Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Carlo Curino
Yahoo! Research
krl@yahoo-inc.com

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

The advent of affordable, shared-nothing computing systems portends a new class of parallel database management systems (DBMS) for on-line transaction processing (OLTP) applications that scale without sacrificing ACID guarantees [7, 9]. The performance of these DBMSs is predicated on the existence of an optimal database design that is tailored for the unique characteristics of OLTP workloads [43]. Deriving such designs for modern DBMSs is difficult, especially for *enterprise-class* OLTP systems, since they impose extra challenges: the use of stored procedures, the need for load balancing in the presence of time-varying skew, complex schemas, and deployments with larger number of partitions.

To this purpose, we present a novel approach to automatically partitioning databases for enterprise-class OLTP systems that significantly extends the state of the art by: (1) minimizing the number distributed transactions, while concurrently mitigating the effects of temporal skew in both the data distribution and accesses, (2) extending the design space to include replicated secondary indexes, (4) organically handling stored procedure routing, and (3) scaling of schema complexity, data size, and number of partitions. This effort builds on two key technical contributions: an analytical cost model that can be used to quickly estimate the relative coordination cost and skew for a given workload and a candidate database design, and an informed exploration of the huge solution space based on large neighborhood search. To evaluate our methods, we integrated our database design tool with a high-performance parallel, main memory DBMS and compared our methods against both popular heuristics and a state-of-the-art research prototype [17]. Using a diverse set of benchmarks, we show that our approach improves throughput by up to a factor of 16× over these other approaches.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design

Keywords

OLTP, Parallel, Shared-Nothing, H-Store, KB, Stored Procedures

1. INTRODUCTION

The difficulty of scaling front-end applications is well known for DBMSs executing highly concurrent workloads. One approach to this problem employed by many Web-based companies is to *partition* the data and workload across a large number of commodity, shared-nothing servers using a cost-effective, parallel DBMS. Many of these companies have adopted various new DBMSs, colloquially referred to as *NoSQL* systems, that give up transactional ACID guarantees in favor of availability and scalability [9]. This approach is desirable if the consistency requirements of the data are "soft" (e.g., status updates on a social networking site that do not need to be immediately propagated throughout the application).

OLTP systems, especially enterprise OLTP systems that handle high-profile data (e.g., financial and order processing systems), also need to be scalable but cannot give up strong transactional and consistency requirements [27]. The only option previously available for these organizations was to purchase more powerful single-node machines or develop custom middleware that distributes queries over traditional DBMS nodes [41]. Both approaches are prohibitively expensive and thus are not an option for many.

As an alternative to NoSQL and custom deployments, a new class of parallel DBMSs, called *NewSQL* [7], is emerging. These systems are designed to take advantage of the partitionability of OLTP workloads to achieve scalability without sacrificing ACID guarantees [9, 43]. The OLTP workloads targeted by these NewSQL systems are characterized as having a large number of transactions that (1) are short-lived (i.e., no user stalls), (2) touch a small subset of data using index look-ups (i.e., no full table scans or large distributed joins), and (3) are repetitive (i.e., typically executed as pre-defined transaction templates or stored procedures [43, 42].)

The scalability of OLTP applications on many of these newer DBMSs depends on the existence of an optimal *database design*. Such a design defines how an application's data and workload is partitioned or replicated across nodes in a cluster, and how queries and transactions are routed to nodes. This in turn determines the number of transactions that access data stored on each node and how skewed the load is across the cluster. Optimizing these two factors is critical to scaling complex systems: our experimental evidence shows that a growing fraction of distributed transactions and load skew can degrade performance by over a factor 10× Hence, without a proper design, a DBMS will perform no better than a single-node system due to the overhead caused by blocking, inter-node communication, and load balancing issues [25, 37].

Many of the existing techniques for automatic database partitioning, however, are tailored for large-scale analytical applications (i.e., data warehouses) [36, 40]. These approaches are based on the notion of *data declustering* [28], where the goal is to spread data across nodes to maximize intra-query parallelism [5, 10, 39, 49]. Much of this work is not applicable to OLTP systems because the multi-node coordination required to achieve transaction consistency dominates the performance gains obtained by this type

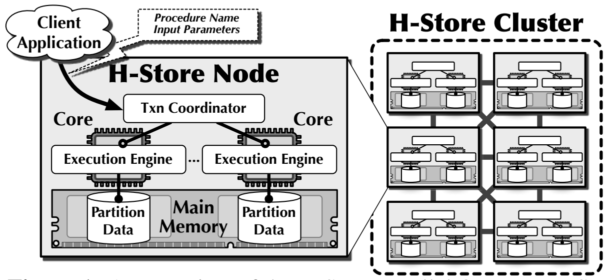


Figure 1: An overview of the H-Store parallel OLTP DBMS.

of parallelism; previous work [17, 24] has shown that, even after ignoring the affects of lock-contention, this overhead can be up to 50% of the total execution time of a transaction when compared to single-node execution. Although other work has focused on parallel OLTP database design [49, 17, 32], these approaches lack three features that are crucial for enterprise OLTP databases: (1) support for stored procedures to increase execution locality, (2) the use of replicated secondary indexes to reduce distributed transactions, and (3) handling of time-varying skew in data accesses to increase cluster load balance. These three salient aspects of enterprise databases hinder the applicability and effectiveness of the previous work. This motivates our research effort.

Given the lack of an existing solution for our problem domain, we present *Horticulture,* a scalable tool to automatically generate database designs for stored procedure-based parallel OLTP systems. The two key contributions in this paper are (1) an automatic database partitioning algorithm based on an adaptation of the *large-neighborhood search* technique [21] and (2) a new analytical cost model that estimates the coordination cost and load distribution for a sample workload. Horticulture analyzes a database schema, the structure of the application's stored procedures, and a sample transaction workload, then automatically generates partitioning strategies that minimizes distribution overhead while balancing access skew. The run time of this process is independent of the database's size, and thus is not subject to the scalability limits of existing solutions [49, 17]. Moreover, Horticulture's designs are not limited to horizontal partitioning and replication for tables, but also include replicated secondary indexes and stored procedure routing.

Horticulture produces database designs that are usable with any shared-nothing DBMS or middleware solution. To verify our work, we integrated Horticulture with the H-Store [1] parallel DBMS. Testing on a main memory DBMS like H-Store presents an excellent challenge for Horticulture because they are especially sensitive to the quality of partitioning in the database design, and require a large number of partitions (multiple partitions for each node).

We thoroughly validated the quality of our design algorithms by comparing Horticulture with four competing approaches, including another state-of-the-art database design tool [17]. For our analysis, we ran several experiments on five enterprise-class OLTP benchmarks: TATP, TPC-C (standard and skewed), TPC-E, SEATS, and AuctionMark. Our tests show that the three novel contributions of our system (i.e., stored procedure routing, replicated secondary indexes, and temporal-skew management) are much needed in the context of enterprise OLTP systems. Furthermore, our results indicate that our design choices provide an overall performance increase of up to a factor 4× against the state-of-the-art tool [17] and up to a factor 16× against a practical baseline approach.

The rest of the paper is organized as follows. In Section 2, we experimentally investigate the impact of distributed transactions and temporal workload skew on throughput in a shared-nothing, parallel OLTP system. Then in Section 3, we present an overview of Horticulture and its capabilities. In Sections 4 and 5, we discuss

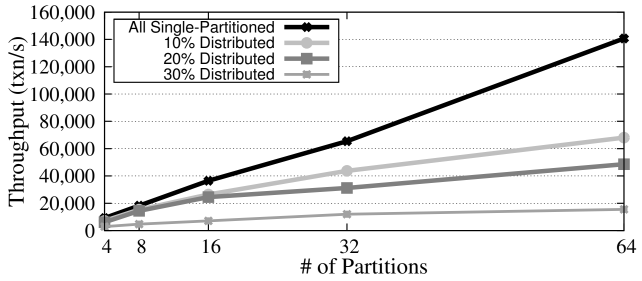


Figure 2: Impact of Distributed Transactions on Throughput

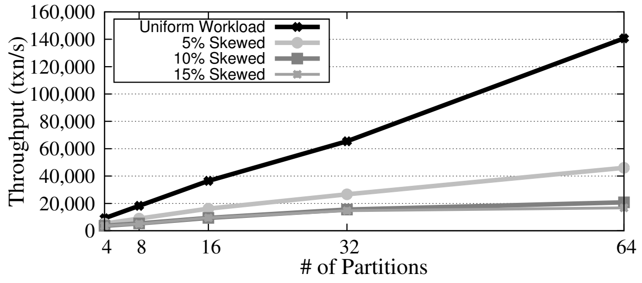


Figure 3: Impact of Temporal Workload Skew on Throughput

the two key technical contributions of this paper: (1) our algorithm to explore potential solutions and (2) our cost model. We discuss various optimizations in Section 6 that allow our tool to scale to large instances of the database design problem. Lastly, we present our experimental evaluation in Section 7.

2. OLTP DATABASE DESIGN MOTIVATION

We now discuss the two key issues when generating a database design for enterprise OLTP applications: distributed transactions and temporal workload skew.

We first note that many OLTP applications utilize stored procedures to reduce the number of round-trips per transaction between the client and the DBMS [42]. Each procedure contains *control code* (i.e., application logic) that invokes pre-defined parameterized SQL commands. Clients initiate transactions by sending the procedure name and input parameters to the cluster.

For each new transaction request, the DBMS determines which node in the cluster should execute the procedure's control code and dispatch queries. In most systems, this node also manages a partition of data. We call this the *base partition* for a transaction [37]. Any transaction that needs to access data from only its base partition is known as a *single-partition* transaction [31]. These transactions can be executed efficiently on a parallel DBMS, as they do not require multi-node coordination [43]. Transactions that need to access multiple partitions, known as *distributed transactions*, require the DBMS to employ two-phase commit or a similar distributed consensus protocol to ensure atomicity and serializability, which adds additional network overhead [25].

Whether or a not a transaction is single-partitioned is based on the physical layout of the database. That is, if tables are divided amongst the nodes such that a transaction's base partition has all of the data that the transaction needs, then it is single-partitioned.

To illustrate how the presence of distributed transactions affects performance, we executed a workload derived from the TPC-C benchmark [45] on H-Store [1], a row-storage, relational OLTP DBMS that runs on a cluster of shared-nothing, main memory-only nodes [43, 26]. We postpone the details of our experimental setting to Section 7. In each round of this experiment, we varied the number of distributed transactions and execute the workload on five different cluster sizes, with at most seven partitions assigned per node.

Fig. 2 shows that a workload mix of just 10% distributed transactions has a significant impact on throughput. The graph shows that the performance difference increases with larger cluster sizes: at 64 partitions, the impact is approximately $2\times$. This is because single-partition transactions in H-Store execute to completion in a single thread, and thus do not incur the overhead of traditional concurrency control schemes [24]. For the distributed transactions, the DBMS's throughput is limited by the rate at which nodes send and receive the two-phase commit messages. These results also show that the performance repercussions of distributed transactions increases relative to the number of partitions because the system must wait for messages from more nodes. Therefore, a design that minimizes both the number of distributed transactions and the number of partitions accessed per transaction will reduce coordination overhead, thereby increasing the DBMS's throughput [49, 17].

Even if a given database design enables every transaction to execute as single-partitioned, the DBMS may still fail to scale linearly if the application's workload is unevenly distributed across the nodes. Thus, one must also consider the amount of data and transactions assigned to each partition when generating a new database design, even if certain design choices that mitigate skew cause some transactions to be no longer singled-partitioned. Existing techniques have focused on *static skew* in the database [49, 17], but failed to consider *temporal skew* [47]. Temporally skewed workloads might appear to be uniformly distributed when measured globally, but can have a significant effect on not only performance but also availability in shared-nothing DBMSs [22].

As a practical example of temporal skew, consider Wikipedia's approach to partitioning its database by language (e.g., English, German) [2]. This strategy minimizes the number of distributed transactions since none of the common transactions access data from multiple languages. This might appear to be a reasonable partitioning approach, however the database suffers from a non-trivial amount of temporal skew due to the strong correlation between languages and geographical regions: the nodes storing the articles for one language are mostly idle when it is night time in the part of the world that speaks that language. If the data set for a particular language is large, then it cannot be co-located with another partition for articles that are mostly accessed by users from another part of the world. At any point during the day the load across the cluster is significantly unbalanced even though the average load of the cluster for the entire day is uniform. Wikipedia's current solution is to over-provision nodes enough to mitigate the skew effects, but a temporal-skew-aware database design may achieve identical performance with lower hardware and energy costs.

We also experimentally tested the impact of temporal skew on our H-Store cluster. In this experiment, we use a 100% single-partition transaction workload (to exclude distribution costs from the results) and impose a time-varying skew. At fixed time intervals, a higher percentage of the overall workload is directed to one partition in the cluster. The results are shown in Fig. 3. For large number of partitions, even when only an extra 5% of the overall load is skewed towards a single-partition, the throughput is reduced by a large factor, more than $3\times$ in our test. This is because the execution engine for the partition that is receiving a larger share of the workload is saturated, which causes other partitions to remain idle while the clients are blocked waiting for results. The latency increases further over time since the target partition cannot keep up with the increased load.

The above examples show that both distributed transactions and temporal workload skew must be taken into account when deploying a parallel database in order to maximize its performance. Manually devising optimal database designs for an arbitrary OLTP application is non-trivial because of the complex trade-offs between distribution and skew: one can enable all requests to execute as single-partitioned transactions if the database is put on a single node (assuming there is sufficient storage), but one can completely remove skew if all requests are executed as distributed transactions that access data at every partition. Hence, a tool is needed that is capable of partitioning stored procedure-based enterprise OLTP databases to balance these conflicting goals.

In the next section, we describe how we solve this problem.

3. AUTOMATIC DATABASE DESIGN

Horticulture is an automatic database design tool that selects the best physical layout for a parallel DBMS that minimizes the number of distributed transactions while also reducing the effects of temporal skew. The administrator provides Horticulture with (1) the database schema of the target OLTP application, (2) a set of stored procedures definitions, and (3) a reference workload trace. A workload trace is a log of previously executed transactions for an application. Each transaction record in the trace contains its procedure input parameters, the timestamps of when it started and finished, and the queries it executed with their corresponding input parameters. Horticulture works under the reasonable assumption that the sample trace is representative of the target application.

Using these inputs, Horticulture explores an application's solution space, where for each table the tool selects whether to (1) horizontally partition or (2) replicate on all partitions, as well as to (3) replicate a secondary index for a subset of its columns. The DBMS uses the column(s) selected in these design elements with either hash or range partitioning to determine at run time which partition stores a tuple. The tool also needs to determine how to enable the DBMS to effectively route incoming transaction requests to the partition that has most of the data that each transaction will need to access [34]. As we will discuss in this section, this last step is particularly challenging for applications that use stored procedures.

3.1 Design Options

Before discussing the specifics of our design algorithms, we first elaborate on the design options supported by Horticulture. These are based on the common assumption that OLTP transactions access tables in a hierarchical manner [43]. These options are illustrated in Fig. 4 using components from TPC-C [45].

Horizontal Partitioning: A table can be horizontally divided into multiple, disjoint fragments whose boundaries are based on the values of one (or more) of the table's columns (i.e., the *partitioning attributes*) [49]. The DBMS assigns each tuple to a particular fragment based on the values of these attributes using either range partitioning or hash partitioning. Related fragments from multiple tables are combined together into a partition [23, 35]. Fig. 4a shows how each record in the `CUSTOMER` table has one or more `ORDER` records. If both tables are partitioned on their `CUSTOMER` id, then all transactions that only access data for a single customer will execute as single-partitioned, regardless of the state of the database.

Table Replication: Alternatively, a table can be replicated across all partitions. This is different than replicating entire partitions for durability and availability. Replication is useful for read-only or read-mostly tables that are accessed together with other tables but do not share foreign key ancestors. For example, the read-only `ITEM` table in Fig. 4b does not have a foreign-key relationship with the `CUSTOMER` table. By replicating this table, transactions do not need to retrieve data from a remote partition in order to access it. Any transaction that modifies a replicated table cannot be executed as single-partitioned, since those changes must be broadcast to ev-

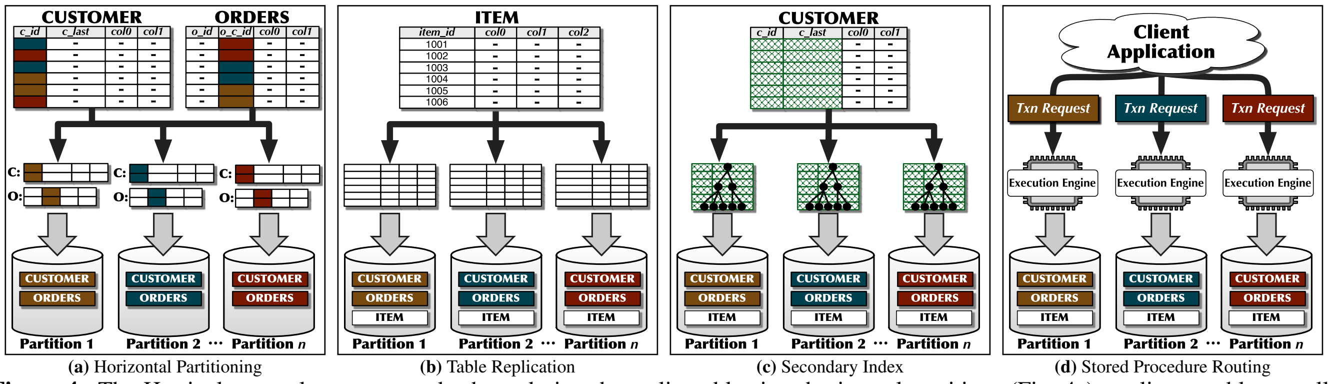


Figure 4: The Horticulture tool generates a database design that splits tables into horizontal partitions (Fig. 4a), replicates tables on all partitions (Fig. 4b), replicates secondary indexes on all partitions (Fig. 4c), and routes transaction requests to the best base partition (Fig. 4d).

ery partition in the cluster. Furthermore, given that some OLTP systems store the entire database in main memory, one must also consider the space needed to replicate a table at each partition.

Secondary Indexes: When a query accesses a table through an attribute that is not the partitioning attribute, it is broadcasted to all nodes. In some cases, however, these queries can become single-partitioned if the database includes a secondary index for a subset of a table's columns that is replicated across all partitions. Consider a transaction for the database shown in Fig. 4c that executes a query to retrieve the id of a `CUSTOMER` using their last name. If each partition contains a secondary index with the id and the last name columns, then the DBMS can automatically rewrite the stored procedures' query plans to take advantage of this data structure, thereby making more transactions single-partitioned. Just as with replicated tables, this technique only improves performance if the columns chosen in these indexes are not updated that often.

Stored Procedure Routing: In addition to partitioning or replicating tables, Horticulture must also ensure that transaction requests can be effectively routed to the partition that has the data that it will need [38]. The DBMS uses a procedure's *routing attribute(s)* defined in a design at run time to redirect a new transaction request to a node that will execute it [34]. The best routing attribute for each procedure enables the DBMS to identify which node has the most (if not all) of the data that each transaction needs, as this allows them to potentially execute with reduced concurrency control [37]. The example in Fig. 4d illustrates how transactions are routed according to the value of the input parameter that corresponds to the partitioning attribute for the `CUSTOMER` table. If the transaction executes on one node but the data it needs is elsewhere, then it must execute with full concurrency control. This is difficult for many applications, because it requires mapping the procedures' input parameters to their queries' input parameters using either a workload-based approximation or static code analysis. Potential designs that partition tables well are discarded if we are unable to generate a good routing plan for procedures.

3.2 Database Design Challenges

The problem of finding an optimal database design is known to be NP-Complete [31, 35], and thus it is not practical to examine every possible design to discover the optimal solution [49]. Even if one can prune a significant number of the sub-optimal designs by discarding unimportant table columns, the problem is still exceedingly difficult when one also includes stored procedure routing parameters—as a reference, the number of possible solutions for TPC-C and TPC-E are larger than 10^{66} and 10^{94}, respectively. Indeed, we initially developed an iterative greedy algorithm similar to the one proposed in [4], but found that it obtained poor results for these complex instances because it is unable to escape local minima. There are, however, existing search techniques from optimization research that make problems such as this more tractable.

Horticulture employs one such approach, called *large-neighborhood search* (LNS), to explore potential designs off-line in a guided manner [21, 18]. LNS compares potential solutions with a *cost model* that estimates how well the DBMS will perform using a particular design for the sample workload trace without needing to actually deploy the database. For this work, we use a cost model that seeks to optimize throughput by minimizing the number of distributed transactions [23, 30, 17] and the amount of access skew across servers [47]. Since the cost model is separate from the search model, one could replace it to generate designs that accentuate other aspects of the database (e.g., minimizing disk seeks, improving crash resiliency). We discuss alternative cost models for Horticulture for other DBMSs in Section 9.

We now present our LNS-based approach in the next section, and then describe in Section 5 how Horticulture estimates the number of distributed transactions and the amount of skew for each design. Various optimization techniques, such as how to extract, analyze, and compress information from a sample workload trace efficiently and to speed up the search time, are discussed in Section 6.

4. LARGE-NEIGHBORHOOD SEARCH

LNS is well-suited for our problem domain because it explores large solution spaces with a lower chance of getting caught in a local minimum and has been shown to converge to near-optimal solutions in a reasonable amount of time [21]. An outline of Horticulture's design algorithm is as follows:

1. Analyze the sample workload trace to pre-compute information used to guide the search process. (Section 6)
2. Generate an initial "best" design $\mathcal{D}_{best}$ based on the database's most frequently accessed columns. (Section 4.1).
3. Create a new incomplete design $\mathcal{D}_{relax}$ by "relaxing" (i.e., resetting) a subset of $\mathcal{D}_{best}$. (Section 4.2)
4. Perform a local search [49] for a new design using $\mathcal{D}_{relax}$ as a starting point. If any new design has a lower cost than $\mathcal{D}_{best}$, then mark it as the new $\mathcal{D}_{best}$. The search stops when a certain number of designs fail to improve on $\mathcal{D}_{best}$ or there are no designs remaining in $\mathcal{D}_{relax}$'s neighborhood. (Section 4.3)
5. If the total time spent thus far exceeds a limit, then halt the algorithm and return $\mathcal{D}_{best}$. Otherwise, repeat Step 3 for a new $\mathcal{D}_{relax}$ derived from $\mathcal{D}_{best}$.

When generating either the initial design in Step 1 or subsequent

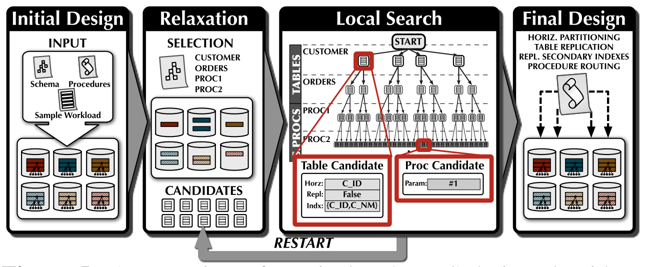


Figure 5: An overview of Horticulture's LNS design algorithm. The algorithm generates a relaxed design from the initial design and then uses local search to explore solutions. Each level of the search tree contains the different candidate attributes for tables and procedures for the target database. After the search finishes, the process either restarts or emits the best solution found.

designs using local search in Step 4, Horticulture verifies whether a design is *feasible* for the target cluster (i.e., the total size of the data stored on each node is less than its storage limit) [18]. Non-feasible designs are immediately discarded.

Next, we describe each of these steps in more detail.

4.1 Initial Design

The ideal initial design is one that is easy to compute and provides a good upper bound to the optimal solution. This allows LNS to discard many potential designs at the beginning of the search because they do not improve on this initial design. To this purpose our system builds compact summaries of the frequencies of access and co-access of tables, called *access graphs*. We postpone the detailed discussion of access graphs and how we derive them from a workload trace to Section 6.1.

Horticulture uses these access graphs in a four-part heuristic to generate an initial design:

1. Select the most frequently accessed column in the workload as the horizontal partitioning attribute for each table.
2. Greedily replicate read-only tables if they fit within the partitions' storage space limit.
3. Select the next most frequently accessed, read-only column in the workload as the secondary index attribute for each table if they fit within the partitions' storage space limit.
4. Select the routing parameter for stored procedures based on how often the parameters are referenced in queries that use the table partitioning columns selected in Step 1.

To identify which read-only tables in the database to replicate in Step 2, we first sort them in decreasing order by each table's *temperature* (i.e., the size of the table divided by the number of transactions that access the table) [16]. We examine each table one-by-one according to this sort order and calculate the new storage size of the partitions if that table was replicated. If this size is still less than the amount of storage available for each partition, then we mark the table as replicated. We repeat this process until either all read-only tables are replicated or there is no more space.

We next select the secondary index column for any non-replicated table as the one that is both read-only and accessed the most often in queries' predicates that do not also reference that table's horizontal partitioning column chosen in Step 1. If this column generates an index that is too large, we examine the next most frequently access column for the table.

Now with every table either replicated or partitioned in the initial design, Horticulture generates *parameter mappings* [37] from the workload trace that identify (1) the procedure input parameters that are also used as query input parameters and (2) the input parameters for one query that are also used as the input parameters for other queries. These mappings allow Horticulture to identify without using static code analysis which queries are always executed with the same input parameters using the actual values of the input parameters in the workload. The technique described in [37] removes spurious results for queries that reference the same columns but with different values. We then select a routing attribute for each stored procedure as the one that is mapped to the queries that are executed the most often with predicates on the tables' partitioning columns. If no sufficient mapping exists for a procedure, then its routing attribute is chosen at random.

4.2 Relaxation

Relaxation is the process of selecting random tables in the database and resetting their chosen partitioning attributes in the current best design. The partitioning option for a relaxed table is undefined in the design, and thus the design is *incomplete*. We discuss how to calculate cost estimates for incomplete designs in Section 5.3.

In essence, relaxation allows LNS to escape a local minimum and to jump to a new neighborhood of potential solutions. This is advantageous over other approaches, such as tableau search, because it is relatively easy to compute and does not require the algorithm to maintain state between relaxation rounds [21]. To generate a new relaxed design, Horticulture must decide (1) how many tables to relax, (2) which tables to relax, and (3) what design options will be examined for each relaxed table in the local search.

As put forth in the original LNS papers [18, 21], the number of relaxed variables (i.e., tables) is based on how much search time remains as defined by the administrator. Initially, this size is 25% of the total number of tables in the database; as time elapses, the limit increases up to 50%[1]. Increasing the number of tables relaxed over time in this manner is predicated on the idea that a tighter upper bound will be found more quickly if the initial search rounds use a smaller number of tables, thereby allowing larger portions of the solution space to be discarded in later rounds [18, 21].

After computing the number of tables to reset, Horticulture then randomly chooses which ones it will relax. If a table is chosen for relaxation, then all of the routing parameters for any stored procedure that references that table are also relaxed. The probability that a table will be relaxed in a given round is based on their temperatures [16]: a table that is accessed frequently more likely to be selected to help the search find a good upper bound more quickly [21]. We also reduce these weights for small, read-only tables that are already replicated in the best design. These are usually the "look-up" tables in OLTP applications [43], and thus we want to avoid exploring neighborhoods where they are not replicated.

In the last step, Horticulture generates the *candidate attributes* for the relaxed tables and procedures. For each table, its candidate attributes are the unique combination of the different design options available for that table (Section 3.1). For example, one potential candidate for `CUSTOMER` table is to horizontally partition the table on the customer's name, while another candidate partitions the table on the customer's id and includes a replicated secondary index on the customer id and name. Multiple candidate attributes for a single table are grouped together as an indivisible "virtual" attribute. The different options in one of these virtual attributes are applied to a design all at once so that the estimated cost never decreases during the local search process.

4.3 Local Search

Using the relaxed design $\mathcal{D}_{relax}$ produced in the previous step, Horticulture executes a two-phase search algorithm to iteratively

[1]These values were empirically evaluated following standard practice guidelines [18].

explore solutions. This process is represented as a search tree, where each *level* of the tree coincides with one of the relaxed database elements. As shown in Fig. 5, the search tree's levels are split into two sections corresponding to the two search phases. In the first phase, Horticulture explores the tables' candidate attributes using a branch-and-bound search [49, 32]. Once all of the relaxed tables are assigned an attribute in $\mathcal{D}_{relax}$, Horticulture then performs a brute-force search in the second phase to select the stored procedures' routing parameters.

As Horticulture explores the table portion of the search tree, it changes the current table's design option in $\mathcal{D}_{relax}$ to each candidate attribute and then estimates the cost of executing the sample workload using that new design. If this cost estimate is less than the cost of $\mathcal{D}_{best}$ and is feasible, then the search traverses down the tree and examines the next table's candidate attributes. But if this cost is greater than or equal to the cost of $\mathcal{D}_{best}$ or if the design is not feasible, the search continues on to the next candidate attribute for the current table. If there are no more attributes for this level, then the search "backtracks" to the previous level.

Horticulture maintains counters for backtracks and the amount of time spent in the current search round. Once either of these exceed a dynamic limit, the local search halts and returns to the relaxation step. The number of backtracks and search time allowed for each round is based on the number of tables that were relaxed in $\mathcal{D}_{relax}$. As these limits increases over time, the search is given more time to explore larger neighborhoods. We explore the sensitivity of these parameters in our evaluation in Section 7.6.

In the second phase, Horticulture uses a different search technique for procedures because their design options are independent from each other (i.e., the routing parameter for one procedure does not affect whether other procedures are routed correctly). Therefore, for each procedure, we calculate the estimated costs of its candidate attributes one at a time and then choose the one with the lowest cost before moving down to the next level in the search tree. We examine the procedures in descending order of invocation frequency so that the effects of a bad design are discovered earlier.

If Horticulture reaches the last level in the tree and has a design that is both feasible and has a cost that is less than $\mathcal{D}_{best}$, then the current design becomes the new best design. The local search still continues but now all comparisons are conducted with the new lower cost. Once either of the search limits is reached or when all of the tree is explored, the process restarts using a new relaxation.

The entire process halts after after an administrator-defined time limit or when Horticulture fails to find a better design after a certain period of time (Section 7.6). The final output is the best design found overall for the application's database. The administrator then configures the DBMS using the appropriate interface to deploy their database according to this design.

5. SKEW-AWARE COST MODEL

Horticulture's LNS algorithm relies on a cost model that can estimate the cost of executing the sample workload using a particular design [16, 35, 49, 29]. Using an analytical cost model is an established technique in automatic database design and optimization [13, 19], as it allows one to determine whether one design choice is better than others and can guide the search process towards a solution that accentuates the properties that are important in a database. But it is imperative that these estimations are computed quickly, since the LNS algorithm can generate thousands of designs during the search process. The cost model must also be able to estimate the cost of an incomplete design. Furthermore, as the search process continues down the tree, the cost estimates must increase monotonically as more variables are set in an incomplete design.

Algorithm 1 $CoordinationCost(\mathcal{D}, \mathcal{W})$

```
$txnCount \leftarrow 0, dtxnCount \leftarrow 0, partitionCount \leftarrow 0$
for all $txn \in \mathcal{W}$ do
    $P \leftarrow GetPartitions(\mathcal{D}, txn)$
    if $|P| > 1$ then
        $dtxnCount \leftarrow dtxnCount + 1$
        $partitionCount \leftarrow partitionCount + |P|$
    end if
    $txnCount \leftarrow txnCount + 1$
end for
return $\left(\frac{partitionCount}{(txnCount \times numPartitions)} \times \left(1.0 + \frac{dtxnCount}{txnCount}\right)\right)$
```

Given these requirements, our cost model is predicated on the key observation that the execution overhead of a multi-partition transaction is significantly more than a single-partition transaction [43, 24]. Some OLTP DBMSs execute a single-partition transaction serially on a single node with reduced concurrency control, whereas any distributed transactions must use an expensive concurrency control scheme to coordinate execution across two or more partitions [43, 25, 37]. Thus, we estimate the run time cost of a workload as being proportional to the number of distributed transactions.

In addition to this, we also assume that (1) either the working set for an OLTP application or its entire database is stored in main memory and (2) that the run times for transactions are approximately the same. This means that unlike other existing cost models [16, 49, 13], we can ignore the amount of data accessed by each transaction, and that all of a transaction's operations contribute an equal amount to the overall load of each partition. In our experience, transactions that deviate from these assumptions are likely analytical operations that are either infrequent or better suited for a data warehouse DBMS.

We developed an analytical cost model that not only measures how much of a workload executes as single-partition transactions, but also measures how uniformly load is distributed across the cluster. The final cost estimation of a workload $\mathcal{W}$ for a design $\mathcal{D}$ is shown below as the function *cost*$(\mathcal{D}, \mathcal{W})$, which is the weighted sum of the normalized coordination cost and the skew factor:

$$cost(\mathcal{D}, \mathcal{W}) = \frac{(\alpha \times CoordinationCost(\mathcal{D}, \mathcal{W})) + (\beta \times SkewFactor(\mathcal{D}, \mathcal{W}))}{(\alpha + \beta)}$$

The parameters α and β can be configured by the administrator. In our setting, we found via linear regression that the values five and one respectively provided the best results. All experiments were run with this parameterization.

This cost model is not intended to estimate actual run times, but rather as a way to compare the quality of competing designs. It is based on the same assumptions used in H-Store's distributed query planner. We show that the underlying principals of our cost model are representative of actual run time performance in Section 7.3.

5.1 Coordination Cost

We define the function $CoordinationCost(\mathcal{D}, \mathcal{W})$ as the portion of the cost model that calculates how well $\mathcal{D}$ minimizes the number of multi-partition transactions in $\mathcal{W}$; the cost increases from zero as both the number of distributed transactions and the total number of partitions accessed by those transactions increases.

As shown in Algorithm 1, the $CoordinationCost$ function uses the DBMS's internal API function $GetPartitions$ to estimate what partitions each transaction will access [12, 37]. This is the same API that the DBMS uses at run time to determine where to route query requests. For a given design $\mathcal{D}$ and a transaction txn, this function deterministically returns the set of partitions P, where for each $p \in P$ the transaction txn either (1) executed at least one query that accessed p or (2) executed its stored procedure control code at the node managing p (i.e., its base partition). The partitions

Algorithm 2 $SkewFactor(\mathcal{D}, \mathcal{W})$

```
$skew \leftarrow [\,], txnCounts \leftarrow [\,]$
**for** $i \leftarrow 0$ to $numIntervals$ **do**
    $skew[i] \leftarrow CalculateSkew(\mathcal{D}, \mathcal{W}, i)$
    $txnCounts[i] \leftarrow NumTransactions(\mathcal{W}, i)$
**end for**
**return** $\left( \dfrac{\sum_{i=0}^{numIntervals} skew[i] \times txnCounts[i]}{\sum txnCounts} \right)$
```

accessed by txn's queries are calculated by examining the input parameters that reference the tables' partitioning columns in $\mathcal{D}$ (if it is not replicated) in the pre-computed query plans.

There are three cases that $GetPartitions$ must handle for designs that include replicated tables and secondary indexes. First, if a read-only query accesses only replicated tables or indexes, then the query executes on the same partition as its transaction's base partition. Next, if a query joins replicated and non-replicated tables, then the replicated tables are ignored and the estimated partitions are the ones needed by the query to access the non-replicated tables. Lastly, if a query modifies a replicated table or secondary index, then that query is broadcast to all of the partitions.

After counting the distributed transactions, the coordination cost is calculated as the ratio of the total number of partitions accessed ($partitionCount$) divided by the total number of partitions that could have been accessed. We then scale this result based on the ratio of distributed to single-partition transactions. This ensures, as an example, that the cost of a design with two transactions that both access three partitions is greater than a design where one transaction is single-partitioned and the other accesses five partitions.

5.2 Skew Factor

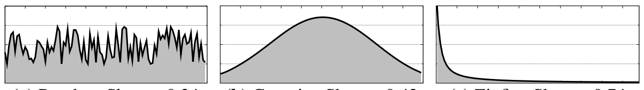
Although by itself $CoordinationCost$ is able to generate designs that maximize the number of single-partition transactions, it causes the design algorithm to prefer solutions that store the entire database in as few partitions as possible. Thus, we must include an additional factor in the cost model that strives to spread the execution workload uniformly across the cluster.

The function $SkewFactor(\mathcal{D}, \mathcal{W})$ shown in Algorithm 2 calculates how well the design minimizes skew in the database. To ensure that skew measurements are not masked by time, the $SkewFactor$ function divides $\mathcal{W}$ into finite intervals ($numIntervals$) and calculates the final estimate as the arithmetic mean of the skew factors weighted by the number of transactions executed in each interval (to accommodate variable interval sizes). To illustrate why these intervals are needed, consider a design for a two-partition database that causes all of the transactions at time t_1 to execute only on the first partition while the second partition remains idle, and then all of the transactions at time t_2 execute only on the second partition. If the skew is measured as a whole, then the load appears balanced because each partition executed exactly half of the transactions. The value of $numIntervals$ is an administrator-defined parameter. In our evaluation in Section 7, we use an interval size that aligns with workload shifts to illustrate that our cost model detects this skew. We leave it as future work to derive this parameter using a pre-processing step that calculates non-uniform windows.

The function $CalculateSkew(\mathcal{D}, \mathcal{W}, interval)$ shown in Algorithm 3 generates the estimated skew factor of $\mathcal{W}$ on $\mathcal{D}$ for the given interval. We first calculate how often partitions are accessed and then determine how much over- or under-utilized each partition is in comparison with the optimal distribution ($best$). To ensure that idle partitions are penalized as much as overloaded partitions, we invert any partition estimates that are less than $best$, and then scale them such that the skew value of a ratio as it approaches zero is the same as a ratio as it approaches one. The final normalized result is the sum of all the skew values for each partition divided by the total skew value for the cluster when all but one partition is idle.

Algorithm 3 $CalculateSkew(\mathcal{D}, \mathcal{W}, interval)$

```
$partitionCounts \leftarrow [\,]$
**for all** $txn \in \mathcal{W}$, where $txn.interval = interval$ **do**
    **for all** $p \in GetPartitions(\mathcal{D}, txn)$ **do**
        $partitionCounts\,[p] \leftarrow partitionCounts\,[p] + 1$
    **end for**
**end for**
$total \leftarrow \sum partitionCounts$
$best \leftarrow \frac{1}{numPartitions}$
$skew \leftarrow 0$
**for** $i \leftarrow 0$ to $numPartitions$ **do**
    $ratio \leftarrow \frac{partitionCounts[i]}{total}$
    **if** $ratio < best$ **then**
        $ratio \leftarrow best + \left(\left(1 - \frac{ratio}{best}\right) \times (1 - best)\right)$
    **end if**
    $skew \leftarrow skew + \log\left(\frac{ratio}{best}\right)$
**end for**
**return** $\left( \dfrac{skew}{\log\left(\frac{1}{best}\right) \times numPartitions} \right)$
```



(a) Random Skew = 0.34 **(b)** Gaussian Skew = 0.42 **(c)** Zipfian Skew = 0.74

Figure 6: Example $CalculateSkew$ estimates for different distributions on the number of times partitions are accessed.

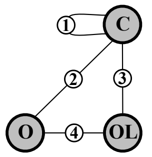
Fig. 6 shows how the skew factor estimates increase as the amount of skew in the partitions' access distribution increases.

5.3 Incomplete Designs

Our cost model must also calculate estimates for designs where not all of the tables and procedures have been assigned an attribute yet [32]. This allows Horticulture to determine whether an incomplete design has a greater cost than the current best design, and thus allows it to skip exploring the remainder of the search tree below its current location. We designate any query that references a table with an unset attribute in a design as being *unknown* (i.e., the set of partitions accessed by that query cannot be estimated). To compute the coordination cost of an incomplete design, we assume that any unknown query is single-partitioned. We take the opposite tack when calculating the skew factor of an incomplete design and assume that all unknown queries execute on all partitions in the cluster. As additional information is added to the design, queries change to a *knowable* state if all of the tables referenced by the query are assigned a partitioning attribute. Any unknown queries that are single-partitioned for an incomplete design $\mathcal{D}$ may become distributed as more variables are bound in a later design $\mathcal{D}'$. But any transaction that is distributed in $\mathcal{D}$ can never become single-partitioned in $\mathcal{D}'$, as this would violate the monotonically increasing cost function requirement of LNS.

6. OPTIMIZATIONS

We now provide an overview of the optimizations that we developed to improve the search time of Horticulture's LNS algorithm. The key to reducing the complexity of finding the optimal database design for an application is to minimize the number of designs that are evaluated [49]. To do this, Horticulture needs to determine which attributes are relevant to the application and are thus good candidates for partitioning. For example, one would not horizontally partition a table by a column that is not used in any query. Horticulture must also discern which relevant attributes are



Edge#	Columns	Weight
(1)	`C.C_ID` ↔ `C.C_ID`	200
(2)	`C.C_ID` ↔ `O.O_C_ID`	100
(3)	`O.O_ID` ↔ `OL.OL_O_ID`	100
(4)	`O.O_ID` ↔ `OL.OL_O_ID` `O.O_C_ID` ↔ `OL.OL_C_ID`	100

Figure 7: An access graph derived from a workload trace.

accessed the most often and would therefore have the largest impact on the DBMS's performance. This allows Horticulture to explore solutions using the more frequently accessed attributes first and potentially move closer to the optimal solution more quickly.

We now describe how to derive such information about an application from its sample workload and store them in a graph structure used in Sections 4.1 and 4.3. We then present a novel compression scheme for reducing the number of transactions that are examined when computing cost model estimates in Section 5.

6.1 Access Graphs

Horticulture extracts the key properties of transactions from a workload trace and stores them in undirected, weighted graphs, called *access graphs* [3, 49]. These graphs allow the tool to quickly identify important relationships between tables without repeatedly reprocessing the trace. Each table in the schema is represented by a vertex in the access graph and vertices are adjacent through edges in the graph if the tables they represent are *co-accessed*. Tables are considered co-accessed if they are used together in one or more queries in a transaction, such as in a join. For each pair of co-accessed attributes, the graph contains an edge that is weighted based on the number of times that the queries forming this relationship are executed in the workload trace. A simplified example of an access graph for the TPC-C benchmark is shown in Fig. 7.

We extend prior definitions of access graphs to accommodate stored procedure-based DBMSs. In previous work, an access graph's structure is based on either queries' join relationships [49] or tables' join order in query plans [3]. These approaches are appropriate when examining a workload on a query-by-query basis, but fail to capture relationships between multiple queries in the same transaction, such as a logical join operation split into two or more queries—we call this an *implicit reference*.

To discover these implicit references, Horticulture uses a workload's parameter mappings [37] to determine whether a transaction uses the same input parameters in multiple query invocations. Since implicit reference edges are derived from multiple queries, their weights are based on the minimum number of times those queries are all executed in a single transaction [49].

6.2 Workload Compression

Using large sample workloads when evaluating a potential design improves the cost model's ability to estimate the target database's properties. But the cost model's computation time depends on the sample workload's size (i.e., the number of transactions) and complexity (i.e., the number of queries per transaction). Existing design tools employ random sampling to reduce workload size [17], but this approach can produce poor designs if the sampling masks skew or other potentially valuable information about the workload [11]. We instead use an alternative approach that compresses redundant transactions and redundant queries without sacrificing accuracy. Our scheme is more efficient than previous methods in that we only consider what tables and partitions that queries access, rather than the more expensive task of comparing sets of columns [11, 19].

Compressing a transactional workload is a two-step process. First, we combine sets of similar queries in individual transactions into fewer weighted records [19]. Such queries often occur in stored procedures that contain loops in their control code. After combining queries, we then combine similar transactions into a smaller number of weighted records in the same manner. The cost model will scale its estimates using these weights without having to process each of the records separately in the original workload.

To identify which queries in a single transaction are combinable, we compute the *input signature* for each query from the values of its input parameters and compare it with the signature of all other queries. A query's input signature is an unordered list of pairs of tables and partition ids that the query would access if each table is horizontally partitioned on a particular column. As an example, consider the following query on the `CUSTOMER` (`C`) table:

```
SELECT * FROM C WHERE C_ID = 10 AND C_LAST = "Smith"
```

Assuming that the input value "10" corresponds to partition #10 if the table was partitioned on `C_ID` and the input value "Smith" corresponds to partition #3 if it was partitioned on `C_LAST`, then this query's signature is $\{(\texttt{C}, 10), (\texttt{C}, 3)\}$. We only use the parameters that are used with co-accessed columns when computing the signature. For example, if only `C_ID` is referenced in the access graph, then the above example's input signature is $\{(\texttt{C}, 10)\}$.

Each transaction's input signature includes the query signatures computed in the previous step, as well as the signature for the transaction's procedure input parameters. Any set of transactions with the same query signatures and procedure input parameter signature are combined into a single weighted record.

7. EXPERIMENTAL EVALUATION

To evaluate the effectiveness Horticulture's design algorithms, we integrated our tool with H-Store and ran several experiments that compare our approach to alternative approaches. These other algorithms include a state-of-the-art academic approach, as well as other solutions commonly applied in practice:

HR+ Our large-neighborhood search algorithm from Section 4.
HR– Horticulture's baseline iterative greedy algorithm, where design options are chosen one-by-one independently of others.
SCH The Schism [17] graph partitioning algorithm.
PKY A simple heuristic that horizontally partitions each table based on their primary key.
MFA The initial design algorithm from Section 4.1 where options are chosen based on how frequently attributes are accessed.

7.1 Benchmark Workloads

We now describe the workloads from H-Store's built-in benchmark framework that we used in our evaluation. The size of each database is approximately 1GB per partition.

TATP: This is an OLTP testing application that simulates a typical caller location system used by telecommunication providers [48]. It consists of four tables, three of which are foreign key descendants of the root `SUBSCRIBER` table. Most of the stored procedures in TATP have a `SUBSCRIBER` id as one of their input parameters, allowing them to be routed directly to the correct node.

TPC-C: This is the current industry standard for evaluating the performance of OLTP systems [45]. It consists of nine tables and five stored procedures that simulate a warehouse-centric order processing application. All of the procedures in TPC-C provide a warehouse id as an input parameter for the transaction, which is the foreign key ancestor for all tables except `ITEM`.

TPC-C (Skewed): Our benchmarking infrastructure also allows us to tune the access skew for benchmarks. In particular, we generated a temporally skew load for TPC-C, where the `WAREHOUSE` id used in the transactions' input parameters is chosen so that at each time interval all of the transactions target a single warehouse. This workload is uniform when observed globally, but at any point in time there is a significant amount of skew. This help us to stress-test our system when dealing with temporal-skew, and to show the potential impact of skew on the overall system throughput.

SEATS: This benchmark models an on-line airline ticketing system where customers search for flights and make reservations [44]. It consists of eight tables and six stored procedures. The benchmark is designed to emulate a back-end system that processes requests from multiple applications that each provides disparate inputs. Thus, many of its transactions must use secondary indexes or joins to find the primary key of a customer's reservation information. For example, customers may access the system using either their frequent flyer number or customer account number. The non-uniform distribution of flights between airports also creates imbalance if the database is partitioned by airport-derived columns.

AuctionMark: This is a 16-table benchmark based on an Internet auction system [6]. Most of its 10 procedures involve an interaction between a buyer and a seller. The user-to-item ratio follows a Zipfian distribution, which means that there are a small number of users that are selling a large portion of the total items. The total number of transactions that target each item is temporally skewed, as items receive more activity (i.e., bids) as the auction approaches its closing time. It is difficult to generate a design for AuctionMark that includes stored procedure routing because several of the benchmark's procedures include conditional branches that execute different queries based on the transaction's input parameters.

TPC-E: Lastly, the TPC-E benchmark is the successor of TPC-C and is designed to reflect the workloads of modern OLTP applications [46]. Its workload features 12 stored procedures, 10 of which are executed in the regular transactional mix while two are periodically executed "clean-up" procedures. Unlike the other benchmarks, many of TPC-E's 33 tables have foreign key dependencies with multiple tables, which create conflicting partitioning candidates. Some of the procedures also have optional input parameters that cause transactions to execute mutually exclusive sets of queries based on which of these parameters are given at run time.

7.2 Design Algorithm Comparison

The first experiment that we present is an off-line comparison of the database design algorithms listed above. We execute each algorithm for all of the benchmarks to generate designs for clusters ranging from four to 64 partitions. Each algorithm is given an input workload trace of 25k transactions, and then is tested using a separate trace of 25k transactions. We evaluate the effectiveness of the designs of each algorithm by measuring the number of distributed transactions and amount of skew in those designs over the test set.

Fig. 8a shows that HR+ produces designs with the lowest coordination cost for every benchmark except TPC-C (Skewed), with HR– and SCH designs only slightly higher. Because fewer partitions are accessed using HR+'s designs, the skew estimates in Fig. 8b greater (this why the cost model uses the α and β parameters). We ascribe the improvements of HR+ over HR– and MFA to the LNS algorithm's effective exploration of the search space using our cost model and escaping local minima.

For TPC-C (Skewed), HR+ chooses a design that increases the number of distributed transactions in exchange for a more balanced load. Although the SCH algorithm does accommodate skew when selecting a design, it currently does not support the temporal skew used in this benchmark. The skew estimates for PKY and MFA are lower than others in Fig. 8b because more of the transactions touch all of the partitions, which causes the load to be more uniform.

7.3 Transaction Throughput

The next experiment is an end-to-end test of the quality of the designs generated in the previous experiment. We compare the designs from our best algorithm (HR+) against the state-of-the-art academic approach (SCH) and the best baseline practical solution (MFA). We execute select benchmarks in H-Store using the designs for these algorithms and measure the system's overall throughput.

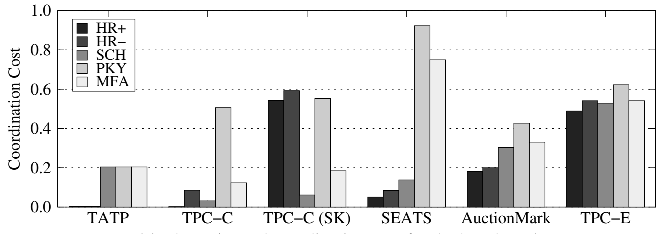
We execute each benchmark using five different cluster sizes of Amazon EC2 nodes allocated within a single region. Each node has eight virtual cores and 70GB of RAM (m2.4xlarge). We assign at most seven partitions per node, with the remaining partition reserved for the networking and administrative functionalities of H-Store. The execution engine threads are given exclusive access to a single core to improve cache locality.

Transaction requests are submitted from up to 5000 simulated client terminals running on separate nodes in the same cluster. Each client submits transactions to any node in the H-Store cluster in a closed loop: after it submits a request, it blocks until the result is returned. Using a large number of clients ensures that the execution engines' workload queues are never empty.

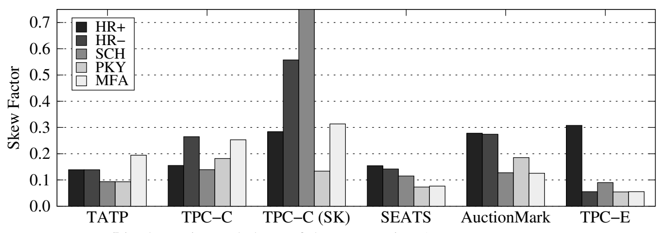
We execute each benchmark three times per cluster size and report the average throughput of these trials. In each trial, the DBMS "warms-up" for 60 seconds and then the throughput is measured after five minutes. The final throughput is the number of transactions completed in a trial run divided by the total time (excluding the warm-up period). H-Store's benchmark framework ensures that each run has the proper distribution of executed procedures according to the benchmark's specification.

All new requests are executed in H-Store as single-partitioned transactions with reduced concurrency control protection; if a transaction attempts to execute a multi-partition query, then it is aborted and restarted with full concurrency control. Since SCH does not support stored procedure routing, the system is unable to determine where to execute each transaction request even if the algorithm generates the optimal partitioning scheme for tables. Thus, to obtain a fair comparison of the two approaches, we implemented a technique from IBM DB2 [15] in H-Store to handle this scenario. Each transaction request is routed to a random node by the client where it will start executing. If the first query that the transaction dispatches attempts to access data not stored at that node, then it is aborted and re-started at the proper node. This ensures that single-partition transactions execute with reduced concurrency control protection, which is necessary for achieving good throughput in H-Store.

The throughput measurements in Fig. 9 show that the designs generated by HR+ improve the throughput of H-Store by factors $1.3\times$ to $4.3\times$ over SCH and $1.1\times$ to $16.3\times$ over MFA. This validates two important hypotheses: (1) that our cost model and search technique are capable of finding good designs, and (2) that by explicitly accounting for stored procedure routing, secondary indexes replication, and temporal-skew management, we can significantly improve over previous best-in-class solutions. Other notable observations are that (1) the results for AuctionMark highlight the importance of stored procedure routing, since this is the only difference between SCH and HR+, (2) the TATP, SEATS, and TPC-C experiments demonstrate the combined advantage of stored procedures and replicated secondary indexes, and (3) that TPC-C (Skewed) illustrates the importance of mitigating temporal-skew. We also note that the performance of H-Store is less than expected for larger cluster sizes due to clock skew issues when choosing transaction identifiers that ensure global ordering [43].

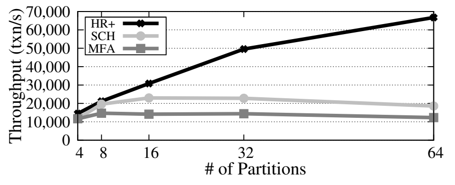


(a) The estimated coordination cost for the benchmarks.

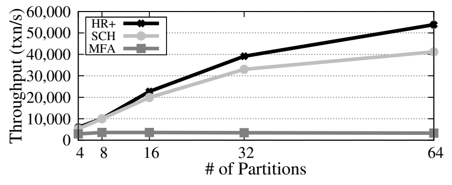


(b) The estimated skew of the transactions' access patterns.

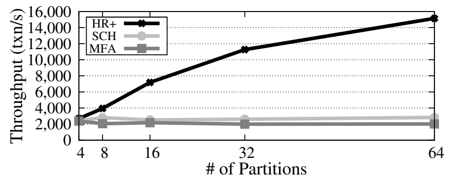
Figure 8: Offline measurements of the designs algorithms in Section 7.2.



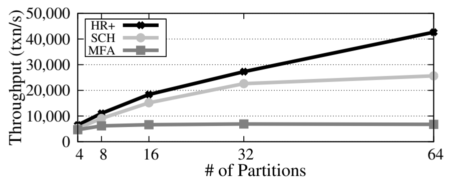
(a) TATP



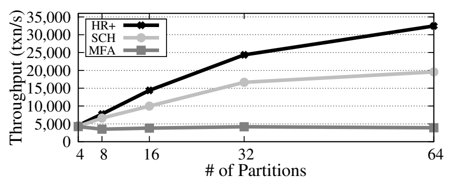
(b) TPC-C



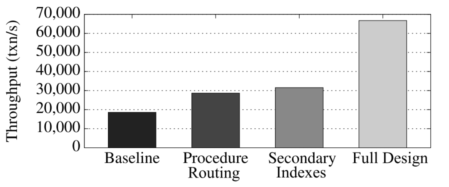
(c) TPC-C (Skewed)



(d) SEATS



(e) AuctionMark



(f) Design Components

Figure 9: Transaction throughput measurements for the HR+, SCH, and MFA design algorithms.

For this last item, we note that TPC-C (Skewed) is designed to stress-test the designs algorithms under extreme temporal-skew conditions to evaluate its impact on throughput; we do not claim this to be a common scenario. In this setting, any system ignoring temporal-skew will choose the same design used in Fig. 9b, resulting in near-zero scale-out. Fig. 9b shows that both SCH and MFA do not improve performance as more nodes are added to the cluster. On the contrary, HR+ chooses a different design (i.e., partitioning by `WAREHOUSE id` and `DISTRICT id`), thus accepting many more distributed transactions in order to reduce skew. Although all the approaches are affected by skew resulting in an overall lower throughput, HR+ is significantly better with more than $6\times$ throughput increase for the same $8\times$ increase in nodes.

To further ascertain the impact of the individual design elements, we executed TATP again using the HR+ design but alternatively removing: (1) client-side stored procedure routing (falling back on the redirection mechanism we built to test SCH), (2) the secondary indexes replication, or (3) both. Fig. 9f shows the relative contributions with stored procedure routing delivering 54.1% over the baseline approach (that otherwise coincide with the one found by SCH), secondary indexes contribute 69.6%, and combined they deliver a $3.5\times$ improvement. This is because there is less contention for locking partitions in the DBMS's transaction coordinators [25].

7.4 Cost Model Validation

Horticulture's cost model is not meant to provide exact throughput predictions, but rather to quickly estimate the relative ordering of multiple designs. To validate that these estimates are correct, we tested its accuracy for each benchmark and number of partitions by comparing the results from Fig. 8 and Fig. 9. We note that our cost model predicts which design is going to perform best in 95% of the experiments. In the cases where the cost model fails to predict the optimal design, our analysis indicates that they are inconsequential because they are from workloads where the throughput results are almost identical (e.g., TATP on four partitions). We suspect that the throughput differences might be due to transitory EC2 load conditions rather than actual difference in the designs. Furthermore, the small absolute difference indicates that such errors will not significantly degrade performance.

7.5 Compression & Scalability

We next measured the workload compression rate for the scheme described Section 6.2 using the benchmark's sample workloads when the number of partitions increases exponentially. The results in Fig. 10 show that the compression rate decreases for all of the benchmarks as the number of partitions increases due to the decreased likelihood of duplicate parameter signatures. The workload for the TPC-C benchmark does not compress well due to greater variability in the procedure input parameter values.

We also analyzed Horticulture's ability to generate designs for large cluster sizes. The results in Fig. 11 shows that the search time for our tool remains linear as the size of the database increases.

7.6 Search Parameter Sensitivity Analysis

As discussed in Section 4.3, there are parameters that control the run time behavior of Horticulture: each local search round executes until either it (1) exhausts its time limit or (2) reaches its backtrack limit. Although Horticulture dynamically adjusts these parameters [21], their initial values can affect the quality of the designs found. For example, if the time limit is too small, then Horticulture will fail to fully explore each neighborhood. Moreover, if it is too large, then too much time will be spent exploring neighborhoods that never yield a better design. The LNS algorithm will continue looking for a better design until either it (1) surpasses the total amount of time allocated by the administrator or (2) has exhausted the search space. In this experiment, we investigate what are good default values for these search parameters.

We first experimented with using different local search and backtrack limits for the TPC-E benchmark. We chose TPC-E because it has the most complex schema and workload. We executed the LNS algorithm for two hours using different local search time limits with an infinite backtrack limit. We then repeated this experiment using an infinite local search time limit but varying the backtrack limit. The results in Fig. 12 show that using the initial limits of approxi-

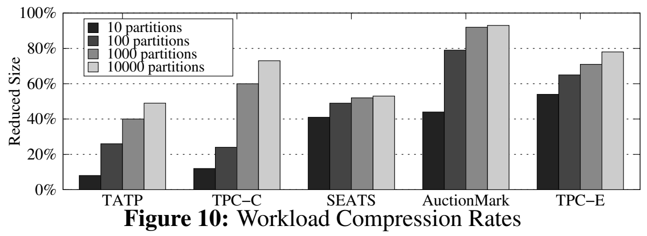


Figure 10: Workload Compression Rates

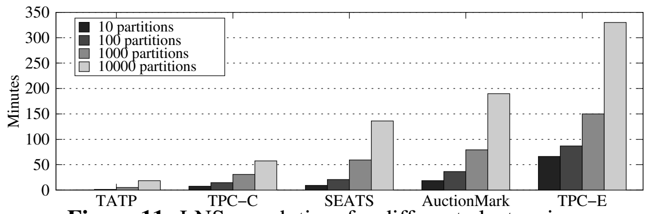


Figure 11: LNS search time for different cluster sizes

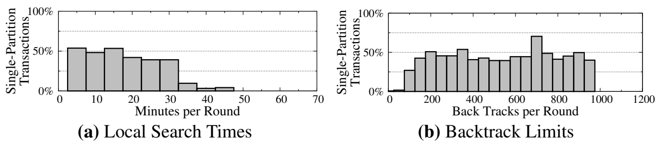


Figure 12: A comparison of LNS-generated designs for TPC-E using different (a) local search times and (b) backtrack limits.

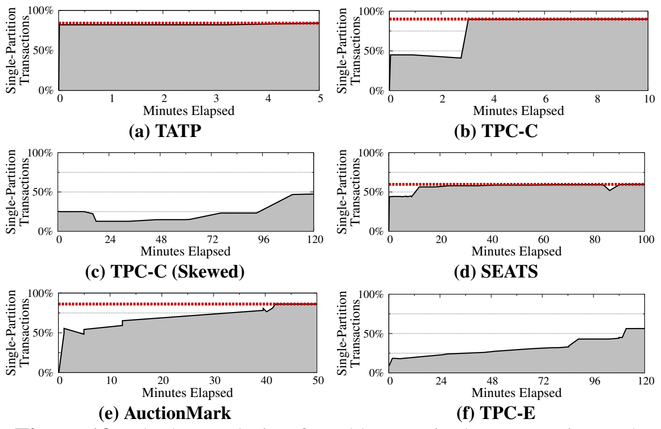


Figure 13: The best solution found by Horticulture over time. The red dotted lines represent known optimal designs (when available).

mately five minutes and 100–120 backtracks produces designs with lower costs more quickly.

Non-deterministic algorithms, such as LNS, are not guaranteed to discover the optimal solution, which means that there is no way for the administrator to know how much time to let Horticulture to continue searching. Thus, we need a way to know when to stop searching for a better design. A naïve approach is to halt when the algorithm fails to find a new solution after a certain amount of time. But this is difficult to estimate for arbitrary inputs, since the search time is dependent on a number of factors.

Another approach is to calculate a lower bound using a theoretical design [32] and then halt the LNS algorithm when it finds a design with a cost that is within a certain distance to that bound [21]. We compute this bound by estimating the cost of the workload using a design where all transactions execute as single-partitioned and with no skew in the cluster (i.e., round-robin assignment). Note that such a design is likely infeasible, since partitioning a database to make every transaction single-partitioned cannot always be done without making other transactions distributed. The graphs in Fig. 13 show the amount of time it takes for the LNS algorithm to find solutions that converge towards the lower bound. We show the quality of the design in terms of single-partition transactions as time proceeds. The red dotted line in each graph represents the best known design we are aware of for each benchmark—we do not have reference designs for TPC-C (Skewed) and TPC-E, other than the one found by our tools. The cost improvements shown in the graphs plateau after a certain point, which is the desired outcome.

Overall, these experiments show that to achieve great performance for OLTP workloads, especially on modern NewSQL systems, it is paramount that a design tool supports stored procedures, replicated secondary indexes, and temporal skew. To the best of our knowledge, Horticulture is the first to consider all of these issues.

8. RELATED WORK

There is an extensive corpus on the problem of automatic database partitioning, including both theoretical [10] and applied research [20]. Most notable are the advancements from two commercial database vendors: Microsoft's SQL Server AutoAdmin [12, 11, 3, 5, 32] and IBM's DB2 Database Advisor [39, 50]. We limit this discussion to the prior work that is relevant for parallel DBMSs.

The major differences amongst previous approaches are in (1) selecting the candidate partitioning attributes and (2) the search process used to find the optimal partitioning scheme. The former examines a sample workload and represents the usefulness of the candidate attributes extracted from the queries in auxiliary structures [10, 49, 49, 3, 39]. Microsoft's AutoAdmin finds sets of candidate attributes for individual queries and then attempts to merge them based on the entire workload [5]. The AutoPart tool identifies conflicting access patterns on tables and creates read-only vertical partitions from disjoint column subsets that are similar to our secondary indexes [36]. Further heuristics can then be applied to prune this candidate set or combine attributes into multi-attribute sets [5]. One can then use these attributes with a partitioning algorithm that uses heuristics [49], exhaustive search [39], or approximation [17]. The study done in [49] compares the performance versus quality trade-offs of different search strategies.

Schism is another automatic partitioning tool that also seeks to minimize distributed transactions [17]. For a given database, Schism populates a graph containing a separate vertex for every tuple and creates an edge between two vertices if the tuples that they represent are co-accessed together in a transaction in a sample workload. It then applies a graph partitioning algorithm to produce balanced boundaries that minimize the number of cross partition edges.

The work in [32] employs a branch-and-bound algorithm similar to our approach to search for table partitioning and replication choices for shared-nothing, disk-based DBMSs, but without the benefit of multiple search rounds in LNS. The lack of support for stored procedures, replicated secondary indexes, and temporal skew handling limit the effectiveness of [17, 32] for the enterprise OLTP applications that we consider.

For stored procedure routing on shared-nothing DBMSs, the authors in [38, 34] provide a thorough discussion of the static, decentralized scheme supported by Horticulture. The affinity-based routing approach in [38] directs requests to the nodes with data that the transactions will need using a broad categorization. The approaches in [33, 37] automatically generate a more fine-grained classification based on the previously executed transactions.

Much of the literature on cost estimation for main memory DBMSs is for single-node systems [14] or do not consider workload skew [29]. The method in [8] generates non-uniform partition sizes to accommodate the start-up delay in multi-node full-table sequential scan queries in main memory systems.

9. FUTURE WORK

We are extending Horticulture to generate database designs for different types of systems. For example, we are working adapting our tool for document-oriented NoSQL DBMSs to select the optimal sharding and index keys, as well to denormalize schemas. This work shows that Horticulture's LNS-based approach is adaptable to many different systems just by changing the cost model. We modified our cost model for NoSQL systems to estimate the number of disk operations per operation [49] and the overall skew using the same technique presented in Section 5.2. Because these systems do not support joins or distributed transactions, we do not need to use our coordination cost estimation.

Supporting database partitioning for a mixed OLTP and analytical workloads in Horticulture is another interesting research area. A new cost model would have to accommodate multiple objectives, such as improving intra-query parallelism in analytical queries while also satisfying service-level agreements for the front-end workload.

We are also developing data placement algorithms that assign the location and sizes of partitions to particular nodes in a cluster [35]. Generating an optimal placement strategy can improve the performance of a distributed transaction by increasing the likelihood that any "non-local" partition is located on the same node.

10. CONCLUSION

We presented a new approach for automatically partitioning a database in a shared-nothing, parallel DBMS. Our algorithm uses a large-neighborhood search technique together with an analytical cost model to minimize the number of distributed transactions while controlling the amount of skew. To the best of our knowledge, the system that we present is the first to target enterprise OLTP systems by supporting stored procedure routing, replicated secondary indexes, and temporal-skew handling. We experimentally prove that these options are important in parallel OLTP systems, and that our approach generates database designs that enable improve performance by up to 16× over other solutions.

11. REFERENCES

[1] H-Store: Next Generation OLTP DBMS Research. `http://hstore.cs.brown.edu`.

[2] Wikipedia MySQL Server Roles. `https://wikitech.wikimedia.org/view/Server_roles`.

[3] S. Agrawal, S. Chaudhuri, A. Das, and V. Narasayya. Automating layout of relational databases. In *ICDE*, pages 607–618, 2003.

[4] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.

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

[6] V. Angkanawaraphan and A. Pavlo. AuctionMark: A benchmark for high-performance oltp systems.

[7] M. Aslett. How will the database incumbents respond to NoSQL and NewSQL? The 451 Group, April 2011.

[8] N. Bassiliades and I. P. Vlahavas. A non-uniform data fragmentation strategy for parallel main-memory database systems. In *VLDB*, pages 370–381, 1995.

[9] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39:12–27.

[10] S. Ceri, S. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Trans. Softw. Eng.*, 9(4):487–504, 1983.

[11] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing SQL workloads. In *SIGMOD*, pages 488–499, 2002.

[12] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. *SIGMOD Rec.*, 27(2):367–378, 1998.

[13] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. In *VLDB*, pages 146–155, 1997.

[14] Y. C. Cheng, L. Gruenwald, G. Ingels, and M. T. Thakkar. Evaluating partitioning techniques for main memory database: Horizontal and single vertical. In *ICCI*, pages 570–574, 1993.

[15] J. Coleman and R. Grosman. Unlimited Scale-up of DB2 Using Server-assisted Client Redirect. `http://ibm.co/fLR2cH`, October 2005.

[16] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in bubba. *SIGMOD*, 17(3):99–108, 1988.

[17] C. Curino, Y. Zhang, E. Jones, and S. Madden. Schism: a workload-drive approach to database replication and partitioning. In *VLDB*, 2010.

[18] E. Danna and L. Perron. Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs. In *Principles and Practice of Constraint Programming*, volume 2833, pages 817–821, 2003.

[19] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTuned. *VLDB*, 2:1246–1257, August 2009.

[20] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 13(1):91–128, 1988.

[21] F. Focacci, F. Laburthe, and A. Lodi. *Handbook of Metaheuristics*, chapter Local Search and Constraint Programming. Springer, 2003.

[22] N. Folkman. So, that was a bummer. `http://blog.foursquare.com/2010/10/05/so-that-was-a-bummer/`, October 2010.

[23] S. Ghandeharizadeh, D. J. DeWitt, and W. Qureshi. A performance analysis of alternative multi-attribute declustering strategies. *SIGMOD*, 21(2):29–38, 1992.

[24] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[25] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, pages 603–614, 2010.

[26] 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.*, 1(2):1496–1499, 2008.

[27] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on Read-Optimized databases using Multi-Core CPUs. *VLDB*, 5:61–72, September 2011.

[28] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. *SIGMETRICS Perform. Eval. Rev.*, 15(1):69–77, 1987.

[29] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*, pages 191–202, 2002.

[30] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, 1997.

[31] R. Mukkamala, S. C. Bruell, and R. K. Shultz. Design of partially replicated distributed database systems: an integrated methodology. *SIGMETRICS Perform. Eval. Rev.*, 16(1):187–196, 1988.

[32] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, SIGMOD, pages 1137–1148, 2011.

[33] C. Nikolaou, A. Labrinidis, V. Bohn, D. Ferguson, M. Artavanis, C. Kloukinas, and M. Marazakis. The impact of workload clustering on transaction routing. Technical report, FORTH-ICS TR-238, 1998.

[34] C. N. Nikolaou, M. Marazakis, and G. Georgiannakis. Transaction routing for distributed OLTP systems: survey and recent results. *Inf. Sci.*, 97:45–82, 1997.

[35] S. Padmanabhan. *Data placement in shared-nothing parallel database systems*. PhD thesis, University of Michigan, 1992.

[36] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, 2004.

[37] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *VLDB*, 5:85–96, October 2011.

[38] E. Rahm. A framework for workload allocation in distributed transaction processing systems. *J. Syst. Softw.*, 18:171–190, May 1992.

[39] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.

[40] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal*, 7(1):48–66, 1998.

[41] J. Sobel. Scaling Out (Facebook). `http://on.fb.me/p7i7eK`, April 2006.

[42] M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54:72–80, June 2011.

[43] 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 *VLDB*, pages 1150–1160, 2007.

[44] M. Stonebraker and A. Pavlo. The SEATS Airline Ticketing Systems Benchmark. `http://hstore.cs.brown.edu/projects/seats`.

[45] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). `http://www.tpc.org/tpcc/`, June 2007.

[46] The Transaction Processing Council. TPC-E Benchmark (Revision 1.23.0). `http://www.tpc.org/tpce/`, June 2010.

[47] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *VLDB*, pages 537–548, 1991.

[48] A. Wolski. TATP Benchmark Description (Version 1.0). `http://tatpbenchmark.sourceforge.net`, March 2009.

[49] D. C. Zilio. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems*. PhD thesis, University of Toronto, 1998.

[50] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.