Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions

Marco Serafini
Qatar Computing Research Institute
mserafini@qf.org.qa

Essam Mansour
Qatar Computing Research Institute
emansour@qf.org.qa

Ashraf Aboulnaga
Qatar Computing Research Institute
aaboulnaga@qf.org.qa

Kenneth Salem
University of Waterloo
kmsalem@uwaterloo.ca

Taha Rafiq
Amazon.com
taharafi@amazon.com

Umar Farooq Minhas
IBM Research Almaden
ufminhas@us.ibm.com

ABSTRACT

Providing the ability to elastically use more or fewer servers on demand (scale out and scale in) as the load varies is essential for database management systems (DBMSes) deployed on today's distributed computing platforms, such as the cloud. This requires solving the problem of dynamic (online) data placement, which has so far been addressed only for workloads where all transactions are local to one sever. In DBMSes where ACID transactions can access more than one partition, distributed transactions represent a major performance bottleneck. Scaling out and spreading data across a larger number of servers does not necessarily result in a linear increase in the overall system throughput, because transactions that used to access only one server may become distributed.

In this paper we present Accordion, a dynamic data placement system for partition-based DBMSes that support ACID transactions (local or distributed). It does so by explicitly considering the *affinity* between partitions, which indicates the frequency in which they are accessed together by the same transactions. Accordion estimates the capacity of a server by explicitly considering the impact of distributed transactions and affinity on the maximum throughput of the server. It then integrates this estimation in a mixed-integer linear program to explore the space of possible configurations and decide whether to scale out. We implemented Accordion and evaluated it using H-Store, a shared-nothing in-memory DBMS. Our results using the TPC-C and YCSB benchmarks show that Accordion achieves benefits compared to alternative heuristics of up to an order of magnitude reduction in the number of servers used and in the amount of data migrated.

1. INTRODUCTION

Today's distributed computing platforms, namely clusters and public/private clouds, enable applications to effectively use resources in an on demand fashion, e.g., by asking for more servers when the load increases and releasing them when the load decreases. Such *elastic* applications fit well with the pay-as-you-go cost model of computing clouds, such as Amazon's EC2. These clouds have a large pool of physical or virtual servers, and can *scale out* and *scale in* depending on the needs of their (time varying) workload in an *online* fashion. However, not all applications are designed to achieve online elastic scalability. We advocate that DBMSes have to be enhanced to provide elasticity because they are at the core of many data intensive applications deployed on clouds. These applications will directly benefit from the elasticity of their DBMSes.

As a basis for DBMS elasticity, (shared nothing or data sharing) *partition-based* database systems could be used. These systems use mature and proven technology for enabling multiple servers to manage a database. The database is horizontally partitioned among the servers and each partition is "owned" by exactly one server [5, 17]. The DBMS coordinates query processing and transaction management among the servers to provide good performance and guarantee the ACID properties. Performance is sensitive to how the data is partitioned, and algorithms for good data partitioning exist (e.g., [8, 20]), but the *placement* of partitions on servers is usually static and is computed offline by analysing workload traces.

To make a partition-based DBMS elastic, the system needs to be changed to allow servers to be added and removed dynamically while the system is running, and to enable live migration of partitions between servers. With these changes, a DBMS can start with a small number of servers that manage the database partitions, and can add servers and migrate partitions to them to scale out if the load increases. Conversely, the DBMS can migrate partitions from servers and remove these servers from the system to scale in if the load decreases. Mechanisms for effectively implementing partition migration have been proposed in the literature [18, 11, 12]. Using these mechanisms requires addressing the important question of *which partitions to migrate when scaling out or in*.

This paper presents *Accordion*, a dynamic partition placement controller for in-memory elastic DBMSes that supports *distributed ACID transactions*, i.e., transactions that access multiple servers. Most partition-based DBMSes support distributed transactions, but the significant impact of these transactions on the design of elasticity mechanisms has been ignored by previous work. Accordion is designed for dynamic settings where the workload is not known in advance, the load intensity can fluctuate over time, and access skew among different partitions can arise at any time in an unpredictable manner. Accordion computes partition placements that (*i*) keep the load on each server below its capacity, (*ii*) minimize the amount of data moved between servers to transition from the current placement to the one proposed by Accordion, and (*iii*) minimize the number of servers used (thus scaling out or in as needed).

A key feature of Accordion compared to prior work is that it can handle workloads that include ACID distributed transactions in addition to single-partition transactions. Distributed transactions appear in many workloads, including standard benchmarks such as TPC-C (in which 10% of New Order transactions and 15% of Payment transactions access more than one partition [25]). These workloads include join operations between tuples in different partitions hosted by different servers. Accordion targets in-memory databases, where distributed transactions represent a major source of overhead [16, 20]. However, mitigating the performance cost of distributed transactions is an important problem also for disk-based systems, as witnessed for example by recent work on Spanner [6], Calvin [24], and RAMP transactions [2]. Coordinating execution between multiple partitions executing a transaction requires blocking the execution of transactions (e.g., in the case of distributed locking) or aborting transactions (e.g., in the case of optimistic concurrency control). In either case, the maximum throughput capacity of a server may be bound not by its CPU, I/O, or networking capacity, which can be easily monitored using operating system level metrics, but by the overhead of transaction coordination.

Any algorithm for dynamic partition placement needs to determine the *capacity* of the servers to decide if a placement is feasible, i.e., does not overload any server. Accordion measures the server capacity in terms of the transactions per second that this server can support. We call this the *throughput capacity* of the server.

The first contribution of Accordion is a server capacity estimation component that specifically models bottlenecks due to distributed transactions. Existing work, such as Kairos [9] and DBSeer [19], models several types of possible throughput bottlenecks but not distributed transactions. Our work fills this gap. The model is based on the *affinity* between partitions, which is the frequency in which two partitions are accessed together by the same transactions. In workloads where each transaction accesses a single partition, we have *null affinity*. In this case, which is the one considered by existing work on dynamic partition placement [26, 10], the throughput capacity of a server is independent of partition placement and we have a fixed capacity for all servers. The second class of workloads we consider have *uniform affinity*, which means that all pairs of partitions are equally likely to be accessed together by a multi-partition transaction. We argue that the throughput capacity of a server in this case is a function of only the number of partitions the server hosts in a given partition placement. Uniform affinity may arise in some specific workloads, such as TPC-C, and more generally in large databases where rows are partitioned in a workload-independent manner. Finally, in workloads with *arbitrary affinity*, certain groups of partitions are more likely to be accessed together. In this case, the server capacity estimator must consider the number of partitions a server hosts as well as the exact rate of distributed transactions a server executes given a partition placement. This rate can be computed considering the affinity between the partitions hosted by the servers and the remaining partitions. Accordion estimates server capacity online, without assuming prior knowledge of the workload.

The second contribution of Accordion is the ability to dynamically compute a new partition placement given the current load on the system and the server capacity estimations obtained online. Distributed transactions also make partition placement more complex. If each transaction accesses only one partition, we can assume that migrating a partition p from a server s_1 to a server s_2 will impose on s_2 exactly the same load as p imposes on s_1, so scaling out and adding new servers can result in a linear increase in the overall throughput capacity of the system. This might not be true in the presence of distributed transactions: after migration, some multi-partition transaction involving p that was local to s_1 might become distributed, imposing additional overhead on both s_1 and s_2. This implies that we must be cautious before scaling out because distributed transactions can make the addition of new servers less beneficial, and in some extreme case even detrimental. Therefore, a partition placement algorithm should consider all solutions that use a given number of servers before choosing to scale out.

Accordion formulates the dynamic partition placement problem as a mixed integer linear programming (MILP) problem, and uses a MILP solver to consider all possible configurations with a given number of servers. We chose a MILP approach because it produces high quality results, it can run reliably without human supervision, and it is fast enough to be solved online. The major complexity of casting the problem as a MILP problem is that, with distributed transactions, some constraints (throughput capacity) are a function of the optimization variable (the placement of partitions). This relation is, in itself, not linear; one of the contributions of this paper is to come up with a linear formulation that can be integrated in the MILP problem and solved reliably and efficiently.

We implemented Accordion and evaluated it using H-Store, a scalable shared-nothing in-memory DBMS [15]. Our results using the TPC-C and YCSB benchmarks show that Accordion outperforms baseline solutions in terms of data movement and number of servers used. The benefit of using Accordion grows as the number of partitions in the system grows, and also if there is affinity between partitions. Accordion can save more than 10x in the number of servers used and in the volume of data migrated compared to other heuristics. We also show that Accordion is fast, so it can be used in an online setting with a large number of partitions.

The contributions of this paper can be summarized as follows: (*i*) a formalization of how distributed transactions and partition affinity determine the throughput capacity of servers, and an algorithm to estimate this relationship online, (*ii*) an online partition placement algorithm that expresses this non-linear relationship using a MILP, and (*iii*) an experimental evaluation using H-Store, which includes a comparison to simpler placement heuristics designed primarily for workloads with single partitions transactions.

The rest of the paper is organized as follows. In Section 2 we review related work. Section 3 presents an overview of Accordion. Section 4 discusses server capacity estimation in Accordion. Section 5 presents the Accordion partition placement algorithm. Section 6 is the experimental evaluation, and Section 7 concludes.

2. RELATED WORK

Our paper is the first to study dynamic and elastic partition placement in DBMSes with ACID distributed transactions. Previous work has addressed several related problems.

Non-elastic partition placement. The problem of partition placement (or more generally data placement) for load balancing has been studied extensively in the past in the context of database systems [1, 5, 14, 17, 21]. Previous research either does not deal with dynamic data re-distribution, or if it does, it works with a fixed number of servers. Clearly, this is not sufficient for the case where servers can be added or removed dynamically for elastic scaling. An additional limitation of most previous research is that the workload is known in advance. We propose a lightweight technique that can be used online with minimal prior knowledge.

Elastic partition placement without distributed transactions. SCADS Director addresses the partition placement problem in the context of simple key-value storage systems using a greedy heuristic [26]. Accordion can handle DBMS workloads where each transaction accesses a single partition, which are similar to key-value workloads, but also more general workloads with distributed transactions. ElasTras [10] is an elastically scalable, fault-tolerant, transactional DBMS for the cloud. ElasTras uses a greedy heuristic to decide which partitions to migrate when scaling, and it does not consider distributed transactions. One of our baselines for evaluation is a greedy heuristic similar to the ones used by SCADS Director and ElasTras; we show that Accordion outperforms such a heuristic by a wide margin.

Data partitioning. Recent work has investigated how to decide which tuples should belong to which database partitions offline, using workload traces. Schism models the database as a graph where each tuple is a vertex [8]. In [20], Pavlo et al. present a technique for identifying the best attributes for data and stored procedure partitioning. This line of work performs complex and computationally expensive offline analysis of static workload traces in order to define partitions that support skew and minimize the amount of distributed transactions. The underlying assumption is that the workload will not change at runtime; if it does change, partitioning must be repeated from scratch. Accordion is orthogonal because it takes the initial definition of partitions as an input and does not modify it. It does not perform offline analysis of static workload traces.

Multi-tenant resource allocation and performance modeling. Some work has focused on tenant consolidation in a multi-tenant OLTP DBMS, such as Kairos [9], RTP [22], and Pythia [13]. A tenant owns one or more independent databases, which must be dynamically allocated to the lowest possible number of servers. All tenants share the same underlying DBMS. Partition placement and tenant placement are different problems. Tenant placement models do not consider the impact of distributed transactions on server capacity, as for example the possible reduction of per-server capacity when scaling out. Tenants are independent so they can run completely uncorrelated and heterogeneous workloads. In partition placement, we consider a single tenant database that is horizontally partitioned. Horizontal partitioning makes the transaction mix homogeneous across all partitions, as we will discuss in Section 4.

Kairos proposes new models to estimate the buffer pool and disk utilization of each tenant in a shared DBMS. It places tenants using non-linear mixed integer programming and guarantees that each server has enough resources to host its tenants. The utilization of a server depends only on the set of tenants placed on that server, and there is no notion of distributed transactions involving remote tenants. The maximum capacity of a server is constant and therefore independent of the placement of tenants. In Accordion, due to the effect of distributed transactions, the throughput capacity of a server depends on the placement of partitions. In addition, Kairos assumes that the workload is known in advance, while Accordion does not require that.

Delphi is a self-managing controller for multi-tenant DBMSes based on a methodology called Pythia [13], which labels tenants, for example as disk- or CPU-intensive. The initial labeling is done by the DBA, running each tenant in isolation. Pythia learns which combinations of tenants are amenable to being placed together because they have complementary resource utilization. Measuring the performance of a single partition in a server while all other partitions run on different servers would generate a large amount of distributed transactions and significantly bias the learning process. In our experiments with TPC-C, we found a 2x difference in the transaction rate a partition can process in a best-case configuration compared to running in isolation. Therefore, the labeling results will be misleading for Pythia, if it is used to solve the partition placement problem. Moreover, Pythia learns whether a group of tenants can be placed together only based on their resource utilization, not based on the frequency of accessing a set of partitions

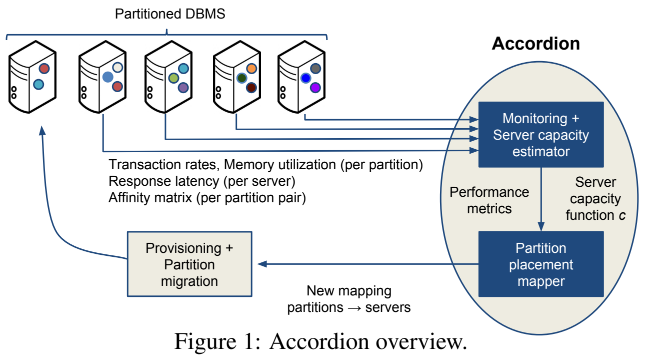

Partitioned DBMS
Accordion
Monitoring + Server capacity estimator
Transaction rates, Memory utilization (per partition)
Response latency (per server)
Affinity matrix (per partition pair)
Performance metrics
Server capacity function *c*
Provisioning + Partition migration
Partition placement mapper
New mapping partitions → servers

Figure 1: Accordion overview.

together. In partition placement, two partitions may have the same resource utilization if run in isolation but they should be treated separately if they have different affinity with other partitions.

DBSeer [19] proposes performance prediction models for contention in disk access, buffer pool access, and fine-grained locks. It does not directly address the problem of tenant placement. We propose a complementary model for bottlenecks due to distributed transactions. As discussed, different bottleneck models can be modeled as independent placement constraints.

3. OVERVIEW OF ACCORDION

This section describes a high-level view of the Accordion system and introduces the important concept of affinity among partitions.

3.1 System Components

The high level architecture of Accordion is illustrated in Figure 1. We consider a homogeneous DBMS cluster in terms of servers and networking capacity. Accordion operates with databases that are partitioned *horizontally*, i.e., where each partition is a subset of the rows of each database table. The partitioning of the database is done by the DBA or by some external partitioning mechanism, and is not modified by Accordion. The goal of Accordion is to migrate these partitions among servers online in order to elastically adapt to a dynamic workload.

The *monitoring* component of Accordion periodically collects: (*i*) an affinity matrix, which describes the behavior of distributed transactions, (*ii*) the rate of transactions processed by each partition, which represents system load and the skew in the workload, (*iii*) the overall request latency of a server, which is used to detect overload, and (*iv*) the memory utilization of each partition a server hosts.

The *server capacity estimator* uses the monitoring information to evaluate a *server capacity function*. This function describes the maximum transaction rate a server can process given the partition placement and affinity among partitions. Accordion estimates the server capacity online, without prior knowledge of the workload.

Monitoring metrics and server capacity functions are given as input to the *partition placement mapper*, which computes a new mapping of partitions to servers. If the new mapping is different from the current one, it is necessary to migrate partitions and possibly add or remove servers from the server pool. Migration and server provisioning are mechanisms external to Accordion.

3.2 Affinity and Affinity Classes

We define the affinity between two partitions p and q as the rate of transactions accessing both p and q. The monitoring component of Accordion constantly observes the current affinity between each pair of partitions and records it in the affinity matrix. The partition placement planner uses the affinity between p and q to determine how many distributed transactions will be generated by placing p and q on different servers.

The affinity matrix is also used by the server capacity estimator to classify the current workload as *null affinity*, *uniform affinity*, or *arbitrary affinity*. In workloads where all transactions access a single partition, the affinity among every pair of partitions is zero; i.e. the workload has *null affinity*. A workload has *uniform affinity* if the affinity value is roughly the same across all partition pairs. Workloads with uniform affinity are particularly likely in large databases where partitioning is done automatically without considering application semantics. A good example is databases that assign a random unique id or hash value to each tuple and use it to determine the partition where tuples should be placed. Finally, *arbitrary affinity* arises when clusters of partitions are more likely to be accessed together by the same transactions.

The affinity classes we have introduced determine the complexity of server capacity estimation and partition planning. Simpler affinity patterns, e.g. null affinity, make capacity estimation simpler and partition placement faster.

4. SERVER CAPACITY ESTIMATION

The first Accordion component we describe is the server capacity estimator. The throughput capacity of a server is the maximum transaction per second (tps) a server can sustain before its response time exceeds a user-defined bound. OLTP DBMSes have several potential throughput bottlenecks. Accordion specifically models bottlenecks due to distributed transactions, which are a major (if not the main) bottleneck in horizontally partitioned databases [2, 3, 6, 7, 20, 24].

In general, multi-partition transactions (of which distributed transactions are a special case) have a server that acts as a coordinator and execute the locking and commit protocols. If all partitions accessed by the transaction are local to the same server, the coordination requires only internal communication inside the server, which is efficient. However, if some of the partitions are remote, blocking time becomes significant. This is particularly evident in high-throughput, in-memory database systems like H-Store, our target system. In H-Store, single partition transactions execute without locks or latches if they do not interfere with multi-partition transactions, and thus can often complete in a few milliseconds.

Accordion characterizes the capacity of a server as a function of the rate of distributed transactions the server executes. It models this rate for a server s as a function of the affinity matrix F and the binary placement mapping matrix A, where $A_{ps} = 1$ if and only if partition p is assigned to server s. Therefore, the server capacity estimator outputs a *server capacity function* of the form:

$$c(s, A, F).$$

The partition planner uses this function to decide whether servers are overloaded and additional servers are thus needed. We consider different types of the capacity functions based on the affinity class of the workload, as discussed in Section 4.2. We show how to estimate these functions in Section 4.3.

We consider two workloads throughout the paper. YCSB [4] is a representative of workloads with only single-partition transactions. The H-Store implementation of YCSB uses a single table and each key-value pair is a tuple of that table. Tuples are partitioned based on the key attribute. Get, put, and scan operations are implemented as transactions that access a single partition. The second workload is TPC-C, which models the activity of a wholesale supplier [25]. In TPC-C, 10% of the transactions access data belonging to multiple warehouses. In the implementation of TPC-C over H-Store, each partition consists of one tuple from the Warehouse table and all the rows of other tables referring to that warehouse through a foreign key attribute. Therefore, 10% of the transactions access multiple partitions. Note that we measure the TPC-C throughput considering transactions of all types, not only New Order transactions as in the benchmark specification.

Transaction type	**Server 1**	**Server 2**
Delivery	3.84%	3.83%
New order	45.17%	45.12%
Order status by customer ID	1.54%	1.54%
Order status by customer name	2.31%	2.30%
Payment by customer ID	17.18%	17.21%
Payment by customer name	25.81%	25.83%
Stock level	4.15%	4.18%
Total no. of transactions	**5392156**	**19794730**

Table 1: Transaction mix for two servers of a six-server cluster running TPC-C with skewed per-partition transaction rate.

4.1 OLTP Workload Characteristics

Accordion characterizes load in terms of transaction rates per partition, which reflects several dimensions of dynamic workloads: (*i*) *horizontal skew*, i.e., some partitions are accessed more frequently than others, (*ii*) *temporal skew*, i.e., the skew distribution changes over time, and (*iii*) *load fluctuation*, i.e., the overall transaction rate submitted to the system varies. Accordion distinguishes between two types of transactions, local and distributed, under the assumption that transactions of the same type have the same average cost. We do not distinguish between single-partition transactions and multi-partition transactions accessing partitions located on the same server because the latter have a very low additional cost compared to multi-server (distributed) transactions. We now discuss why this assumption typically applies to *horizontally partitioned* DBMSes having a *stable workload*.

Workload stability is required because each capacity function is specific to a *global transaction mix* expressed as a tuple $\langle f_1, \ldots, f_n \rangle$, where f_i is the fraction of transactions of type i in the current workload. Every time the transaction mix changes significantly, the current estimate of the capacity function c is discarded and a new estimate is rebuilt from scratch. Until sufficient capacity estimates are collected, Accordion may overestimate server capacity. Accordion thus adapts to changes in the mix as long as they are infrequent, which is what happens in OLTP workloads [23].

When the global transaction mix is stable, we can often assume, thanks to horizontal partitioning, that the per-partition and per-server transaction mixes are the same as the global transaction mix. This is because horizontal partitioning spreads tuples from all database tables uniformly over all partitions. For example, Table 1 reports the fraction of transactions per type received by two servers (the most and least loaded) in a cluster of 6 servers and 256 TPC-C partitions after a run in H-Store. The transaction mix is exactly the same for both servers and is equal to the global transaction mix. The same holds for all other servers. All our experiments with YCSB and TPC-C and varied per-partition access skew and configurations have confirmed this observation. Therefore, it is reasonable to assume that the average cost of transactions of the same type (local or distributed) is uniform.

4.2 Affinity Classes and Capacity Functions

We now discuss how different affinity classes correspond to different types of capacity functions. For all our experiments we use a cluster where each server has two Intel Xeon quad-core processors running at 2.67GHz, and 32 GB of RAM. The servers are connected to a 10Gb Ethernet network. The purpose of these experiments is to show typical performance trends; the server capacity functions we devise are generic and extend to other DBMSes and workloads. The values reported are the average of five runs.

4.2.1 Null Affinity

We start by discussing the server capacity function for workloads where each transaction accesses a single partition. In these workloads the affinity between every pair of partitions is zero and there are no distributed transactions. Transactions accessing different partitions do not interfere with each other. Therefore, scaling out the system should result in a nearly linear capacity increase; in other words, the server capacity function is equal to a constant $\bar{c}$ and is independent of the value of A and F:

$$c(s,A,F) = \bar{c} \tag{1}$$

This is consistent with our claim that server capacity is a function of the rate of distributed transactions: if this rate is equal to zero regardless of A, then the capacity is constant.

We validate this observation by evaluating YCSB databases of different sizes, ranging from 8 to 64 partitions overall, where the size of each partition is fixed. For every database size, we consider two placement matrices A: one where each server hosts 8 partition and one where each server hosts 16 partitions. We chose the configuration with 8 partitions per server because we use servers with 8 cores; with 16 partitions we have doubled this figure. Figure 2 reports the capacity per server in different configurations of the system. The results show that, for a given total database size (x axis), the capacity of a server is not impacted by the placement A. Consider for example a system with 32 partitions: if we go from a configuration with 8 partitions per server (4 servers in total) to a configuration with 16 partitions (2 servers in total) the throughput per server does not change. This also implies that scaling out from 2 to 4 servers doubles the overall system capacity.

4.2.2 Uniform Affinity

In workloads with uniform affinity, where each pair of partitions is (approximately) equally likely to be accessed together, the rate of distributed transactions depends only on the number of partitions a server hosts: the higher the partition count per server, the lower the distributed transaction rate. Based on these observations, Equation 2 defines the server capacity function with uniform affinity workloads, where P is the set of partitions in the database.

$$c(s,A,F) = f(|\{p \in P : A_{ps} = 1\}|) \tag{2}$$

To validate this function, we consider the TPC-C workload. The TPC-C workload has uniform affinity because each multi-partition transaction randomly selects the partitions (i.e., the warehouses) it accesses following a uniform distribution. Our experiments, reported in Figure 3, confirm that distributed transactions have a major impact on server capacity. We consider the same set of configurations as in the previous experiments. Doubling the number of servers and going from 16 to 8 partitions per server significantly decreases the capacity of a server in each of the configuration we consider. In terms of global throughput capacity, scaling out is actually detrimental in some configurations. This is because server capacity is a function of the rate of distributed transactions.

Consider the example of a database having a total of 32 partitions. The maximum throughput per server in a configuration with 16 partitions per server and 2 servers in total is approximately two times the value with 8 partitions per server and 4 servers in total. Therefore, scaling out does not increase the total throughput

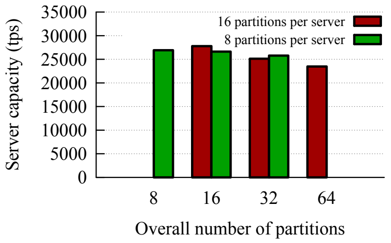


Figure 2: Server capacity with null affinity (YCSB).

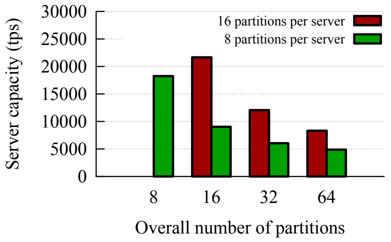


Figure 3: Server capacity with uniform affinity (TPC-C).

of the system in this example. This is because in TPC-C most multi-partition transactions access two partitions. With 2 servers about 50% of the multi-partition transactions are local to a server. After scaling out to 4 servers, this figure drops to approximately 25% percent (i.e., we have ∼75% of distributed transactions). We see a similar effect when there is a total of 16 partitions. Scaling from 1 to 2 servers actually results in a reduction in performance, as we transition from having only local multi-partition transactions to having ∼50% distributed multi-partition transactions.

Scaling out is more advantageous in configurations where every server hosts a smaller fraction of the total database. We see this effect starting with 64 partitions in Figure 3. With 16 partitions per server (i.e., 4 servers) the capacity per server is less than 10000 so the total capacity is less than 40000. With 8 partitions per server (i.e., 8 servers) the total capacity is 40000. This gain increases as the size of the database grows. Consider for example a larger database with 256 partitions and uniform affinity workload: if each server hosts 16 partitions, i.e. less than 7% of the database, then less than 7% of the multi-partition transactions access only partitions that are local to a server. If a scale out leaves a server with 8 partitions only, the fraction of partitions hosted by the server becomes ∼3.5%, so the rate of distributed transactions per server does not vary significantly in absolute terms. Therefore, the additional servers actually increase the overall capacity of the system.

4.2.3 Arbitrary Affinity

In the more general setting where distributed transactions exhibit arbitrary affinity, the rate of distributed transactions for a server can be expressed as a function of the placement matrix A and the affinity matrix F. Consider a server s, a partition p hosted by s, and a partition q hosted by another server (so $A_{ps} = 1$ and $A_{qs} = 0$) The two partitions add a term equal to F_{pq} to the rate of distributed transactions executed by s. Since we have arbitrary affinity, the F_{pq} values will not be uniform. Based on these observations, we model server capacity using several server capacity functions, one for each value of the number of partitions a server hosts. Each of these functions depends on the rate of distributed transactions a server executes. The capacity function can be formalized as:

$$c(s,A,F) = f_{q(s,A)}(d_s(A,F)) \tag{3}$$

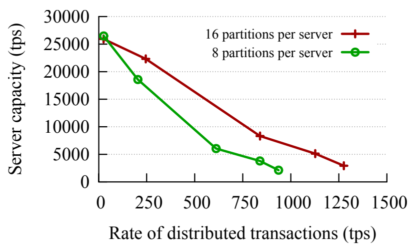


Figure 4: Server capacity with arbitrary affinity (TPC-C with varying multi-partition transactions rate).

where $q(s,A) = |\{p \in P : A_{ps} = 1\}|$ is the number of partitions hosted by server s and P is the set of partitions in the database.

For arbitrary affinity, we run new experiments in which we vary the rate of distributed transactions executed by a server. For these experiments we use TPC-C, since it has multi-partition transactions, some of which are distributed, and we change the rate of distributed transactions by modifying the fraction of multi-partition transactions in the benchmark. Figure 4 shows the variation in server capacity when we vary the rate of distributed transactions in a setting with 4 servers, each hosting 8 or 16 TPC-C partitions.

The results give two important insights. First, the shape of the capacity curve depends on the number of partitions a server hosts. A server with more partitions can execute transactions even if some of these partitions are blocked by distributed transactions. If a server with 8 cores runs 16 partitions, it is able to utilize its cores even if some of its partitions are blocked by distributed transactions. Therefore, the capacity drop is not as strong as with 8 partitions. The second insight is that the relationship between the rate of distributed transactions and the capacity of a server is not necessarily linear. For example, with 8 partitions per server, approximating the curve with a linear function would overestimate capacity by almost 25% if there are 600 distributed transactions per second.

4.3 Estimating the Capacity Function

The server capacity estimator component of Accordion estimates the function $c(s,A,F)$ online, by measuring transaction rate and transaction latency for each server. A *configuration* is a set of inputs tuples (s,A,F) that c maps to the same capacity value. The definition of a configuration depends on the affinity class of the workload. With null affinity, c returns a single capacity, so there is only one configuration for all values of (s,A,F). With uniform affinity, c returns a different value depending on the number of partitions of a server, so a configuration includes all input tuples where s hosts the same number of partitions according to A. With arbitrary affinity, every tuple in (s,A,F) represents a different configuration.

Accordion maintains the capacity function by collecting *capacity estimates* for each configuration. Initially, when no estimate is available, Accordion returns for every configuration an optimistic (i.e., high) bound that is provided, as a rough estimate, by the DBA. Over time, Accordion estimates the values of the capacity function for all configurations and refines the DBA estimate with actual observed capacity. Accordion also allows the DBA to specify a maximum number of partitions per server beyond which capacity drops to zero, although we do not use this feature in this paper.

Let c be the current configuration of a server s. Whenever latency exceeds a pre-defined service level objective (SLO) for a server s, Accordion considers the current total transaction rate of s (considering all transaction of all types) as an *estimate e* of the server capacity for c. In our experiments, the SLO is exceeded if the per-second average latency exceeds 100 milliseconds for more than 30 consecutive seconds, but different bounds, as for example 95th percentile latency, can be used. If the monitoring component of Accordion is continuously active, it can measure capacity (and activate reconfigurations) before latency and throughput degrade. Accordion currently supports DBMSes providing a single SLO for all applications. We leave support for multiple applications with different QoS guarantees as future work.

The capacity function c is continuously adjusted with the current estimates. With null and uniform affinity, the output of c for a given configuration is the average of all estimates for that configuration. For arbitrary affinity, Accordion keeps separate capacity functions based on the number of partitions a server hosts. All the estimates for configurations where a server hosts $q(s,A) = n$ partitions are used to determine the capacity function f_n of Equation 3. The partition placement requires that this function be approximated as a piecewise linear function, as discussed in Section 5.

5. PARTITION PLACEMENT MAPPER

This section discusses the partition placement mapper component of Accordion. We formulate the partition placement problem as a Mixed Integer Linear Programming (MILP) problem. MILP formulations can be solved using robust optimization tools that efficiently explore the complex space of possible reconfigurations. In our experiments the MILP solver was always able to find a solution if the size of the problem is feasible. Non-linear solvers, on the other hand, can fail in complex ways and often require human supervision to find a solution, so having a linear formulation is important for online reconfiguration.

5.1 General Algorithm

Accordion runs multiple times during the database lifetime. It can be invoked periodically or whenever the workload varies significantly. An invocation of Accordion is called a *decision point* t. We use the superscript t to denote variables and measurements for decision point t. At decision point t, the algorithm runs one or more instances of a MILP problem, each having a fixed number of servers N^t. We speed up partition placement by running multiple MILP instances in parallel for different values of N^t: if the previous number of servers is N^{t-1}, we run instances for $N^t = N^{t-1} - k, \ldots, N^{t-1}$ servers in parallel if the overall load is decreasing and $N^t = N^{t-1}, \ldots, N^{t-1} + k$ if the load is increasing. We select the solution with the lowest number of servers (i.e. lowest k), if present, and we consider a new batch of $k+1$ MILP instances with decreasing (resp. increasing) number of servers otherwise. We set $k = 1$ since in our experiments scale out and scale in operations entail adding or removing at most one server at a time. If the MILP solver terminates without finding a feasible solution, this can be an indicator that the database partitions need to be redefined.

The performance of Accordion was acceptable using our simple static MILP formulation. Given the incremental nature of reconfigurations, the problem can be cast as an online optimization problem if it is necessary to scale to larger problems.

The MILP problem for decision point t and a given number of servers N^t is formalized in Equation 4. At decision point t, Accordion calculates a new placement A^t based on the previous placement A^{t-1}. The optimization goal is to minimize the amount of data moved for the reconfiguration; m_p^t is the memory size of partition p and $S = \max(N^{t-1}, N^t)$. The first constraint expresses the throughput capacity of a server where r_p^t is the rate of transactions accessing partition p. It uses the server capacity function $c(s,A,F)$ defined in Section 4. The second constraint guarantees that the memory M of a server is not exceeded. This also places a limit on the number of partitions on a server, which counterbalances the desire to place many partitions on a server to minimize distributed transactions. The third constraint ensures that every partition is replicated k times. Some systems can be configured so that every partition is replicated a certain number of times for durability. The last two constraints express that the *first* N^t servers must be used. This is stricter than necessary but it helps reduce the solution time.

$$\begin{aligned}
\textit{minimize} \quad & \sum_{p=1}^{P}\sum_{s=1}^{S}(|A^t_{ps}-A^{t-1}_{ps}|\cdot m^t_p)/2 \\
s.t. \quad & \forall s\in[1,S]: \sum_{p=1}^{P} A^t_{ps}\cdot r^t_p < c(s,A,F) \\
& \forall s\in[1,S]: \sum_{p=1}^{P} A^t_{ps}\cdot m^t_p < M \\
& \forall p\in[1,P]: \sum_{s=1}^{S} A^t_{ps} = k \\
& \forall P\in[1,N^t]: \sum_{s=1}^{S} A^t_{ps} > 0 \\
& \forall P\in[N^t+1,S]: \sum_{s=1}^{S} A^t_{ps} = 0
\end{aligned} \tag{4}$$

The input parameters r^t and m^t are provided by the monitoring component of Accordion. The server capacity function $c(s,A,F)$ is provided by the capacity estimator.

One source of non-linearity in this problem formulation is the absolute value $|A^t_{ps}-A^{t-1}_{ps}|$ in the objective function. We make the formulation linear by introducing a new decision variable y that replaces $|A^t_{ps}-A^{t-1}_{ps}|$ in the objective function, and by adding two constraints of the form $A^t_{ps}-A^{t-1}_{ps}-y\leq 0, -(A^t_{ps}-A^{t-1}_{ps})-y\leq 0$.

5.2 Modelling Non-Linear Capacity Functions

In workloads with no distributed transactions and null affinity, the server capacity function $c(s,A,F)$ is equal to a constant $\bar{c}$ (Equation 1) so Equation 4 represents a MILP problem. If we consider distributed transactions, however, the throughput capacity function and the associated capacity constraint become non-linear. As we have previously discussed, solving non-linear optimization problems is significantly more complex and less reliable than solving linear problems. Hence, a linear formulation is important for Accordion in order to obtain partition placements automatically and online. This section describes how we formulated the server capacity constraint in a linear form with distributed transactions.

5.2.1 Uniform Affinity

In workloads with uniform affinity, the capacity of a server is a function of the number of partitions the server hosts. Therefore, we express c in the MILP formulation as a function of the new placement A^t (see Equation 2). If we substitute $c(s,A,F)$ in the first constraint of Equation 4 using the expression of Equation 2, we obtain the following load constraint:

$$\forall s\in[1,S]: \sum_{p=1}^{P} A^t_{ps}\cdot r^t_p \leq f(|\{p\in P: A^t_{ps}=1\}|) \tag{5}$$

where the function $f(q)$, which is provided as input by the server capacity estimator component of Accordion, returns the maximum throughput of a server hosting q partitions.

The function $f(q)$ is neither linear nor continuous. We express the load constraint of Equation 5 in linear terms in our MILP formulation by using a set of binary indicator variables z^t_{qs} indicating the number of partitions hosted by servers: given a server s, z^t_{qs} is 1 if and only if s hosts exactly q partitions. In a MILP formulation, the only way to assign values to variables is through a set of constraints. Therefore, we add the following constraints to Equation 4:

$$\begin{aligned}
\forall s\in[1,S]: \sum_{q=1}^{P} z^t_{qs} &= 1 \\
\forall s\in[1,S]: \sum_{p=1}^{P} A^t_{ps} &= \sum_{q=1}^{P} q\cdot z^t_{qs}
\end{aligned}$$

The first equation mandates that, given a server s, exactly one of the variables z^t_{qs} has value 1. The second equation has the number of partitions hosted by s on its left hand side, so if the server hosts q' partitions then $z^t_{q's}$ will be equal to 1.

We can now reformulate the capacity constraint of Equation 5 by using the indicator variables to select the correct capacity bound:

$$\forall s\in[1,S]: \sum_{p=1}^{P} A^t_{ps}\cdot r^t_p \leq \sum_{q=1}^{P} f(q)\cdot z^t_{qs}$$

The function $f(q)$ gives the capacity bound for a server with q partitions. It follows from the definition of the indicator variables z^t_{qs} that if a server s hosts q' partitions then the sum at the right hand side of the equation will be $f(q')$, as expected.

5.2.2 Arbitrary Affinity

For workloads where affinity is arbitrary, it is important to place partitions that are more frequently accessed together on the same server because this can substantially increase capacity, as shown in Figure 4. We must thus reformulate the first constraint of Equation 4 by substituting $c(s,A,F)$ according to Equation 3:

$$\forall s\in[1,S]: \sum_{p=1}^{P} A^t_{ps}\cdot r^t_p \leq f_{q(s,A^t)}(d^t_s(A^t,F^t)) \tag{6}$$

where $q(s,A^t)$ is the number of partitions hosted by the server s according to A^t. The rate of distributed transactions for server s, d^t_s, must be computed by the MILP formulation itself since its value depends on the output variable A^t. We aim at expressing the non-linear function d^t_s in linear terms.

Determining the distributed transactions rate. Since we want to count only distributed transactions, we need to consider only the entries of the affinity matrix related to partitions that are located on different servers. Consider a server s and two partitions p and q. If only one of the partitions is hosted by s, s has the overhead of executing the distributed transactions accessing p and q. Accordion marks the partitions requiring distributed transactions using a binary three dimensional *cross-server matrix* C^t, where $C^t_{psq}=1$ if and only if partitions p and q are mapped to different servers in the new placement A^t but at least one of them is mapped to server s:

$$C^t_{psq} = A^t_{ps}\oplus A^t_{qs}$$

were the *exclusive or* operator $\oplus$ is not linear. Instead of using the non-linear exclusive or operator, we define the value of C^t_{psq} in the context of the MILP formulation by adding the following linear constraints to Equation 4:

$$\begin{aligned}
\forall p,q\in[1,P], s\in[1,S]: \quad C^t_{psq} &\leq A^t_{ps}+A^t_{qs} \\
\forall p,q\in[1,P], s\in[1,S]: \quad C^t_{psq} &\geq A^t_{ps}-A^t_{qs} \\
\forall p,q\in[1,P], s\in[1,S]: \quad C^t_{psq} &\geq A^t_{qs}-A^t_{ps} \\
\forall p,q\in[1,P], s\in[1,S]: \quad C^t_{psq} &\leq 2-A^t_{ps}-A^t_{qs}
\end{aligned}$$

The affinity matrix F^t and the cross-server matrix are sufficient to compute the rate of distributed transactions per server s as:

$$d_s^t = \sum_{p,q=1}^{P} C_{psq}^t \cdot F_{pq}^t \tag{7}$$

Expressing the load constraint in linear terms. As discussed in Section 4, the capacity bound in the presence of workloads with arbitrary affinity can be expressed as a set of functions. If a server s hosts q partitions, its capacity is modeled by the function $f_q(d_s^t)$, where the rate of distributed transactions of s, d_s^t, is the independent variable (see Equation 3). The server capacity component of Accordion approximates each function $f_q(d_s^t)$ as a continuous piecewise linear function. Consider the sequence of delimiters u_i, with $i \in [0,n]$, that determine the boundary values of d_s^t corresponding to different segments of the function. Since d_s^t is non negative, we have $u_0 = 0$ and $u_n = C$, where C is an approximate, loose upper bound on the maximum transaction rate a server can ever reach. The reason for assuming a finite bound C will be discussed shortly. Each piecewise capacity function $f_q(d_s^t)$ is defined as follows:

$$f_q(d_s^t) = a_{iq} \cdot d_s^t + b_{iq} \quad \text{if } u_{i-1} \le d_s^t < u_i \text{ for some } i > 0$$

For each value of q, the server capacity component provides as input to the partition placement mapper an array of constants a_{iq} and b_{iq}, for $i \in [1,n]$, to describe the segments of the capacity function $f_q(d_s^t)$. We assume that $f_q(d_s^t)$ is non decreasing, so all a_{iq} are smaller than or equal to 0. This is equivalent to assuming that the capacity of a server does not increase when its rate of distributed transaction increases.

The capacity function provides an upper bound on the load of a server. If the piecewise linear function $f_q(d_s^t)$ is concave (i.e., the area above the function is concave) or linear, we could simply bound the capacity of a server to the *minimum* of all linear functions determining the segments of $f_q(d_s^t)$. This could be done by replacing the current load constraint with the following constraint:

$$\forall s \in [1,S], i \in [1,n] : \sum_{p=1}^{P} A_{ps}^t \cdot r_p^t \le a_i \cdot d_s^t + b_i$$

However, the function $f_q(d_s^t)$ is not concave or linear in general. For example, the capacity function of Figure 4 with 8 partitions is convex. If we would take the minimum of all linear functions constituting the piecewise capacity bound $f_q(d_s^t)$, as done in the previous equation, we would significantly underestimate the capacity of a server: the capacity would already go to zero with $d_s^t = 650$ due to the steepness of the first piece of the function.

We can deal with convex functions by using binary indicator variables v_{si} indicating the specific linear piece of the capacity function we need to consider for a server s given its current distributed transaction rate d_s^t. Formally, v_{si} is equal to 1 if and only if $d_s^t \in [u_{i-1}, u_i]$. Since we are using a MILP formulation, we need to define these variables through constraints as follows:

$$\begin{aligned}
\forall s \in [1,S] \quad &: \quad \sum_{i=1}^{n} v_{si} = 1 \\
\forall s \in [1,S], i \in [1,n] \quad &: \quad d_s^t \ge u_{i-1} - u_{i-1} \cdot (1 - v_{si}) \\
\forall s \in [1,S], i \in [1,n] \quad &: \quad d_s^t \le u_i + (C - u_i) \cdot (1 - v_{si})
\end{aligned}$$

In these expressions, C can be arbitrarily large, but a tighter upper bound improves the efficiency of the solver because it reduces the solution space. We set C to be the optimistic capacity value provided by the DBA (see Section 4.3).

The first constraint we added mandates that exactly one of the indicators v_{si} has to be 1. If $v_{si'}$ is equal to 1 for some $i = i'$, the next two inequalities require that $d_s^t \in [u_{i'-1}, u_i]$. The constraints for every other $i \ne i'$ are trivial since they just state that $d_s^t \in [0, C]$.

We can use the indicator variables z_{qs} defined in Section 5.2.1, for which z_{qs} = 1 if server s hosts q partitions, to select the correct function f_q for server s, and the new indicator variables v_{si} to select the right piece i of f_q to be used in the constraint. A straightforward specification of the load constraint of Equation 7 would use the indicator variables as factors, as in the following form:

$$\forall s \in [1,S] : \sum_{p=1}^{P} A_{ps}^t \cdot r_p^t \le \sum_{q=1}^{P} z_{qs} \cdot \Big(\sum_{i=1}^{n} v_{si} \cdot (a_{iq} \cdot d_s^t + b_{iq}) \Big)$$

However, z_{qs}, v_{si} and d_s^t are all variables derived from A^t, so this expression is polynomial and thus non-linear.

Since load constraints are upper bounds, our formulation considers all linear constraints for all the piecewise linear functions f_q and all their constituting linear segments, and use the indicator variables to make these constraint trivially met when they are not selected. The load constraint can thus be expressed as follows:

$$\begin{aligned}
&\forall s \in [1,S], q \in [1,P], i \in [1,n] : \\
&\qquad \textstyle\sum_{p=1}^{P} A_{ps}^t \cdot r_p^t \le \quad C \cdot (1 - a_{iq}) \cdot (1 - v_{si}) \\
&\qquad\qquad\qquad\qquad\quad + C \cdot (1 - a_{iq}) \cdot (1 - z_{qs}) \\
&\qquad\qquad\qquad\qquad\quad + a_{iq} \cdot d_s^t + b_{iq}
\end{aligned}$$

For example, if a server s' hosts q' partitions, its capacity constraint if given by the capacity function $f_{q'}$. Let us assume that the rate of distributed transactions of s lies in segment i', i.e., $d_{s'}^t \in [u_{i'-1}, u_{i'}]$. The indicator variables activate the correct constraint for s' since $v_{s'i'} = 1$ and $z_{q's'} = 1$ so the constraint for server s' becomes:

$$\sum_{p=1}^{P} A_{ps'}^t \cdot r_p^t \le a_{i'q'} \cdot d_{s'}^t + b_{i'q'}$$

which selects the function $f_{q'}(d_{s'}^t)$ and the segment i' to express the capacity bound of s'. All remaining constraints become trivial due to the indicators. In fact, for all values of $q \ne q'$ and $i \ne i'$ it holds that either $v_{s'i'} = 0$ or $z_{q's'} = 0$, so the constraints just say that the load of a server is lower than the upper bound C, which is true by definition. This holds since all functions $f_q(d_s^t)$ are non-increasing, so $a_{iq} \le 0$ and $C \cdot (1 - a_{iq}) + a_{iq} \cdot d_s^t = C - a_{iq}(C - d_s^t) \ge C$.

Implicit clustering. In the presence of arbitrary affinity, the placement should cluster affine partitions together and try to place each cluster on a single server. Executing clustering as a pre-step could be enticing but is not viable since clustering and placement should be solved together: since clusters of partitions are to be mapped onto a single server, the definition of the clusters needs to take into consideration the load on each partition, the capacity constraints of the server that should host the partition, and the migration costs of transferring all partitions to the same server if needed.

Our MILP formulation implicitly lets the solver place partitions with high affinity onto the same server. In fact, the solver explores all feasible solutions for a given number of servers searching for the one that minimizes data migration. One way to minimize data migration is to maximize the capacity of each server, which is done by placing partitions with high affinity onto the same server. Therefore, Accordion does not need to run an external partition clustering algorithm as a pre-processing step.

In summary, we were able to express the complex interdependencies involved in partition placement as linear constraints of a MILP problem. This problem can be solved effectively by standard optimization tools, which enables Accordion to reliably obtain optimal partition placements online.

6. EVALUATION

6.1 Experimental Setup

All our experiments use two workloads, TPC-C and YCSB. We run the workloads on H-Store, using the cluster specifications described in Section 4: each server has two Intel Xeon quad-core processors running at 2.67GHz and 32 GB of RAM. The servers are connected to a 10Gb Ethernet network. The server capacity functions we use in our experiments are the ones obtained in Section 4. We used the popular CPLEX optimization tool to efficiently solve the MILP problem.[1] The controller node used to run the placement algorithms has two Intel Xeon processors, each with 12-cores running at 2.40 GHz, and 54GB of RAM.

We used databases of different sizes, split into 64 to 1024 partitions. Our experiments shows that Accordion efficiently scales up to 1024 partitions regardless of the partition size, since our problem complexity is a function of only the number of partitions and servers. Hence, Accordion can handle very large databases as long as the total size of distributed memory is sufficient for the database. In our experiments we want to stress-test the partition placement algorithms, so we make sure that placements are not ruled out just because some server becomes memory bound. For migration, Accordion uses a simple live migration system for H-Store inspired by Zephyr [12]. H-Store support for replication is still experimental, so we do not require partitions to be replicated.

For our experiments, we need a fluctuating workload to drive the need for reconfiguration. The fluctuation in overall intensity (in transactions per second) of the workload that we use follows the access trace of Wikipedia for a randomly chosen day, October 8th, 2013. In that day, the maximum load is 50% higher than the minimum. We repeat the trace, so that we have a total workload covering two days. The initial workload intensity was chosen to be high enough to stress the placement algorithms by requiring frequent reconfigurations. We run reconfiguration periodically, every 5 minutes, and we report the results for the second day of the workload (representing the steady state). We skew the workload such that 20% of the transactions access "hot" partitions and the rest access "cold" partitions. The set of hot and cold partitions is changed at random in every reconfiguration interval.

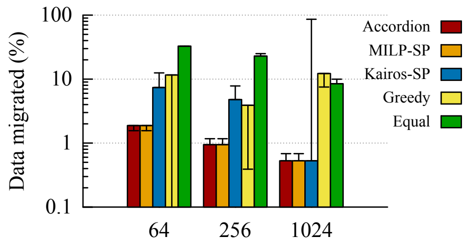
6.2 Baselines for Comparison

The *Equal* partitioning algorithm models the behaviour of typical hash- or range-based partitioning placements that assign an equal number of partitions to every server. Equal uses a single constant server capacity bound, and uses the minimum number of servers sufficient to support the current workload. For scaling out or in, Equal moves the minimal number of partitions per server to reach an equal placement. For all the baselines, we use a capacity bound equal to the maximum server throughput in the standard configuration with one partition per core.

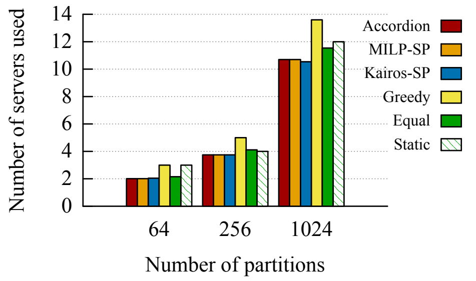
It is common practice in distributed data stores and DBMSes to provision for peak load and assign an equal amount of data to each server. We call this a *Static* policy because it does not require data migration. Our static policy uses the maximum number of servers used by Equal over all reconfigurations. This is a best-case static configuration because it assumes knowledge of online workload dynamics that might not be known a priori, when a static configuration is devised.

The *Greedy* best fit heuristic explicitly takes skew into account. Heuristics similar to Greedy have been applied in elastic key-value stores, whose workloads can be implemented in a DBMS using only single partition transactions [26]. Greedy labels servers as overloaded or underloaded based on how close their load is to their capacity. First, Greedy migrates the hottest partitions of overloaded servers until no server is overloaded. Second, it tries to migrate all partitions out of the most underloaded servers in order to reduce the overall number of servers. Every time a partition is migrated, the receiving server is the underloaded server that currently has the highest load and enough capacity to host the partition. If no such server is available in the cluster when migrating partitions from an overloaded server, Greedy adds a new server. Greedy uses a single constant server capacity bound, like Equal.

MILP-SP uses the MILP formulation of Accordion that assumes null affinity. MILP-SP represents an adaptation of the tenant placement algorithm of Kairos [9] to partition placement. Like Kairos, MILP-SP is not aware of distributed transactions, so it uses a single capacity bound like the other heuristics. *Kairos-PP* is similar to MILP-SP but it minimizes load imbalance, like the original Kairos tenant placement algorithm, instead of data movement.



(a) Data migrated per reconfiguration (logscale)



(b) Average number of servers used

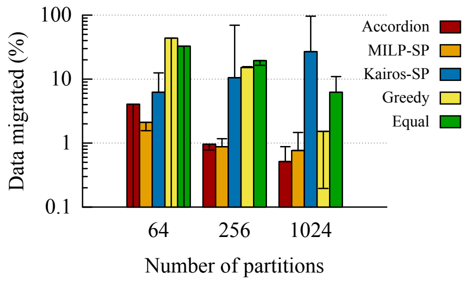
Figure 5: Partition placement - workload with null affinity (YCSB)

6.3 Elasticity with Null Affinity

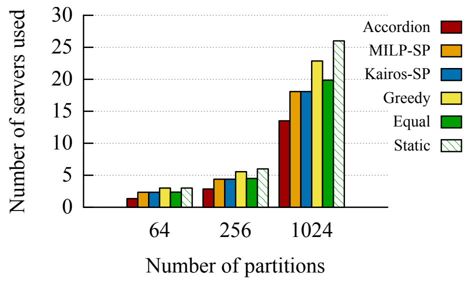
We start our evaluation by considering YCSB, a workload where all transactions access only a single partition. We considered YCSB instances with 64, 256, and 1024 partitions. Depending on the number of partitions, initial loads range from 40,000 to 240,000 transactions per second. We report two metrics: the amount of data each algorithm migrates and the number of servers it requires. We do not report throughput figures because in our experiments all algorithms were able to adapt to the offered load as needed. That is, all algorithms have similar throughput.

With null affinity, Accordion and MILP-SP are the same algorithm and therefore have the same performance, as shown in Figure 5. The colored bars in Figure 5(a) represent the average data migrated, and the vertical lines represent the 95th percentile. Figure 5(b) represents the average number of servers used. Both Accordion and MILP-SP migrate a very small fraction of partitions: the average and 95th percentile are below 2%. They use 2 servers

[1]Available under `http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/`

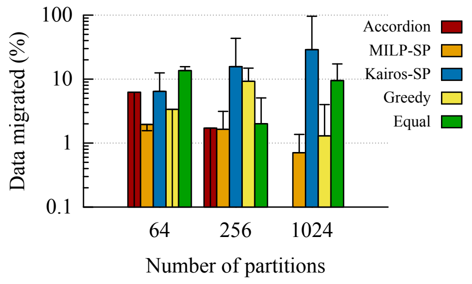


(a) TPC-C: Data migrated per reconfiguration (logscale)

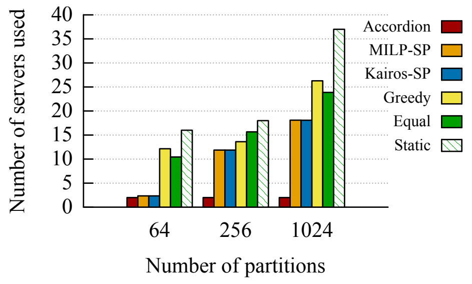


(b) TPC-C: Average number of servers used

Figure 6: Partition placement - TPC-C with uniform affinity



(a) Data migrated per reconfiguration (logscale)



(b) Average number of servers used

Figure 7: Partition placement - TPC-C with arbitrary affinity

on average with 64 partitions, 3.7 with 256 partitions, and 10.7 with 1024 partitions. Kairos-PP moves between 4% and 10.5% of the database on average and uses approximately the same average number of servers as Accordion and MILP-SP (for 1024 partitions, there is a slight reduction to 10.54 servers on average). Even though Equal and Greedy are designed for single-partition transactions, they still perform worse than Accordion. The Equal placement algorithm uses a slightly higher number of servers on average than Accordion, but it migrates between 16x and 24x more data than Accordion on average, with very high 95th percentiles. Greedy migrates slightly less data than Equal, but uses a factor between 1.3x and 1.5x more servers than Accordion, and barely outperforms the Static policy.

Our results show the advantage of using MILP formulations, such as Accordion or Kairos-PP, over heuristics in workloads without distributed transactions. No heuristic can achieve the same quality in trading off the two conflicting goals of minimizing the number of servers and the amount of data migrated. The Greedy heuristic induces fewer migrations, but it cannot effectively aggregate the workload onto fewer servers. The Equal heuristic aggregates more aggressively at the cost of more migrations.

6.4 Elasticity with Uniform Affinity

This experiment considers a workload, like TPC-C, having distributed transactions and uniform affinity. The initial transaction rates are 9,000, 14,000, and 46,000 tps for configurations with 64, 256 and 1024 partitions, respectively. Figure 6(a) reports the average fraction of partitions moved in all reconfiguration steps in the TPC-C scenario and also the 95th percentile. As expected, since Equal and Greedy are not designed to deal with distributed transactions, Accordion achieves higher gains with TPC-C than with YCSB. It migrates less than 4% in the average case, while the other heuristics migrate significantly more data.

MILP-SP migrates a similar number of partitions as Accordion because it is less aggressive in minimizing the number of servers. However, it uses between 1.3x to 1.7x more servers on average than Accordion. Accordion also outperforms all other algorithms in terms of number of servers used, as shown in Figure 6(b). Like in the previous case, Greedy does not significantly outperform the Static policy.

Accordion takes into account distributed transactions and estimates the capacity boost of co-locating partitions on fewer servers. This explains why Accordion achieves better resource utilization than the baseline placement algorithms, which assume a constant capacity bound.

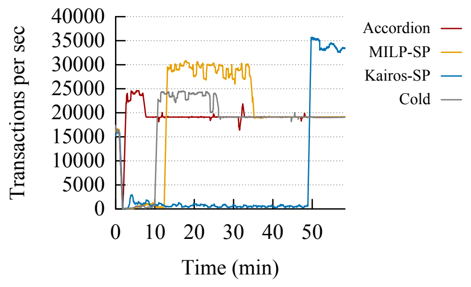
6.5 Elasticity with Arbitrary Affinity

This section considers workloads with arbitrary affinity. We modify TPC-C to bias the affinity among partitions: each partition belongs to a cluster of 4 partitions in total. Transactions accessing partitions across clusters are 1% of the multi-partition transactions. For Equal and Greedy, we select an average capacity bound that corresponds to a random distribution of 8 partitions to servers.

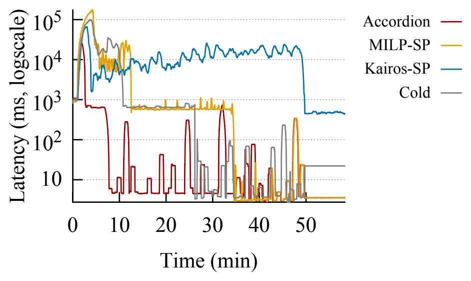
Figure 7 reports our results with an initial transaction rate of 40000 tps. The results show the highest gains using Accordion across all the workloads we considered. Accordion manages to reduce the average number of servers used by a factor of more than 1.3x with 64 partitions and of more than 9x with 1024 partitions, with a 17x gain compared to Static.

The significant cost reduction achieved by Accordion is due to its implicit clustering. By placing together partitions with high affinity, Accordion boosts the capacity of the servers and thus needs fewer servers to support the workload. The results also show that Accordion is able to leverage the additional flexibility given by the use of a larger number of partitions.

Accordion is robust enough to keep the same margin in a variety of situations. We have repeated these experiments with 256 partitions and varied *(i)* the fraction of multi-partition transactions between 5% and 20%, *(ii)* the size of affinity clusters between 4 and 16 partitions, and *(iii)* the fraction of inter-cluster multi-partition transactions from 0.1% to 10%. In all cases, we got very similar results as the ones of Figure 7.



(a) Throughput



(b) Latency

Figure 8: Performance during TPC-C reconfiguration.

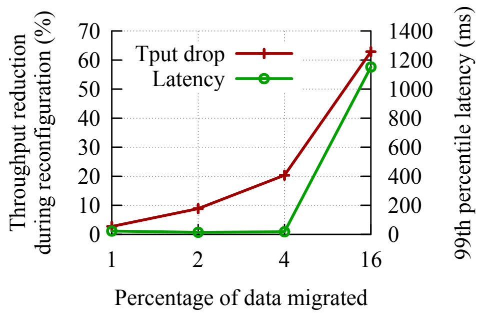


Figure 9: Effect of migrating fractions of the YCSB database.

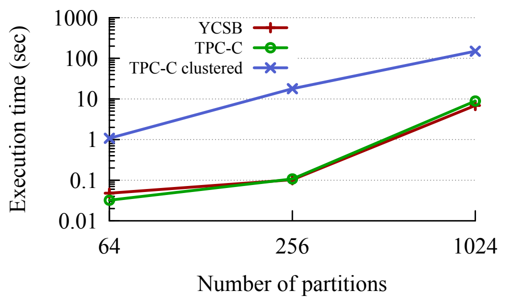


Figure 10: Execution time of Accordion

6.6 Cost of Data Migration

We now evaluate the transient disruption caused by reconfigurations. We used a TPC-C database of 33GB split into 256 partitions. Figure 8 shows the effect of one reconfiguration on the throughput and latency of TPC-C using multiple planners, all starting from the same initial configuration of 3 servers with one overloaded server. The plots report the second-by-second moving average of the last 50 seconds. The planners are Accordion, MILP-SP, and Kairos-PP. Accordion migrates the minimal amount of data needed to offload a certain amount of transactions per second (tps) from a server. This implicitly favors moving fewer hot partitions over more cold partitions. We also consider a planner called "Cold" that transfers the same amount of tps among each pair of servers as Accordion but it moves a larger number of partitions because it selects the coldest ones.

All planners go through two phases: (*i*) a migration phase, which is characterized by very low throughput, and (*ii*) a catch up phase after reconfiguration has terminated. The catch up phase has higher throughput but also high latency because clients submit at a very high rate all requests that were blocked during migration. After catch up terminates, all planners converge to the same throughput, which is about 20% higher than the initial one. The figure does not show that the number of servers used by the planners is different: Accordion and Cold simply migrate partitions among the 3 existing servers, whereas MILP-SP and Kairos-SP scale out to 5 servers.

The impact of large data migrations is evident in Figure 8. All planners start reconfiguration after one minute into the run. Accordion moves 1.2% of the data, MILP-SP 15.7%, Cold 18.8%, and Kairos-SP 94.1%. The migration phase of Accordion is very short, less than 1 minute. MILP-SP and Cold have migration phases of about 10 minutes. Kairos-SP, which does not try to minimize data movement, has a very long migration of about 50 minutes.

The comparison between Accordion and Cold shows experimentally the advantage of moving hot partitions. To understand the reason for this advantage, consider the example of an overloaded server that needs to offload 1000 transactions per second (tps). Assume that each partition can be migrated in 1 second. If a hot partition receives 1000 tps, it is sufficient to migrate only that partition. If we disregard the cost of setting up and coordinating a reconfiguration, the migration could potentially block or abort 1000 transactions. The exact numbers of blocked and aborted transactions will depend on the specific migration mechanisms, for example if it is stop-and-copy or live. If we decide to migrate 10 cold partitions receiving 100 tps each, the cost of each migration will be smaller, but the reconfiguration will take 10 seconds, so we will again block or abort 1000 transactions. In practice, however, longer reconfigurations end up being less effective as shown by our experiments.

We have also tested the impact of reconfiguration on YCSB. Figure 9 illustrates the throughput and latency during live migration of different amounts of data. If less than 2% of the database is migrated, the throughput reduction is minor (less than 10%), but it starts to be noticeable (20%) when 4% of the database or more is migrated, and it becomes major (60%) when transferring 16% of the database.

6.7 Execution Time of Accordion

Accordion must be able to quickly find new partition placements to support online elasticity. The simple heuristics, Equal and Greedy, execute in a very short time: even in the case of 1024 partitions, they find placements in less than 50 milliseconds. Accordion is slower because it uses CPLEX to solve a non-trivial MILP problem but it is fast enough to be used in an online setting. The average execution time of one decision point of Accordion is shown in Figure 10. For YCSB, which has null affinity, and for TPC-C, which has uniform affinity, the execution time is smaller than 10 seconds. For workloads with arbitrary affinity, like TPC-C with clustered partitions, execution time goes up to almost 2 minutes for 1024 partitions. However, this longer execution time pays off very well in terms of saved costs, as shown in Figure 7. Note that 2 minutes is acceptable in many cases where the workload does not fluctuate frequently and reconfiguration is infrequent.

7. CONCLUSION

Dynamic partition placement with distributed transactions is an important problem, and will become more important as different DBMSes add elasticity mechanisms and migrate to pay-as-you-go clouds. It is a complex problem consisting of many interdependent subproblems: finding a good partition placement that supports the offered load, determining the capacity of each server with a varying placement and affinity, considering skew in workloads that may not be known a priori, and minimizing the amount of data migrated. Accordion successfully addresses all these problems using an online server capacity estimation and a MILP formulation that can be efficiently solved online. Our results show that using accurate server capacity models for distributed transactions significantly improves the quality of the placements.

As future work, we plan to combine our capacity model for distributed transactions with other types of performance models, in order to cover also disk-based systems. We also plan to consider elasticity mechanisms that, instead of considering partitions as pre-defined, split and merge partitions at run-time based on load.

8. REFERENCES

[1] P. M. G. Apers. Data allocation in distributed database systems. *Trans. on Database Systems (TODS)*, 13(3), 1988.

[2] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *ACM SIGMOD Conference*, 2014.

[3] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*, 2011.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. Symposium on Cloud Computing (SoCC)*, 2010.

[5] G. P. Copeland, W. Alexander, E. E. Boughter, and T. W. Keller. Data placement in Bubba. In *Proc. SIGMOD Int. Conf. on Management of Data*, 1988.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally-distributed database. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[7] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proc. USENIX Annual Technical Conf. (ATC)*, 2012.

[8] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow. (PVLDB)*, 3(1-2), 2010.

[9] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *Proc. SIGMOD Int. Conf. on Management of Data*, 2011.

[10] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: an elastic transactional data store in the cloud. In *Proc. HotCloud*, 2009.

[11] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow. (PVLDB)*, 4(8), 2011.

[12] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proc. SIGMOD Int. Conf. on Management of Data*, 2011.

[13] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant DBMSs. In *Proc. SIGMOD Int. Conf. on Management of Data*, 2013.

[14] K. A. Hua and C. Lee. An adaptive data placement scheme for parallel database computer systems. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1990.

[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. *Proc. VLDB Endow. (PVLDB)*, 1(2), 2008.

[16] J. Lee, Y. S. Kwon, F. Farber, M. Muehle, C. Lee, C. Bensberg, J. Y. Lee, A. H. Lee, and W. Lehner. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2013.

[17] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal*, 6(1), 1997.

[18] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, J. Ng, and S. Robertson. Elastic scale-out for partition-based database systems. In *Proc. Int. Workshop on Self-managing Database Systems (SMDB)*, 2012.

[19] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proc. SIGMOD Int. Conf. on Management of Data*, 2013.

[20] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proc. SIGMOD Int. Conf. on Management of Data*, 2012.

[21] D. Saccà and G. Wiederhold. Database partitioning in a cluster of processors. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1983.

[22] J. Schaffner, T. Januschowski, M. Kercher, T. Kraska, H. Plattner, M. J. Franklin, and D. Jacobs. RTP: Robust tenant placement for elastic in-memory database clusters. In *Proc. SIGMOD Int. Conf. on Management of Data*, 2013.

[23] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2007.

[24] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proc. SIGMOD Int. Conf. on Management of Data*, 2012.

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

[26] 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 *Proc. of USENIX Conf. on File and Storage Technologies (FAST)*, 2011.