuple as a potential chunk, then it becomes impossible to build an accurate capacity model for every configuration.

Coarse-grained approaches have problems with skewed workloads where multiple hot tuples can end up in the same chunk. E-Store [28] supports tree-schemas by using a two-tiered approach for load-balancing. It uses fine-grained partitioning for a small number of hot tuples, and a coarse-grained partitioning for the rest of the database. Targeting hot tuples in this manner allows the system to identify hot spots, but it has limitations. Consider the case where two hot tuples are frequently accessed together in a transaction. E-Store ignores such co-accesses, so it can independently place hot tuples on different servers, thereby generating a large number of distributed transactions. To avoid this problem, E-Store must assume that the database schema is tree-structured and every transaction accesses only the tree of one root tuple. Hence, a root tuple and its descendants are moved as a unit. Lastly, E-store fails to address co-accesses to hot dependent tuples.

Online workload monitoring has been used to deal with hot keys also in stream processing systems [18, 19] and in general sharding systems like Slicer [1]. However, these systems have no notion of transactions or co-accesses.

4. PROBLEM STATEMENT

We now define the data placement problem that Clay seeks to solve. A database consists of a set of tables $\langle T_1 \ldots T_t \rangle$. Each table T_i has one or more *partitioning attributes* $\langle A_1^i, \ldots, A_h^i \rangle$, which are a subset of the total set of attributes of T_i. Tuples are horizontally partitioned across a set of servers $s_1, \ldots, s_j$. All tuples of table T_i with the same values of their partitioning attributes $\langle A_1^i = x_1, \ldots, A_h^i = x_h \rangle$ are placed on the same server and are modeled as a *vertex*. The database sample is represented as a set of vertices $V = \{v_1, \ldots, v_n\}$, where each vertex v has a weight $w(v)$ denoting how frequently the vertex is accessed. Co-accesses between two vertices are modeled as an edge, whose weight denotes the frequency of the co-accesses. We call the resulting graph $G(V, E)$ the *heat graph* having vertices in V and edges in E.

Data placement is driven by a *partitioning plan* $P : V \to \Pi$ that maps each vertex to a partition in Π based on the value of its partitioning attributes. A single partition can correspond to a server, or multiple partitions can be statically mapped onto a single server.

4.1 Incremental Data Placement Problem

Clay solves an *incremental data placement problem* that can be formulated as follows. The system starts from an initial plan P. Let $L_P(p)$ be the load of partition p in the plan P, let ϵ be the percentage of load imbalance that we allow in the system, and let θ be the average load across all partitions in the plan P multiplied

by $1 + \epsilon$. Let P be the current partitioning plan, P' be the next partitioning plan identified by Clay, and $\Delta(P, P')$ be the number of vertices mapped to a different partition in P and P'. Given this, the system seeks to minimize the following objective function:

$$\text{minimize} \quad |P'|, \Delta(P, P') \qquad (1)$$
$$s.t. \qquad \forall p \in \Pi : \; L_{P'}(p) < \theta$$

We specify the two objectives in order of priority. First, we minimize the number of partitions in P'. Second, we minimize the amount of data movement among the solutions with the same number of partitions. In either case, we limit the load imbalance to be at most ϵ. We define w as the weight of a vertex or edge, E as the set of edges in the heat graph, and $k > 0$ as a constant that indicates the cost of multi-partition tuple accesses, which require additional coordination. Given this, the load of a partition $p \in \Pi$ in a partitioning plan P is expressed as follows:

$$L_P(p) = \sum_{\substack{v \in V \\ P(v)=p}} w(v) + \sum_{\substack{v \in V \\ u \in V \\ P(v)=p \\ \langle v,u \rangle \in E \\ P(u) \neq p}} w(\langle v, u \rangle) \cdot k \qquad (2)$$

The parameter k indicates how much to prioritize solutions that minimize distributed transactions over ones that balance tuple accesses. Increasing k gives greater weight to the number of distributed transactions in the determination of the load of a partition.

4.2 Comparison with Graph Partitioning

We now revisit the issue of comparing Clay with graph partitioning techniques, and in particular to the common variant solved by Metis [16] in Schism [5]. The incremental data placement problem is different from constrained n-way graph partitioning on the heat graph, where n is the number of database partitions. The first distinction is incrementality, since a graph partitioner ignores the previous plan P and produces a new plan P' from scratch. By computing a new P', the DBMS may have to shuffle data to transition from P to P', which will degrade its performance. We contend, however, that the difference is not limited to incrementality.

Graph partitioning produces a plan P' that minimizes the number of edges across partitions under the constraint of a maximum load imbalance among partitions. The load of a partition is expressed as the sum of the weights of the vertices in the partition:

$$\hat{L}_P(p) = \sum_{\substack{v \in V \\ P(v)=p}} w(v) \qquad (3)$$

To be more precise, consider the Metis graph partitioner that solves the following problem:

$$\text{minimize} \quad |\{\langle v, u \rangle \in E : \; P'(v) \neq P'(u)\}| \qquad (4)$$
$$s.t. \qquad \forall p, q \in \Pi : \; \hat{L}_{P'}(p)/\hat{L}_{P'}(q) < 1 + \eta$$

where η is an *imbalance constraint* provided by the user.

The load balancing constraint is over a load function, $\hat{L}_P$, which does not take into account the cost of distributed transactions. In graph terms, the function does not take into account the load caused by cross-partition edges. This is in contrast with the definitions of Equations 1 and 2, where the load threshold considers at the same time both local and remote tuple accesses and their respective cost.

The formulation of Equations 1 and 2 has two advantages over Equations 3 and 4. The first is that the former constrains the number of cross-partition edges *per partition*, whereas Equation 4 minimizes the *total* number of cross-partition edges. Therefore, Equation 4 could create a "star" partitioning setting where all cross-partition edges, and thus all distributed transactions, are incident on a single server, causing that server to be highly overloaded.
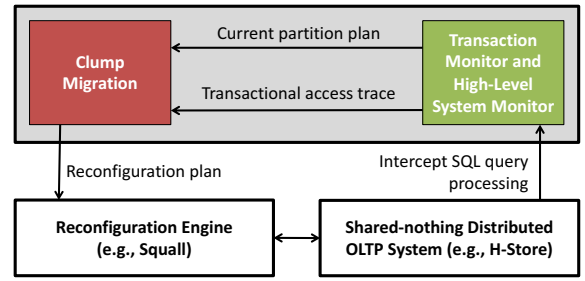

Figure 3: The Clay framework.

The second advantage of our formulation is that it combines the two conflicting goals of reaching balanced tuple accesses and minimizing distributed transactions using a single load function. Considering the two goals as separate makes it difficult to find a good level of η, as our experiments show. In fact, if the threshold is too low, Metis creates a balanced load in terms of single partition transactions, but it also causes many distributed transactions. If the threshold is too large, Metis causes fewer distributed transactions, but then the load is not necessarily balanced.

One could consider using L_P instead of $\hat{L}_P$ as the expression of load in Equation 4. Unfortunately, this is not possible since vertex weights need to be provided as an input to the graph partitioner, whereas the number of cross-partition edges depends on the outcome of the partitioning itself.

5. SYSTEM ARCHITECTURE

Clay runs on top of a distributed OLTP DBMS and a reconfiguration engine that can dynamically change the data layout of the database (see Figure 3). The monitoring component of Clay is activated whenever performance objectives are not met (e.g., when the latency of the system does not meet an SLA). If these conditions occur, Clay starts a transaction monitor that collects detailed workload monitoring information (Section 6). This information is sent to a centralized reconfiguration controller that builds a heat graph. The controller then runs our migration algorithm that builds clumps on the fly and determines how to migrate them (Section 7).

Although Clay's mechanisms are generic, the implementation that we use in our evaluation is based on the H-Store system [12, 15]. H-Store is a distributed, in-memory DBMS that is optimized for OLTP workloads and assumes that most transactions are short-lived and datasets are easily partitioned. The original H-Store design supports a static configuration where the set of partitions and hosts and the mapping between tuples and partitions are all fixed. The E-Store [28] system relaxes some of these restrictions by allowing for a dynamic number of partitions and nodes. E-Store also changes how tuples are mapped to partitions by using a two-tiered partitioning scheme that uses fine-grained partitioning (e.g., range partitioning) for a set of "hot" tuples and then a simple scheme (e.g., range partitioning of large chunks or hash partitioning) for large blocks of "cold" tuples. Clay uses this same two-tier partitioning scheme in H-Store. It also uses Squall for reconfiguration [8], although its techniques are agnostic to it.

6. TRANSACTION MONITORING

The data placement problem of Section 4 models a database as a weighted graph. The monitoring component collects the necessary information to build the graph: it counts the number of accesses to tuples (vertices) and the co-accesses (edges) among tuples.

Monitoring tracks tuple accesses by hooking onto the transaction routing module. When processing a transaction, H-Store breaks SQL statements into smaller fragments that execute low-level operations. It then routes these fragments to one or more partitions based on the values of the partitioning attributes of the tuples that are accessed by the fragment.

Clay performs monitoring by adding hooks in the DBMS's query processing components that extract the values of the partitioning attributes used to route the fragments. These values correspond to specific vertices of the graph, as discussed in Section 4. The monitoring component is executed by each server and writes tuple accesses onto a monitoring file using the format $\langle tid, T, x_1, \dots, x_h \rangle$, where tid is a unique id associated with the transactions performing the access, T is the table containing the accessed tuple, h is the number of partitioning attributes of table T, and x_i is the value of the ith partitioning attribute in the accessed tuple. When a transaction is completed, monitoring adds an entry $\langle END, tid \rangle$.

Query-level monitoring captures more detailed information than related approaches. It is able to determine not only which tuples are accessed, but also which tuples are accessed together by the same transaction. E-Store restricted monitoring to root tuples because of the high cost of using low-level operations to track access patterns for single tuples [28]. Our evaluation shows that Clay's monitoring is more accurate and has low overhead.

Clay disables monitoring during normal transaction execution and only turns it on when some application-specific objectives are violated (e.g., if the 99th percentile latency exceeds a pre-defined target). After being turned on, monitoring remains active for a short time. Our experiments established that 20 seconds was sufficient to detect frequently-accessed hot tuples.

Once a server terminates monitoring, it sends the collected data to a centralized controller that builds the heat graph (see Figure 2) and computes the new plan. For every access to a tuple/vertex v found in the monitoring data, the controller increments v's weight by one divided by the length of the monitoring interval for the server, to reflect the rate of transaction accesses. Vertices accessed by the same transactions are connected by an edge whose weight is computed similarly to a vertex weight.

7. CLUMP MIGRATION

The clump migration algorithm is the central component of Clay. It takes as input the current partitioning plan P, which maps tuples to partitions, and the heat graph G produced by the monitoring component. Its output is a new partitioning plan (see Section 4). We now describe the algorithm more in detail.

7.1 Dealing with Overloaded Partitions

The clump migration algorithm of Clay starts by identifying the set of "overloaded" partitions that have a load higher than a threshold θ. The load per partition is defined by the formula in Equation 2 (we used a value of $k = 50$ in all our experiments since we found that distributed transactions impact performance much more than local tuple accesses). For each overloaded partition P_o, the migration algorithm dynamically defines and migrates clumps until the load of P_o is below the θ threshold (see Algorithm 1). A clump created to offload a partition P_o will contain some tuples of P_o but it can also contain tuples from other partitions. A *move* is a pair consisting of a clump and a destination partition.

Initializing a clump. The algorithm starts with an empty clump M. It then sets M to be the hottest vertex h in the *hot tuples list* $H(P_o)$, which contains the most frequently accessed vertices for

---

Algorithm 1: Migration algorithm to offload partition P_o

> look-ahead $\leftarrow A$;
> **while** $L(P_o) > \theta$ **do**
>   **if** $M = \emptyset$ **then**
>   // initialize the clump
> $h \leftarrow$ next hot tuple in $H(P_o)$;
> $M \leftarrow \{h\}$;
> $d \leftarrow$ *initial-partition*(M);
>   **else if** *some vertex in M has neighbors* **then**
>   // expand the clump
> $M \leftarrow M \cup$ *most-co-accessed-neighbor*(M, G);
> $d \leftarrow$ *update-dest*(M, d);
>   **else**
>   // cannot expand the clump anymore
>   **if** $C \neq \emptyset$ **then**
>   move $C.M$ to $C.d$;
> $M \leftarrow \emptyset$;
>   look-ahead $\leftarrow A$;
>   **else**
>   add a new server and restart the algorithm;
>   // examine the new clump
>   **if** *feasible*(M, d) **then**
> $C.M \leftarrow M$;
> $C.d \leftarrow d$;
>   **else if** $C \neq \emptyset$ **then**
>   look-ahead $\leftarrow$ look-ahead -1;
>   **if** look-ahead $= 0$ **then**
>   move $C.M$ to $C.d$;
> $M \leftarrow \emptyset$;
>   look-ahead $\leftarrow A$;

Algorithm 2: Finding the best destination for M

> **function** update-dest(M, d)
>   **if** $\neg$ *feasible*(M, d) **then**
> $a \leftarrow$ partition most frequently accessed with M;
>   **if** $a \neq d \wedge$ *feasible*(M, a) **then**
>   **return** a;
> $l \leftarrow$ least loaded partition;
>   **if** $(l \neq d) \wedge (\Delta_r(M, a) < \Delta_r(M, l)) \wedge$ *feasible*(M, l)
>   **then**
>   **return** l
>   **return** d;

---

each partition (i.e., those having the highest weight, in descending order of access frequency).

The algorithm then picks the destination partition that minimizes the overall load of the system. The function *initial-partition* selects the destination partition d having the lowest *receiver delta* $\Delta_r(M, d)$, where $\Delta_r(M, d)$ is defined as the load of d after receiving M minus the load of d before receiving M. Given the way the load function is defined (see Equation 2), the partition with the lowest receiver delta is the one whose tuples are most frequently co-accessed with the tuples in M, so moving M to that partition minimizes the number of distributed transactions. The initial selection of d prioritizes partitions that do not become overloaded after the move, if available. Among partitions with the same receiver delta, the heuristic selects the one with the lowest overall load. In systems like H-Store that run multiple partitions on the same physical server, the cost function assigns a lower cost to transactions that access partitions in the same server than to distributed transactions.

Expanding a clump. If M is not empty, it is extended with the neighboring tuple of M that is most frequently co-accessed with a tuple in M. This is found by iterating over all the neighbors of

vertices of M in G and selecting the one with the highest incoming edge weight. A clump can be extended with a tuple t located in a partition p different from the overloaded partition P_o. In this case, it is important to verify that P_o does not become overloaded when t is transferred to another partition. The algorithm guarantees this by checking that the *sender delta* $\Delta_s(M, p)$ (i.e., the difference between the load of p after M is moved to another partition and before the move) is non-positive.

The best destination partition of a clump can change after we add new tuples to the clump. Verifying which other partitions can take a newly extended clump, however, entails computing the receiver delta for many potential destination partitions, which is expensive. Therefore, the *update-dest* function shown in Algorithm 2 does not consider changing the current destination partition for the clump if the move of M to d is *feasible*, which means that either d does not become overloaded after the move, or the receiver delta of d is non-positive. In the latter case, d actually gains from receiving M, so the move is allowed even if d is overloaded. Formally, the feasibility predicate is defined as follows:

$$feasible(M, d) = \big(L(d) + \Delta_r(M, d) < \theta\big) \vee \big(\Delta_r(M, d) \le 0\big)$$

If the current move is not feasible, *update-dest* tries to update the destination by first considering the partition having the hottest edges connecting to the clump, and then the least loaded partition.

Note that sometimes a clump cannot be expanded anymore because there are no more neighbors or because its size has reached an upper limit. We will discuss this case shortly.

Moving a clump. After expanding a clump, the algorithm checks if the move to d is feasible. Clay calls the best feasible move it has found so far the *candidate* move C. A candidate move is not immediately reflected on the output plan because it may still be sub-optimal. For example, it could still generate many multi-partition transactions. Clay keeps expanding C for a certain number of steps in search for a (local) optimum, updating C every time it finds a better move. If no better move than C is found after A steps, Clay concludes that C is a local optimum and it modifies the output plan according to C. Larger values of A increase the likelihood of finding a better optimum, but they also increase the running time of the algorithm. We found $A = 5$ to be a good setting in our evaluation.

Candidate moves are also applied when a clump cannot be expanded anymore due to lack of neighbors. In this case, if a candidate move exists, we modify the plan according to it. If not, we add a new server and we restart the search for an optimal clump from the latest hot tuple that has not yet been moved, this time considering the new server as an additional destination option.

Example. Consider the example shown in Figure 2. The partition that we want to offload is P3. Initially, the clump consists of the hottest tuple in P3, which is the supplier tuple #1. The algorithm selects P2 as the destination partition, since tuple #1 is frequently accessed with the part tuple #5. However, moving tuple #1 to P2 would create new distributed transactions associated with the new cross-partition edges: two hot edges (from #1 to #2 and #3) and a cold edge. Assuming that these additional distributed transactions would make P2 overloaded, then this move is not feasible.

Since we do not have a feasible move yet, the algorithm expands the clump with the vertices that are most frequently co-accessed with tuple #1. The parts tuples #2 and #3 have hot edges connecting them to tuple #1, so they are added to the clump, but assume that this still does not make the move to P2 feasible due to the hot cross-edge connecting tuples #2 and #3 to the product tuple #4. Finally, tuple #4 is added (note that it is not directly accessed together with the original hot tuple #1) and let us assume that the
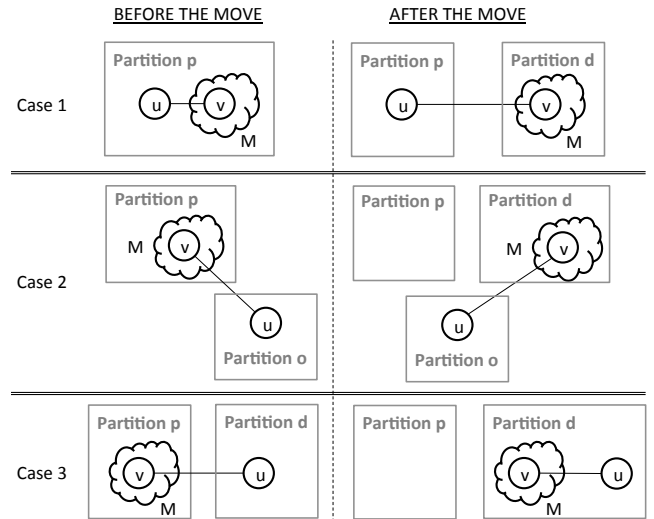

Figure 4: Types of moves when computing sender/receiver deltas.

move to P2 is now feasible. The algorithm then registers the pair $\langle\{\#1, \#2, \#3, \#4\}, \text{P2}\rangle$ as a candidate move. The move relieves P3 of its hotspot, but it still creates a few new distributed transactions for P3 due to the warm edge to the part tuple #8 in partition P1.

The algorithm keeps expanding the clump to find a better move. After adding a few tuples in P2, the clump eventually incorporates the part tuple #8, as illustrated in Figure 2. This clump has a lower receiver delta for P2 than the previous candidate since it eliminates the warm edge to tuple #8. Therefore, the new candidate move is set to transfer the new clump to the destination partition P2. If after doing A further expansions no better clump can be found, the candidate move is enacted and the clump is moved to P2.

7.2 Estimating Deltas Efficiently

The clump migration algorithm repeatedly computes the sender and receiver deltas of the moves it evaluates. The simplest way to compute these deltas it is to compare the load of the sender or receiver partition before and after the move. Computing this is expensive since it requires examining all of a partition's vertices and incident edges, both of which can be in the millions. We now discuss a more efficient, incremental way to calculate deltas that only has to iterate through the vertices of the clump and their edges.

Let us start from the sender delta $\Delta_s(M, p)$, where M is the clump to be moved and p is the sender partition. Moving M can result in additional distributed transactions for p. This occurs if we have two neighboring vertices u and v such that $v \in M$ and $u \notin M$ (see Figure 4, Case 1). After the move, the edge $\langle v, u\rangle$ results in new distributed transactions for p. The move can also eliminate distributed transactions for p. This occurs when a neighbor u of v is located in a different partition o (see Figure 4, Case 2). In this case, the sender saves the cost of some distributed transactions it was executing before the reconfiguration. Note that the same holds if o is equal to the destination partition d or if u is in M.

We can summarize these aforementioned cases in the following expression for the sender delta:

$$\Delta_s(M, p) = - \sum_{\substack{v \in M \\ P(v)=p}} w(v) + \sum_{\substack{v \in M \\ \langle v, u\rangle \in E \\ u \notin M \\ P(v)=p \\ P(u)=p}} w(\langle v, u\rangle) \cdot k - \sum_{\substack{v \in M \\ \langle v, u\rangle \in E \\ P(v)=p \\ P(u)\ne p}} w(\langle v, u\rangle) \cdot k$$

where P is the current plan. $P(v)$ represents the partition that tuple v is assigned to by the current plan that is being modified. The first

450

term represents the cost of the tuple accesses that are transferred among partitions, without considering distributed transactions. The other two terms represent Cases 1 and 2 of Figure 4, respectively. This expression of $\Delta_s(M, p)$ can be computed by iterating only over the elements of M and its edges.

A similar approach can be used for the receiver delta $\Delta_r(M, d)$, where d is the receiving partition. Partition d incurs additional distributed transactions when a tuple v in M is co-accessed with a tuple u that is not in M or d (see Figure 4, Case 2). Note that partition o and p can be the same. The receiver delta incurs fewer distributed transactions when M contains a tuple v that was remote to d and is co-accessed with a tuple u in d (see Figure 4, Case 3).

Considering the previous cases, we can express the receiver delta function Δ_r for a clump M and receiving partition d as follows:

$$\Delta_r(M, d) = \sum_{\substack{v \in M \\ P(v) \neq d}} w(v) + \sum_{\substack{v \in M \\ \langle v, u \rangle \in E \\ u \notin M \\ P(v) \neq d \\ P(u) \neq d}} w(\langle v, u \rangle) \cdot k - \sum_{\substack{v \in M \\ \langle v, u \rangle \in E \\ P(v) \neq d \\ P(u) = d}} w(\langle v, u \rangle) \cdot k$$

where the first term corresponds to the cost of tuple accesses without considering distributed transactions, and the other two terms correspond to Cases 2 and 3 of Figure 4, respectively.

7.3 Scaling In

The methods described for Clay thus far are for scaling out the database when the workload demands increase. An elastic DBMS, however, must also consolidate the database to fewer partitions when the workload demands decrease. Clay scales in by first checking if there are underloaded partitions having a load lower than a certain threshold. This threshold can be computed as the average load of all partitions multiplied by $1 - \epsilon$, where ϵ is the imbalance factor introduced in Section 4. For each underloaded partition p, it tries to find a destination partition d that can take all of the tuples of p. If d reduces its load after receiving all of the tuples of p, or if d does not become overloaded, then the data of p is migrated into d and p is removed from the partitioning plan. Clay minimizes the number of active partitions by scanning underloaded partitions in descending id order and destination partitions in ascending id order.

7.4 Routing Table Compaction

Clay maintains two data structures for its internal state. The first is the heat graph that is discarded after each reconfiguration. The second is the DBMS's *routing table*, which encodes the partitioning plan and is replicated on each server. We now discuss how to keep this table from growing too large.

The routing table consists of a set of ranges for each table. Initially, there are a few large, contiguous ranges. When clumps are moved, however, contiguous ranges may be broken into subranges and hot tuples may be represented as small ranges. Over time, the routing table can get fragmented and large. If the routing table is too large, then the DBMS will incur substantial overhead when determining where to direct transactions and queries.

To solve this problem, Clay maintains a separate index that tracks which clumps have been moved from their initial location. Each clump is indexed by the hot tuple that started the clump. Before executing its clump migration algorithm, Clay computes a new routing table where the tuples within any clump whose starting tuple is no longer hot are moved back to their original location. Next, Clay invokes the clump migration algorithm with this new routing table as its input and migrates any data as needed based on the new routing table. The extra step guarantees that the size of the routing table only depends on the size of the clumps of currently hot tuples.
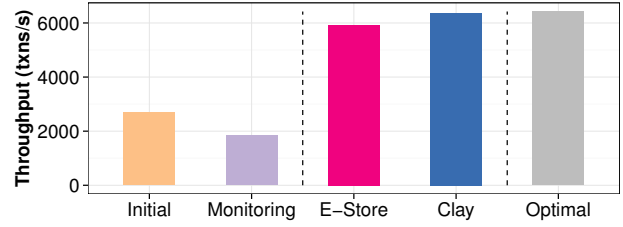

Figure 5: TPC-C Throughput – The measured throughput for H-Store using different partitioning algorithms before monitoring is started, during monitoring, and after the reconfiguration.

8. EVALUATION

We now evaluate the efficacy of Clay's clump migration mechanism using three benchmarks. The first is TPC-C, which is the most well-known tree-based OLTP benchmark. The second workload is Products-Parts-Suppliers (PPS), a "bird's nest" benchmark modeled after the database shown in Figure 1. The third benchmark is inspired by Twitter, the popular micro-blogging website.

We begin with experiments that use small database sizes to speed up loading time. We then explore larger databases to show that the databases size is not a key factor in the performance of Clay.

8.1 Environment

All of our experiments were conducted on an H-Store database deployed on a 10-node cluster running Ubuntu 12.04 (64-bit Linux 3.2.0), connected by a 1 Gb switch. Each machine had four 8-core Intel Xeon E7-4830 processors running at 2.13 GHz with 256 GB of DRAM.

In addition to Clay, we implemented two other reconfiguration algorithms in the H-Store environment. For all of the experiments, we provide the same workload monitoring information to all the algorithms to ensure a fair comparison. **Metis** uses its graph partitioning tool [16] to partition the same heat graph used by Clay. In order to obtain the correct number of partitions with small heat graphs, like the ones of TPC-C, we use multilevel recursive bisectioning. **E-Store** employs the greedy-extended reconfiguration heuristic proposed in [28]. It uses the monitoring traces only to calculate access frequencies and does not consider co-accesses.

We have selected the configurations of E-Store and Clay that result in the most balanced load, according to their different definition of load. For Metis, we report results under different settings for its imbalance threshold. The DBMS's migration controller is started on demand and runs on a separate node to avoid interference with other processes. We start all experiments from an initial configuration where data is spread uniformly, regardless of the hotness of the partitions.

8.2 TPC-C

The goal of this first experiment is to validate that Clay can generate the same partitioning scheme as E-Store without using the database's schema. For this, we use the TPC-C benchmark that models a warehouse-centric order processing application [29]. TPC-C's database has a well-defined tree-structure where tables are partitioned by their parent warehouse. We start with an initial placement that co-locates tuples of the same warehouse and uniformly distributes warehouses among the partitions.

We use a database with 100 warehouses (~10 GB) that are split across three servers in 18 partitions. In TPC-C, the NewOrder and Payment transactions make up about 85% of the workload and access multiple warehouses in 10% and 15% of the cases, respectively. E-Store performs best in scenarios with high skew, so
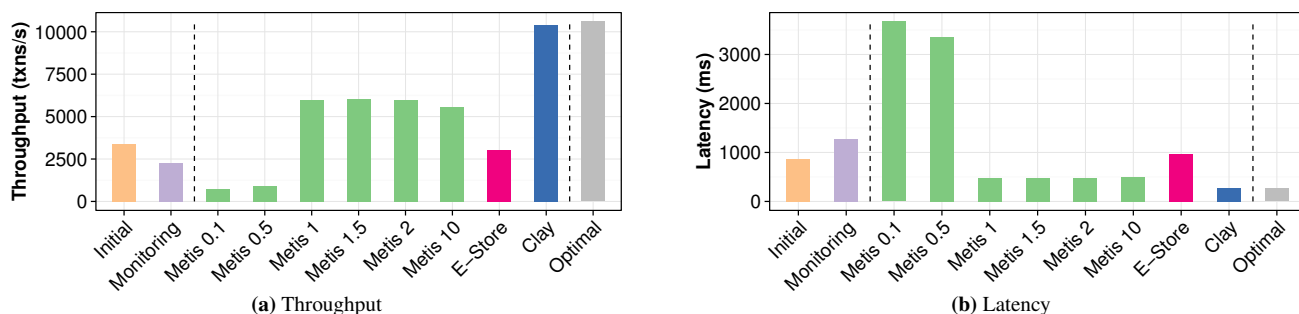
Figure 6: TPC-C/S Runtime Performance – The measured throughput and latency for H-Store using different partitioning algorithms before monitoring is started, during monitoring, and after the reconfiguration.

we modified the TPC-C workload generator such that 80% of the transactions access only three warehouses concentrated in the same partition. We use the default behavior of TPC-C, where every group of warehouses is equally likely to be accessed together. In this setting, it is sufficient for the reconfiguration algorithm to move all trees together and to relieve hotspots. Since no pair of warehouses is co-accessed more often than others, the placement of warehouses does not impact the rate of distributed transactions as long as all partitions receive approximately the same number of warehouses.

The performance of the reconfiguration algorithms is shown in Figure 5. As expected, Clay has similar performance to E-Store. The two algorithms increase throughput by 137% and 121% and reduce latency by 59% and 55%, respectively. Clay performs well because it moves warehouses in a block, even if it only relies on the heat graph to identify the tree structures related to warehouses. Both E-Store and Clay move the same two warehouses from the overloaded partition to another partition. We observed that Clay's look-ahead mechanism allows it to accurately identify trees. Tuples in the WAREHOUSE, STOCK, and DISTRICT tables are accessed more frequently than others, so Clay finds feasible migrations that offload overloaded servers by moving the tuples from these tables. Clay detects that moving other tuples of the same tree (i.e., related to the same warehouse) further reduces distributed transactions. The rate of distributed transactions does not change much after reconfiguration for both heuristics. The marginal performance improvement with Clay is because it achieves a slightly better balance in the amount of distributed transactions per server.

We also devised an optimal configuration that assumes a priori knowledge of future hotspots and co-accesses among tuples. We placed an equal number of warehouses on each server and evenly spread hot warehouses among servers. E-Store and Clay obtain plans that are very similar to the best-case one. They both spread hot warehouses correctly but they do not place a perfectly even number of cold warehouses on each server.

Figure 5 also shows the performance overhead of monitoring. The overhead is visible but not significant since monitoring runs only for 20 seconds. For this short period of time, throughput is reduced by 32% and latency is increased by 42%.

8.3 TPC-C/S

We next test whether Clay is able to find better configurations than E-Store and Metis by explicitly considering how TPC-C tuples from different warehouses are accessed together. We tweak the previous setup such that the probability of co-accesses among warehouses is skewed and each warehouse is only co-accessed with its "paired" warehouse. We call this version of the benchmark TPC-C/S. In this scenario, it is important that the migration plan places pairs of co-accessed warehouses on the same server.

Figure 6 shows the throughput and latency obtained with the reconfiguration algorithms. As in Figure 5, monitoring overhead is visible but not of great concern due to the short duration. Throughput is reduced by 33% and latency is increased by 46%. These results also show that Clay provides excellent performance; by explicitly tracking co-accesses, it manages to place together the paired warehouses, resulting in a 3.1× throughput increase and a 68% latency reduction. As in the previous experiment, Clay is able to detect the presence of a tree structure by looking at co-accesses among tuples, independent from the database structure. It moves all tuples related to the same warehouse together in the same clump.

We again created an optimal configuration that assumes knowledge of the hot tuples and their co-accessed tuples ahead of time. We grouped tuples in the same warehouse together, placed all the warehouses that are paired in the same partition, and uniformly distributed hot tuples across different servers. The main difference between the best-case plan and the one produced by Clay is that, in the latter, some cold paired warehouses are not placed together. Clay's solution is, however, close to the optimal because correctly placing hot tuples has more significant impact on performance.

For Metis, we consider multiple values of the imbalance threshold parameter (from 10% to 1000%). The results show that if this threshold is too low, then the solution space is limited and Metis does not manage to produce a low edge cut. Even if the load is balanced in terms of accesses to single tuples, there are still a large number of distributed transactions, which Metis does not regard as "load" (see Figure 7). This degrades the DBMS's performance after reconfiguration. Increasing the threshold reduces edge cuts and thus causes fewer distributed transactions. If we set the imbalance to be 100%, then Metis eliminates all distributed transactions by paring warehouses correctly. But the plans produced in this case have high imbalance, as shown in Figure 8, and thus the DBMS never achieves the same performance gains as with Clay. Furthermore, Metis only minimizes the total number of edges cut in the entire graph, and not the maximum number of edges cut per partition. As such, some servers run a disproportionate fraction of distributed transactions. The performance of E-Store is worse than Metis. This is because both try to balance the same load metric of Equation 3, which only reflects the frequency of tuple accesses and does not consider the cost of distributed transactions. Unlike E-Store, however, Metis also minimizes the number of distributed transactions and thus achieves better performance for some configurations.

Figure 9 reports the complexity of data movement, expressed in terms of the percentage of the database that needs to be migrated. As expected, E-Store moves the same two hot warehouses in TPC-C, since it does not detect the different co-access skew of TPC-C/S. Clay also moves two hot warehouses, but it places them together with their paired warehouse. The third hot warehouse is already
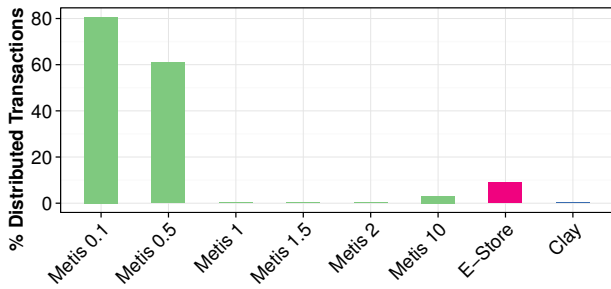
Figure 7: TPC-C/S Distributed Transactions – The percentage of transactions in the workload that access multiple partitions after reconfiguration.
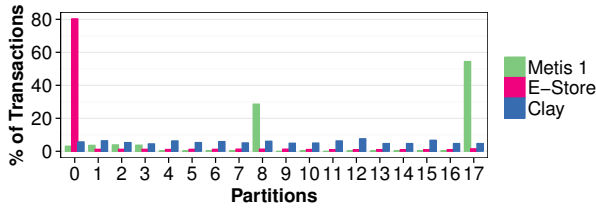

Figure 8: TPC-C/S Load Distribution – The measured distribution of the load among partitions for different heuristics. For each partition id, we show the percentage of transactions having that partition as base partition.
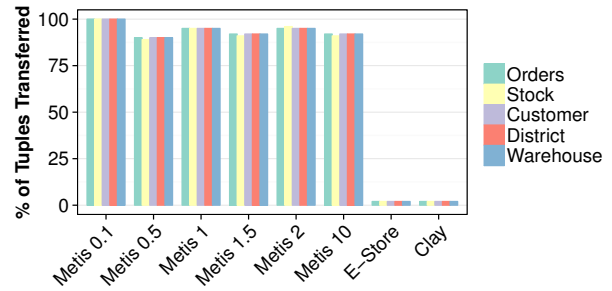

Figure 9: TPC-C/S Data Migration – Amount of data migrated by the different heuristics.

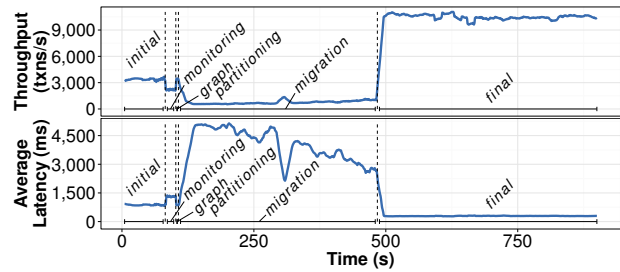**Figure 10: TPC-C/S Live Migration Timeline** – The sustained throughput and latency of the DBMS over time during a migration.

initially located with the paired one. Metis computes a new plan from scratch and thus transfers almost the whole database.

Figure 10 shows a timeline of the DBMS's sustained throughput and latency during data migration using Squall. Reconfiguration takes 373 seconds, during which the system remains live, albeit with reduced throughput. This performance impact is due to H-Store's single threaded execution engine that can either be migrating data or executing transactions. This impact is exacerbated when migration objects are large, such as all data related to a warehouse in TPC-C. Squall can achieve higher performance at the expense of a longer time to complete the reconfiguration [8].

8.4 Products-Parts-Suppliers

We implemented a Products-Parts-Suppliers benchmark like the one in Figure 1 to test the capabilities of the different heuristics. In our experiments, each product requires 10 parts, which are used only for that product. Suppliers produce 200 parts each and each part is produced by two different suppliers. The database has 30 partitions across 5 servers, and its size is 5 GB. We will discuss in the section on scalability why choosing such a small database is appropriate. Initially, we uniformly distribute products and suppliers across four servers. Since each part is uniquely associated with a single product, we colocate parts with their products. One extra server is left available for the system to scale out and re-balance whenever required.

The workload consists of the following five transactions, each corresponding to 20% of the transaction mix: (1) get one part, (2) get one product, (3) get one supplier, (4) get all parts of a product, and (5) get some of the parts of a supplier selected according to a Zipfian distribution. There are six hot suppliers, and the workload is skewed such that each hot supplier receives 3.33% of all "get one supplier" and "get some of the parts of a supplier" transactions. In total, the hot suppliers and their co-accessed tuples receive 20% of the workload for these two transactions, or 8% of all transactions. The remaining suppliers are accessed according to a gradual Zipfian distribution, where we set the skew parameter to 0.1. The other three transactions ("get one part", "get one product", and "get all parts of a product") access all parts and products uniformly.

The results in Figure 11 show the DBMS's performance obtained using schemes generated by the reconfiguration algorithms. As in the previous case, the overhead of monitoring is limited both in throughput and latency. Clay performs well, improving throughput by more than $3\times$ and reducing latency by 94%. It identifies hot suppliers and moves them to a partition on the extra server, along with their parts and some of their related products.

We again devised an optimal scenario that assumes perfect knowledge of the future workload behavior. We uniformly spread hot suppliers across servers, and co-located suppliers with their parts and parts with their products. Clay produces again a similar plan as the best-case, but it is more fragmented than the optimal one because the monitoring data misses some co-accesses between cold tuples. This also results in a slightly larger routing table.

E-Store can also scale the database out to the extra server. But because there are not well-defined blocks in the database, E-Store moves each tuple separately regardless of its co-access patterns. The results in Figure 12 show that it is unable to improve throughput or latency because it creates more distributed transactions.

The results for Metis are once again interesting. Unlike with TPC-C, setting a larger imbalance threshold does not result in better performance because the heat graph is a sample of the tuples that are accessed during the monitoring interval. Since transactions in TPC-C mostly accesses a small set of tuples, the sample is representative of the actual workload. Therefore, minimizing the number of edge cuts in the heat graph reduces the number of distributed transactions. But with the PPS benchmark, there is a much broader set of tuples that are accessed, and the heat graph only contains the hottest ones along with the set of colder tuples that happened to be accessed during the monitoring interval. This implies that heat graph accurately models the co-accesses with hot tuples, but not necessarily co-accesses among cold tuples. Metis does not consider the initial plan, so it freely shuffles tuples in the heat graph if this does not increase the number of edge cuts. As such, Metis often separates cold tuples whose co-accesses are not observed, and this causes a large number of distributed transactions.
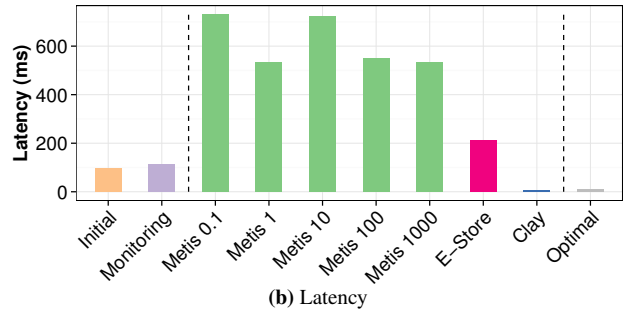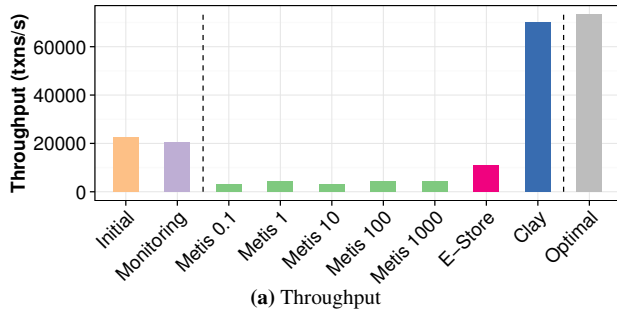
(a) Throughput

(b) Latency

Figure 11: PPS Runtime Performance – The measured throughput and latency for H-Store using different partitioning algorithms before monitoring is started, during monitoring, and after the reconfiguration.
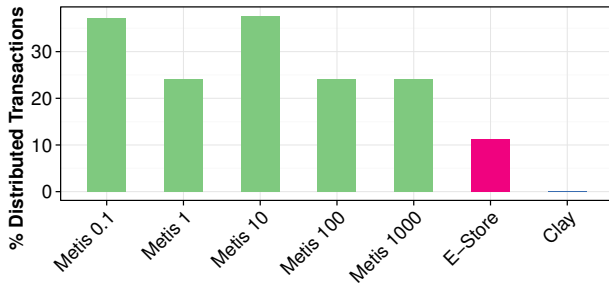

Figure 12: PPS Distributed Transactions – The percentage of transactions in the workload that access multiple partitions after reconfiguration.
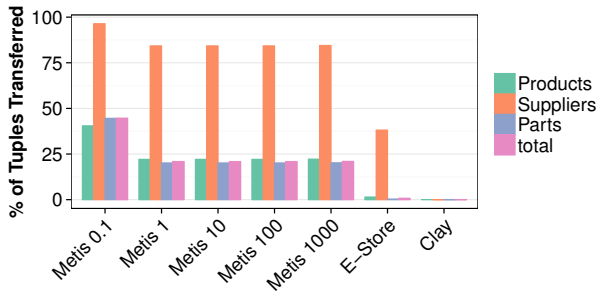

Figure 13: PPS Data Migrations – Amount of data migrated by the different heuristics.

Figure 14: PPS Live Migration Timeline – The sustained throughput and latency of the DBMS during a migration.

Figure 15: PPS Scale In Timeline – The sustained throughput and latency of the DBMS as it coalesces the database to a smaller cluster.

The problems with Metis are not specific to graph partitioning: any technique that builds a scheme based on a sampled graph will overlook a large number of low-frequency edges that are not included in the sample, and thus will create a large number of distributed transactions. Clay does not have this problem since it moves only hot tuples and tuple clumps that are co-accessed together. These hot tuples and edges are more accurately reflected in a sampled graph, like the heat graph, resulting in almost no distributed transactions. Although E-Store fails to identify co-accesses, it does construct a new plan incrementally, thereby reducing data shuffling and limiting the increase in distributed transactions.

The amount of data moved by the heuristics is reported in Figure 13. As in the case of TPC-C/S, Metis migrates a large number of tuples (i.e., more than 20% of the database). E-Store only transfers a few hot tuples, and as a result it migrates a tiny fraction of the database, 0.07%. Clay migrates even less data, 0.003% of the database, or 1225 tuples, because it has a more accurate load model and focuses on the tuples that generate the highest number of distributed transactions. Figure 14 shows that Squall can reconfigure PPS within a few seconds with limited performance impact.
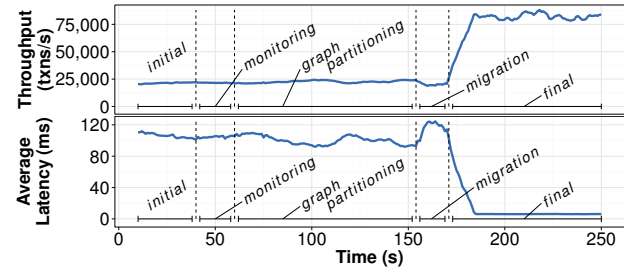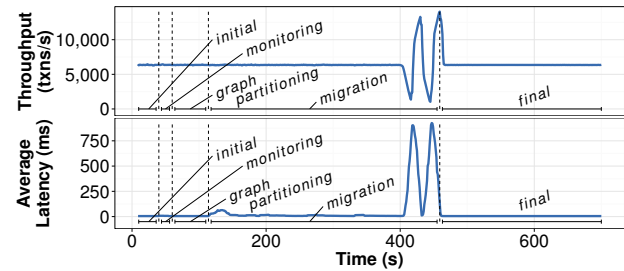
Clay also supports scaling-in as shown in Figure 15, which is triggered by reducing the clients' transaction rates by 10×. It retracts to one less server without a major performance impact.

8.5 Twitter

To test Clay's performance on a real-world non-tree schema, we used the Twitter benchmark from the OLTP-Bench testbed [7], which is inspired by the popular micro-blogging website. This benchmark contains five transactions: (1) get a user's tweets, (2) get the names of a user's followers, (3) insert a tweet, (4) retrieve a tweet, and (5) get tweets from all users that a given user is following. Transactions (1) and (3) are always local and account for 95% of the workload. The other three transactions account for the remaining 5% and may or may not be local.

To make the benchmark's access patterns more realistic, we defined the "follows" relationships between users according to a subset of the real Twitter social graph from August 2009 that contains 51m users and 2b "follows" relationships [4]. This means that any set of users identified by Clay as highly connected in our Twitter benchmark represents a cluster of highly connected users in the real social network. We also limited transactions (2) and (5) to 10

Table 1: Scalability of Clay using PPS variants – We reduced the skew by halving the fraction of requests sent to the hottest tuples (*Low skew*), increased the size of the database by $10\times$ (*10x DB size*), doubled the number of hosts ($2\times$ *hosts*), and considered a workload with three joins per transaction (*3 joins*). We also ran a large-scale experiment with 600 GB and 200 partitions (*120x DB size*). Throughput/latency variation is the ratio between the throughput after and before reconfiguration. t_p is the time required by the controller to load the graph and find a new plan. t_r is the time Squall takes to perform live migration. The total time to complete the reconfiguration is the monitoring time (typically 20s) plus ($t_p + t_r$).

	Vertices	Edges	Thrpt. Var.	Lat. Var.	t_p	t_r
Original PPS	1.08m	5.41m	3.1×	15×	94s	21s
Low Skew	2.12m	11.98m	1.8×	2.6×	75s	31s
10× DB Size	1.64m	7.39m	3.9×	6.9×	83s	18s
2× Hosts	1.42m	7.37m	2.9×	18×	128s	16s
3 Joins	800.7k	5.76m	4.6×	6.1×	130s	19s
All	775.7k	4.68m	2.6×	3.1×	51s	32s
120× DB Size	2.28m	10.11m	2.5×	0.4×	99s	-

followers/followees, selected according to a Zipfian distribution. Lastly, we skewed the access patterns of the original benchmark so that nodes (users) are chosen for transactions (1), (2), and (3) with a frequency proportional to the log of their in-degree (number of followers), and chosen for transaction (5) with a frequency proportional to the log of their out-degree (number of followees). The fact that a user's frequency of activity in a social network is related to their degrees has been observed in previous work [13, 10].

For this experiment, we evenly distributed the first 60,000 user IDs from the 2009 Twitter graph across a cluster of four machines, each with six partitions. Initially, the tweets and user profiles were range-partitioned across the cluster by user ID in large contiguous chunks. After using Clay to repartition the database, throughput increased by 55% and latency decreased by 37%. The percentage of distributed transactions was reduced from 3.2% to only 1.4%.

8.6 Scalability

In this section, we modify the PPS benchmark along several axes and establish that workload skew is the dimension that has the strongest impact on Clay's effectiveness and running time.

We focus our evaluation on PPS because the size of the heat graphs for TPC-C are much smaller: even if we increase the number of TPC-C warehouses to 500, thus obtaining a database of 50 GB, the heat graph still has only 2,496 vertices and 10,602 edges. This is because a single value of a partitioning attribute in TPC-C corresponds to a larger number of tuples than in PPS.

Detailed Scalability Experiments. Table 1 reports the results of different scalability tests on PPS. If we increase the size of the database to 50 GB or double the number of hosts we see similar gains as in the original PPS, and the complexity of clump migration does not grow significantly. Next, we changed PPS to include a transaction that executes three joins instead of two. Specifically, we replaced the transaction that accesses parts by product by a transaction that accesses products by supplier. Finally, we combined all the aforementioned variations. In all cases, the size of the graph and the time to compute a new plan do not vary significantly. The heat graph always fits in the 4 GB of heap space assigned to our controller, and the time to terminate a reconfiguration is on the order of 100s of seconds. The highest increase in the size of the graph was obtained by reducing the skew to a low degree (Zipfian with parameter 0.1 for all tuples except the hottest tuples, which get 4% of the transactions). Even in this case, the time necessary to compute a new configuration is low. The performance improvement of Clay in this case is lower since the initial performance level is better.
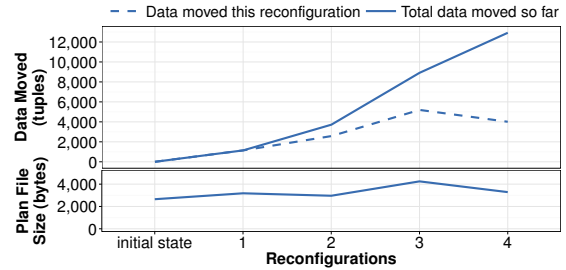

Figure 16: PPS Multiple Reconfigurations – The impact of multiple reconfigurations on the amount of data moved and the size of the plan file.

Table 2: Skew: Fraction of Requests to Hot Tuples in PPS – Effects of changing the percentage of requests sent to the 6 hot suppliers (and their co-accessed tuples) in PPS. See Table 1 for measurement descriptions.

% requests to hot tuples	Vertices	Edges	Thrpt. Var.	Lat. Var.	t_p	t_r
40	1.19m	6.04m	3.3×	4.1×	50s	14s
20	1.08m	5.41m	3.1×	15×	75s	31s
10	2.12m	11.98m	1.8×	2.6×	94s	21s
5	1.98m	11.38m	1.6×	2.3×	83s	17s
0	1.88m	10.53m	0.9×	0.9×	95s	11s

Table 3: Skew: Number of Hot Tuples in PPS – Effects of changing the number of hot tuples in PPS, compared to the baseline with 6 hot suppliers (and their co-accessed tuples). See Table 1 for measurement descriptions.

# hot tuples	Vertices	Edges	Thrpt. Var.	Lat. Var.	t_p	t_r
1x	1.08m	5.41m	3.1×	15×	75s	31s
10x	1.89m	10.56m	2.2×	3.2×	106s	19s
100x	1.83m	10.45m	1.1×	1.1×	181s	15s

Large-scale experiment. Our experiments so far considered relatively small datasets because populating a large database with H-Store can be very time-consuming. We now show that changing the database size alone does not significantly impact Clay's behavior.

In this experiment, we run Clay with a very large deployment of PPS with 600 GB and 200 partitions across 10 machines, thus increasing the size of the database by more than 100 times. All the other parameters of the benchmark are unchanged. As shown in Table 1, the performance of Clay does not change significantly. By increasing the database size 120 times, the number of edges and vertices in the heat graph only doubles compared to the original 5 GB PPS database. Throughput and latency performance is slightly reduced compared to the smaller database because of hardware limitations: we have only 10 servers for all client and server threads, not enough to fully saturate such a large database. Squall does not yet support 200 partitions, which is why we do not have a value for t_r. The amount of data moved is nearly identical to the 5 GB database, so we expect that t_r would be similar.

This experiment shows that the computational complexity of Clay does not change significantly. Increasing the size of the database without changing the number of hot tuples only increases the number of cold tuples, which are not observed by Clay's monitoring. As shown in Table 1, the size of the heat graph in the large-scale experiment is about the maximum size that we obtain by changing other parameters such as skew. In general, the maximum size of the heat graph is bounded since the vertices in the heat graph represent tuples accessed during the monitoring interval, and both the monitoring interval and the throughput of the system are bounded.

Metadata Management. In all our experiments on PPS, the size of the metadata (i.e., the routing table) does not vary significantly as

an effect of reconfiguration: in all the small experiments of Table 1, the size of the routing table stayed within 2.2 KB and 4.4 KB. In the large-scale experiment with 600 GB and 200 partitions, the routing table ranged between 19.6 KB and 22.6 KB. This is because Clay only moves a relatively small set of hot tuples. Routing tables in TPC-C are similar to PPS in the standard setups.

Next, we validated that the size of the metadata is only a function of the number of currently hot clumps, and it does not grow over time. We executed multiple reconfigurations in the PPS benchmark. In each configuration, we completely change the set of hot tuples. The results are shown in Figure 16. The upper plot shows the amount of data transferred at each reconfiguration as well as the cumulative amount of data moved up to that reconfiguration. The size of the routing table remains bounded because it is a function of the data transferred in the current reconfiguration. It does not depend on the cumulative amount of data moved over time.

The Role of Skew. As discussed, Clay is optimized to deal with hot tuples, so it is expected to shine in workloads that are highly skewed and to have lower performance with lower skew. Higher skew also results in a smaller heat graph, which in turns speeds up the reconfiguration process. Our evaluation confirms this observation. Table 2 reports the performance of Clay when varying the percentage of requests that go to the hot tuples. As expected, lower skew leads to a slightly larger graph, longer running times of the controller, and lower performance gains. With no hot tuples Clay minimally degrades performance, but in this circumstance we would not expect Clay to be triggered. Table 3 shows the effect of changing the number of hot tuples while keeping the fraction of requests to them constant. Increasing the number of hot tuples has a similar effect to reducing the skew, because now more hot tuples are sharing the same number of requests. In all our experiments, we observe that the size of the heat graph and the running time of the clump migration algorithm stay within reasonable bounds.

9. CONCLUSION

This paper presented Clay, an on-line DBMS elasticity algorithm that makes no assumptions about the database schema, and simultaneously balances load and minimizes distributed transactions. Clay defines the unit of data migration, called the clump, in a novel way by starting from a hot tuple and including other tuples co-accessed with that hot tuple so that migrating the clump does not overload the sending or receiving server with distributed transactions. Our experiments show that Clay substantially outperforms state-of-the-art database elasticity techniques.

10. REFERENCES

[1] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 739–753, 2016.

[2] J. Baker, C. Bond, J. C. Corbett, 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 *Conference on Innovative Data Systems Research*, pages 223–234, 2011.

[3] P. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting Microsoft SQL server for cloud computing. In *IEEE International Conference on Data Engineering*, pages 1255–1263, 2011.

[4] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *AAAI Conference on Web and Social Media*, pages 10–17, 2010.

[5] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[6] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems*, 38(1):5:1–5:45, 2013.

[7] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

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

[9] M. Ghosh, W. Wang, G. Holla, and I. Gupta. Morphus: Supporting online reconfigurations in sharded nosql systems. In *IEEE International Conference on Autonomic Computing*, pages 1–10, 2015.

[10] A. Gionis, F. Junqueira, V. Leroy, M. Serafini, and I. Weber. Piggybacking on social networks. *Proceedings of the VLDB Endowment*, 6(6):409–420, 2013.

[11] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *ACM Symposium on Cloud Computing*, 2013.

[12] H-Store: A Next Generation OLTP DBMS. http://hstore.cs.brown.edu.

[13] B. A. Huberman, D. M. Romero, and F. Wu. Social networks that matter: Twitter under the microscope. *SSRN 1313405*, 2008.

[14] A. Jindal and J. Dittrich. Relax and let the database do the partitioning online. In *International Workshop on Enabling Real-Time Business Intelligence*, pages 65–80. 2012.

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[16] G. Karypis and V. Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 5.0. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[17] B. Liu, J. Tatemura, O. Po, W.-P. Hsiung, and H. Hacigumus. Automatic entity-grouping for OLTP workloads. In *IEEE International Conference on Data Engineering*, pages 712–723, 2014.

[18] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *IEEE International Conference on Data Engineering*, pages 137–148, 2015.

[19] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *IEEE International Conference on Data Engineering*, pages 589–600, 2016.

[20] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *ACM SIGMOD International Conference on Management of data*, pages 1137–1148, 2011.

[21] NuoDB. http://www.nuodb.com.

[22] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page latch-free shared-everything OLTP. *Proceedings of the VLDB Endowment*, 4(10):610–621, 2011.

[23] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2012.

[24] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. Atrapos: Adaptive transaction processing on hardware islands. In *IEEE International Conference on Data Engineering*, pages 688–699, 2014.

[25] A. Quamar, K. A. Kumar, and A. Deshpande. Sword: scalable workload-aware data placement for transactional workloads. In *International Conference on Extending Database Technology*, pages 430–441, 2013.

[26] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046, 2014.

[27] M. Stonebraker, A. Pavlo, R. Taft, and M. L. Brodie. Enterprise database applications and the cloud: A difficult road ahead. In *IEEE International Conference on Cloud Engineering*, pages 1–6, 2014.

[28] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[29] The TPC-C Benchmark, 1992. http://www.tpc.org/tpcc/.

[30] K. Q. Tran, J. F. Naughton, B. Sundarmurthy, and D. Tsirogiannis. JECB: A join-extension, code-based approach to OLTP data partitioning. In *ACM SIGMOD International Conference on Management of Data*, pages 39–50, 2014.

[31] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The SCADS director: Scaling a distributed storage system under stringent performance requirements. In *USENIX Conference on File and Storage Technologies*, pages 163–176, 2011.

[32] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *ACM SIGMOD International Conference on Management of Data*, pages 517–528, 2012.

[33] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *ACM SIGMOD International Conference on Management of Data*, pages 17–30, 2015.