JECB: a Join-Extension, Code-Based Approach to OLTP Data Partitioning

Khai Q. Tran*
Oracle Labs
khai.x.tran@oracle.com

Jeffrey F. Naughton
University of
Wisconsin-Madison
naughton@cs.wisc.edu

Bruhathi Sundarmurthy
University of
Wisconsin-Madison
bruhathi@cs.wisc.edu

Dimitris Tsirogiannis†
Cloudera
dtsirogiannis@cloudera.com

ABSTRACT

Scaling complex transactional workloads in parallel and distributed systems is a challenging problem. When transactions span data partitions that reside in different nodes, significant overheads emerge that limit the throughput of these systems. In this paper, we present a low-overhead data partitioning approach, termed JECB, that can reduce the number of distributed transactions in complex database workloads such as TPC-E. The proposed approach analyzes the transaction source code of the given workload and the database schema to find a good partitioning solution. JECB leverages partitioning by key-foreign key relationships to automatically identify the best way to partition tables using attributes from tables. We experimentally compare our approach with the state of the art data-partitioning techniques and show that over the benchmarks considered, JECB provides better partitioning solutions with significantly less overhead.

Categories and Subject Descriptors

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

Keywords

OLTP; Partitioning; Parallel; Distributed

1. INTRODUCTION

Exploiting parallelism to improve transaction processing performance has received renewed attention in recent years, perhaps due to the ubiquity of parallel computing platforms ranging from multi-core processors to database appliances to cloud computing services. Scaling transactional workloads through parallelism is limited by the efficacy of the techniques used to distribute data across multiple processing units, for a simple reason: if each compute node in a distributed transaction processing system accesses only local data, there is no need for a distributed concurrency control mechanism. Thus, if we want to execute transactional workloads at scale by exploiting parallelism, we need to partition so as to minimize the number of distributed transactions.

A common approach to solve this problem is to rely on human intuition, that is, to ask the database administrator to propose a partitioning scheme. This works well for simple workloads involving a small number of tables, where the ratio of the insight of the DBA to the inherent difficulty of the problem is high. Unfortunately, for more complex workloads, the situation is harder, and some automated support for chosing partitioning strategies is desirable.

Two pioneering state-of-the-art proposals for automated data partitioning are Schism [11] and Horticulture [17]. Both approaches utilize workload traces and information from the database schema to identify a partitioning solution for a given workload. These approaches are powerful enough to partition arbitrary databases, and in principle, given enough resources and training examples, can always find the best partitioning solution. However, while they perform well when the solution search space is reasonable, in more challenging and complex situations, our experimental results show that they may be overly expensive and/or suboptimal.

In addition to proposing a specific mechanism, one can view our work as exploring the tradeoff between generality and performance. Schism is the most general in that it requires only a workload trace, Horticulture is somewhat less general in that it requires the schema and a workload trace, whereas our approach, which we call "JECB", is the least general, requiring a trace, a schema, and the code for the stored procedures that specify the transactions to be run. We have implemented our approach and tested it over a diverse set of benchmarks, both simple and complex, all of which have been used by previous work investigating transaction processing performance. Our results show that over these benchmarks, our approach never produces worse partitionings than Schism or Horticulture, and often produces substantially better partitionings. This indicates that at least for this class of benchmarks, a computationally simple approach like JECB that exploits source code, schemas, and traces can be very effective.

In our work we found that the use of key-foreign key joins in the context of partitioning transactional workloads is particularly beneficial. To be sure, we are not the first to propose partitioning databases through the use of key-foreign key constraints; key-foreign key partitioning has been explored and even deployed many times previously. However, to the best of our knowledge, no automated way of systematically considering the partitioning space induced by key-foreign key attributes for minimizing distributed

*Work done while the author was at the University of Wisconsin-Madison.

†Work done while the author was at Microsoft Gray Systems Lab.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610532 .

transactions appears in the literature, nor has a comparison of such an approach to Schism and Horticulture been published.

A summary of our contributions is:

- An automatic partitioning method that utilizes transaction source code and the database schema and considers local and foreign key attributes as candidate partitioning attributes.
- A method to search and prune the space of partitioning alternatives.
- A comparison between this method and two state-of-the-art approaches on commonly studied transaction processing benchmarks.

The rest of the paper is organized as follows. In the next section, we describe related work. In Section 3, we present an overview of our approach. The details of our approach appear in Sections 4, 5 and 6. In Section 7, we present experimental results, and we conclude in Section 8 along with a brief discussion on issues of data skew and choice of cost model.

2. RELATED WORK

Horizontal data partitioning has long been used by parallel database systems [12], and has been studied intensively [13, 19, 9, 15, 20]. The goal of partitioning in this work is to exploit partitioned parallelism to speed up read-only workloads. This differs from our focus here — partitioning to avoid multi-site transactions in OLTP workloads — hence the techniques developed in the context of parallel DBMS partitioning are not directly applicable.

As an example, consider two representative efforts at automated partitioning for decision support workloads: [15] and [20]. These techniques consider multiple partitioning options, evaluating each by consulting the query optimizer to arrive at an estimated cost for a given workload over a given partitioning. This may result in a partitioning strategy that would also be effective for a transactional workload, but this is far from certain. A main reason for this is that none of the optimizer cost models employed consider the effect of concurrency control costs due to distributed transactions. Extending the optimizer cost model to apply this kind of solution to partitioning transactional workloads is an interesting area for future work.

In work more directly related to ours, there has been a growing focus on partitioning to support OLTP workloads, in particular Schism [11] and Horticulture [17]. Schism begins by modeling training transactions as a tuple graph, with nodes as tuples and edges between all tuple pairs that are accessed together in some transaction. The graph is then partitioned into the desired number of partitions while minimizing the number of edges cut. The resultant partitioning is input to a machine learning algorithm that trains a classifier to partition arbitrary tuples from the database (not just those seen in the training trace). Schism also proposes exploiting joins to optimize the case where the tuples of one table are frequently accessed via a join path from another table. However, no automated method of selecting joins is presented, nor is there a discussion for handling multiple joins alternatives nor are there experimental results from this approach to considering joins.

In contrast to Schism's learning approach, Horticulture begins by using the database schema to generate proposed partitionings. For each table accessed by the transaction stream, it picks a partitioning attribute and uses a training trace to evaluate its quality. This process is repeated by choosing different partitioning attributes in a "generate and test" search process. Unlike our work, Horticulture uses a sophisticated cost model and addresses the problem of data skew. Their cost function incorporates the number of distributed transactions, along with the total number of partitions touched in the distributed transactions and the workload temporal skew factor. Also, Horticulture proposes an optimization to look into the transaction source code and combine sets of similar queries/transactions in order to improve scalability through workload compression. However, there is no discussion on its effect on the quality of the resulting partitioning solutions.

SWORD [18] is similar to Schism in that it models the workload as a hypergraph over the tuples, compresses the hypergraph, and applies graph partitioning techniques to obtain partitioning solutions. However, the main focus of the work is on incremental re-partitioning, effective workload modeling and compression, and not on algorithms to initially partition the data.

The concept of partitioning via key-foreign key joins is supported by at least one commercial product (Oracle [4]) where it is referred to as partition by reference or REF partitioning [4]. However, this feature is designed for improving the performance of key-foreign joins between two tables, and not for choosing the best data partitioning solution for transactional workloads. Along similar lines, HadoopDB [6] supports referential partitioning to enable faster execution of data warehouse queries. Google's F1 [21] supports hierarchical schemas, which co-locate tuples that join on key-foreign keys.

An extension of the work done by Ceri et al. [8], as described by Zilio [23], partitions a set of relations by first categorizing the relations into primary relations, which are relations with no foreign keys, or secondary relations, which have at least one foreign key. The partitioning keys for the primary relation are selected based on minterms constructed from the query predicates. The partitioning keys for the secondary relations are the foreign keys in the relations that are involved in the most frequent joins with one of these other relations. However, no algorithm/strategy has been published to achieve this partitioning automatically, nor has it been compared to modern solutions such as Schism or Horticulture.

3. OVERVIEW OF JECB

The "JE" in "JECB" stands for "Join Extension." This captures the fact that JECB leverages key-foreign key constraints to propagate partitioning decisions from one table to another. For concreteness and as background for what follows we illustrate the approach an example:

EXAMPLE 1. *Suppose we have a subset of the TPC-E database as shown in Figure 1, where (HS_S_SYMB, HS_CA_ID), T_ID, and CA_ID are the primary keys of the tables HOLDING_SUMMARY, TRADE, and CUSTOMER_ACCOUNT. Furthermore, suppose we have a workload with one transaction type that takes as input a customer identifier and retrieves: (a) the total number of securities that customer holds, and (b) the average number of securities the customer bought in each trade. This transaction class, which we call* CustInfo*, can be implemented by the following stored procedure (note that this is not a real TPC-E transaction class):*

```
CustInfo(@cust_id bigint)
{
    SELECT  SUM(HS_QTY)
    FROM    HOLDING_SUMMARY join CUSTOMER_ACCOUNT
            on HS_CA_ID = CA_ID
    WHERE   CA_C_ID = @cust_id

    SELECT  AVERAGE(T_QTY)
    FROM    TRADE join CUSTOMER_ACCOUNT
            on T_CA_ID = CA_ID
    WHERE   CA_C_ID = @cust_id
```

} *Our goal is to split this database into two partitions such that for any valid value of* cust_id*, the transaction is single-partitioned, that is, all tuples the transaction accesses belong to the same partition.*

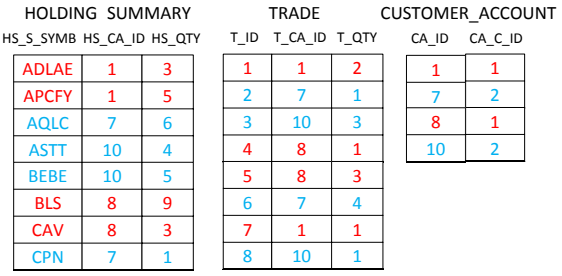

Figure 1: The "CustInfo" Transaction

Consider first the table TRADE. There is no obvious hash or range partitioning that can be applied to any of its three columns T_ID, T_CA_ID, and T_QTY to make transactions single-partitioned. However, this database and transaction class actually has a very natural partitioning strategy: TRADE should be partitioned by column CA_C_ID of the CUSTOMER_ ACCOUNT table by following the key-foreign key join between TRADE and CUSTOMER_ACCOUNT. The same holds for HOLDING_ SUMMARY. The result of this partitioning is shown in Figure 1, where the database is partitioned into two partitions: the red partition for tuples associated with $CA\_C\_ID = 1$*, and the blue partition for tuples associated with* $CA\_C\_ID = 2$*.*

The preceding example is illustrative in another aspect as well: If we partition on column CA_C_ID of the CUSTOMER_ACCOUNT table, and then use this to partition TRADE and HOLDING_ SUMMARY via their foreign key joins with CUSTOMER_ACCOUNT, then executions of the specified transaction will be single partitioned for any instance of the database that adheres to the key-foreign key requirements of the schema.

With this *join-extension* approach, a table can be partitioned by columns of any table connected by one or more key-foreign key joins. Note however, that multiple join paths may exist for a given pair of tables. Therefore, the total number of possible partitioning solutions for a table in our join-extension approach is often bigger than that of approaches that only consider intra-table attributes for partitioning, which makes our search space larger. For example, the TRADE table in TPC-E has only 15 columns, but more than 100 possible join-extension solutions.

Another issue is the overhead of evaluating the cost of each candidate solution. With join-extension partitioning, computing the cost of a specific solution can be expensive since the values of the partitioning attributes may not be available from the trace. To obtain the values of the partitioning attributes for a tuple, we may need to evaluate the key-foreign key join sequence defining the partitioning solution and use it to access the values needed.

We address these two issues by exploiting the following observations. First, OLTP workloads are often composed of a predetermined set of stored procedures. A stored procedure is a transaction template which may already define a specific join path. Hence, by inspecting the SQL code of these templates and the database schema, we can significantly prune the search space of partitioning solutions. We exploit this idea in our approach, and it is the reason for the "CB" ("Code-Based") part of "JECB."

As an example, for the *CustInfo* example in Section 1, we first use the SQL queries to discover that HOLDING_SUMMARY joins CUSTOMER_ACCOUNT via $HS\_CA\_ID = CA\_ID$ and TRADE joins CUSTOMER_ACCOUNT via T_CA_ID=CA_ID, and we then use the workload trace to identify that CA_C_ID is the best partitioning attribute.

Our second observation is that when the workload consists of multiple classes of transactions, the global optimal partitioning solution must be related to the local partitioning solutions of each transaction class. Therefore, we search for the global partitioning solution in the search space created from local partitioning solutions, which is much smaller than the general search space. Horticulture uses a related idea to compress the input to their approach.

Putting these observations together, JECB takes as input a workload trace, the database schema, the SQL code of transaction classes and the number of desired partitions and generates a partitioning solution. JECB has three phases:

- *Phase 1: Pre-processing*. In this phase, we gather information that supports the next two phases. We collect the workload trace, which contains sets of tuples accessed in each executed transaction. Read-only and read-mostly tables, which will be replicated by default, are also identified in this phase.

- *Phase 2: Partitioning individual transaction classes.* To find the best solution for each transaction class, we first use the SQL code for the class and the database schema to construct a graph that connects accessed tables with key-foreign key joins. We then use the workload trace to select connection structures that produce the partitioning solutions with the lowest cost.

- *Phase 3: Combining solutions* After obtaining the partitioning solutions for each transaction type, we combine them in order to find the best global partitioning solution for the database. A table is often accessed in multiple transaction types, thus, it often has multiple potential partitioning solutions. To search for the best combination of partitioning solutions, we employ two heuristics based on the concept of *compatibility* among partitioning solutions.

These phases are elaborated in the next three sections.

A current limitation of our approach is that we only consider key-foreign key joins. As we will see in our experimental section, this is sufficient to perform well on the benchmarks we consider. There may be other workloads for which general joins must be considered. This is an interesting yet potentially challenging area for future work, as general joins represent many-many relationships and using them in partitioning decisions may require something akin to multi-valued dependencies.

Finally, as with any partitioning strategy for OLTP workloads, at runtime one needs to route transactions to partitions. Here we briefly touch on how this can be done.

In general, to route a query or a stored procedure, we select a routing attribute from attributes appearing in selection conditions in WHERE clauses, or from attributes that map to the parameter set of the stored procedure. In the simplest case where the routing attribute matches the partitioning attribute, we simply use the value of the partitioning attribute to decide the partition. However, that does not happen in many cases, especially in our approach since the global partitioning attribute may not belong to every table.

There have been several approaches [22, 14, 7] proposed to address the mismatch problem between partitioning attributes and routing attributes, and among them, the lookup table approach is the one that best fits our solution. A lookup table is a mapping for a table column that maps each value of the column to a set of partition ids that store the corresponding tuples. We can see that the lookup table for the attribute in this case is the mapping function that we have defined in Definition 10. Note that the coarser the attribute, the less space we need to store its lookup table. To route a query or stored procedure, we find a relevant attribute that

is compatible and finer than the partitioning attribute and build a lookup table on it via a join path. If no such attribute exists, then in some fundamental sense the query or stored procedure in question does not match our partitioning, and we are forced to broadcast the query or stored procedure to all partitions.

4. PHASE 1: PRE-PROCESSING

We begin with definitions of transactions and workloads.

DEFINITION 1. *A database D is a set of tables. We represent a transaction by the tuples it reads and writes. Given a database D, a transaction τ is a set of tuples drawn from a read set and a write set, $\tau = R \cup W$, where $R = \{r_1, r_2, ..., r_n\}, r_i \in D$ for $i = \overline{1..n}$, and $W = \{w_1, w_2, ..., w_m\}, w_j \in D$ for $j = \overline{1..m}$.*

A workload *is a bag of transactions. If all transactions of a workload access tuples belonging to the same set of tables via the same SQL queries, we call the workload a* homogeneous workload*. The set of transactions generated by a stored procedure represents a transaction class. A trace of a transaction class is a homogeneous workload, and thus we use "transaction class" and "homogeneous workload" interchangeably.*

The pre-processing phase comprises the following:

Collecting the workload trace: To obtain the tuples accessed within each transaction, we instrument each stored procedure with additional SQL statements to collect tuples accessed in queries of the stored procedure.

Identifying the set of tables that need to be partitioned: Using the workload trace, we identify read-only and read-mostly tables in the workload and the remaining tables will be partitioned. We first replicate read-only tables among all partitions. We also replicate read-mostly tables, which makes transactions updating these tables distributed by default. However, if the percentage of such transactions is small, this replication will not have a substantial affect on the quality of the final partitioning result.

Splitting the trace into different streams: The trace collected from the driver is further partitioned into sub-traces, one for each transaction class.

5. PHASE 2: PARTITIONING

The goal of this phase is to find partitioning strategies for individual transaction classes. To do so, we find solutions that partition every table referenced in an individual transaction class; however, we also consider solutions that only partition some of the tables referenced by the transaction class. These partial solutions may be more useful in the construction of the global strategy than the total solutions, as they are often of finer granularity.

As mentioned in Section 3, we use key-foreign key joins to allow a table T to be partitioned by an attribute of a table other than T. We call the sequence of key-foreign key joins connecting the partitioned table T to the partitioning attribute X the *join path*, defined as follows:

DEFINITION 2. *Given a database D and a table T, a sequence of attribute sets $\{X_0, X_1, ..., X_n\}$ is called a* join path*, denoted by $p(X_0, X_n)$, if:*

1. *X_n consists of only one attribute.*
2. *For every $i \in \{0 \dots n\}$, all attributes of X_i belong to the same table.*
3. *For every $i \in \{0 \dots n-1\}$, if the attributes of X_i and X_{i+1} belong to the same table, then X_i is the primary key of the table. Otherwise, X_i is a foreign key referring to X_{i+1}.*

We call the first node (X_0) and the last node (X_n) the *source* and *destination* nodes, respectively, of the join path. Given a value of X_0, we can uniquely identify a value of X_n via the join path, thus the join path specifies a functional dependency from X_0 to X_n: $X_0 \rightarrow X_n$.

EXAMPLE 2. *In Figure 1, the join paths from the primary keys of HOLDING_SUMMARY and TRADE to CA_C_ID are {{HS_S_SYMB, HS_CA_ID}, HS_CA_ID, CA_ID, CA_C_ID} and {T_ID, T_CA_ID, CA_ID, CA_C_ID}.*

A join path $p(key(T), X)$, where $key(T)$ is the primary key of T, can also be considered to be a mapping from every tuple of T to a unique value of X. To retrieve the value of X for a given tuple of T, we can use a query constructed by following the joins in the join path. Therefore, by connecting a table to columns of other tables, we say that join paths extend the column set of a table.

We observe that often one of the best ways to partition a homogeneous workload is to partition all tables of the workload by a common attribute connected to these tables via join paths. For example, the best way to partition the tables accessed by the *CustInfo* transactions presented in Section 1 is to partition all of them by CA_C_ID. We call the structure that connects tables accessed in a homogeneous workload on the same attribute a *join tree*, which is defined as follows:

DEFINITION 3. *Given a homogeneous workload W and an attribute X, A* join tree *over W and X, denoted $Tree(W, X)$, is a combination of join paths from all tables accessed in W to X. X is called the root of the tree.*

A discussion on choosing of join paths and of cases where join trees cannot be formed is provided in Section 5.2.

For an example of a join tree, refer Figure 2. It represents the join tree for the *CustInfo* transactions presented in Section 1. In this graph, blue nodes represent the source nodes of join paths, i.e., the primary keys of tables, and the red node represents the root of the tree.

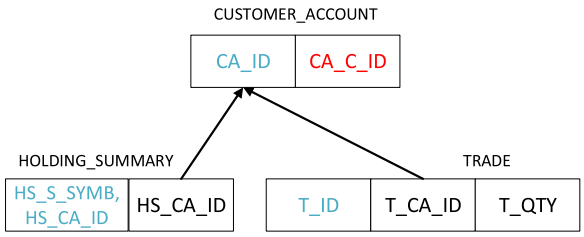

Figure 2: A join tree for *CustInfo* transactions. Blue nodes represent the source nodes of join paths, and the red node represents the root of the tree.

A join tree $Tree(W, X)$ for the workload W maps each tuple t accessed in the workload to a value x of X. Therefore, a partitioning solution for a homogeneous workload is defined by a join tree and a mapping function on X as follows.

DEFINITION 4. *A* partitioning solution *of a homogeneous workload W on a partitioning attribute X has the following two components:*

- *A join tree from W to X, $Tree(W, X)$*
- *A mapping function over X, $f_{k,X}$, that maps each value x of X to an integer $i \in [0..k]$, where k is the number of partitions, and $i = 0$ means all tuples associated with x are replicated.*

The partitioning solution is denoted as $P_W = (Tree(W, X), f_{k,X})$.

Given a partitioning solution, we define a distributed transaction, followed by the definition of the cost of a given partitioning solution.

DEFINITION 5. *Given a database* D*, transaction* $\tau = R \cup W$ *and a partitioning solution* P_D*,* τ *is called a* distributed transaction *if either of the following conditions hold:*

1. $P_D(w) = 0$ *for any* $w \in W$
2. $|\cup_{t \in \tau} P_D(t)| > 1$

The first condition means the transaction writes a replicated tuple, while the second condition means the transaction accesses tuples belonging to more than one partition. If a transaction is not distributed, it is called a *single-partition* or *local* transaction.

DEFINITION 6. *Given a database* D*, a workload* W*, and a partitioning solution* P_D*, the* cost *of* P_D *on* W*, denoted as* $cost(P_D, W)$*, is the percentage of distributed transactions in* W.

Now, for each transaction class, we consider two types of partitioning solutions:

- *Total solutions*: these are partitioning solutions for a homogeneous workload that incur the lowest cost.
- *Partial solutions*: these are partitioning solutions of a workload obtained by eliminating one or more tables from a homogeneous workload. These are not complete solutions for the homogeneous workload, but they have the potential to produce a better global solution when combined with solutions from other homogeneous workloads.

We use the following three steps to find these total and partial solutions:

1. Construct a join graph, which is a graph of tables and columns connected by key-foreign key joins that reflects how tuples are connected to each other in the schema.
2. Enumerate all join trees in the join graph.
3. Find all total and partial solutions from these join trees.

We consider each step in more detail.

5.1 Step 1: Constructing the join graph

We construct the join graph by inspecting the SQL code from the stored procedure of the transaction class. We use the SQL queries in the stored procedures to first identify which tables they access, and also potential candidates for partitioning attribute (candidate attributes), and finally all possible key-foreign key joins to connect the tables.

The tables accessed are those appearing in FROM clauses, and the candidate attributes are attributes appearing in WHERE clauses. After obtaining accessed tables and candidate attributes, we use schema information to identify key-foreign key pairs appearing in accessed attributes and connect the corresponding tables.

EXAMPLE 3. *Suppose we have a following simple query:*

```
SELECT  CA_C_ID
FROM    TRADE, CUSTOMER_ACCOUNT
WHERE   T_CA_ID = CA_ID AND
        T_ID = @t_id
```

Using the procedure above, the tables accessed are TRADE and CUSTOMER_ACCOUNT, and the candidate attributes are T_ID, T_CA_ID, and CA_ID. Since T_CA_ID is a foreign key referring to CA_ID, we connect TRADE and CUSTOMER_ACCOUNT by this key-foreign key join.

However, this approach fails to capture implicit joins. For example, the above query can be rewritten into two separate queries as follows.

```
SELECT  @cust_acct = T_CA_ID
FROM    TRADE
WHERE   T_ID = @t_id
```

and

```
SELECT  CA_C_ID
FROM    CUSTOMER_ACCOUNT
WHERE   CA_ID = @cust_acct
```

To capture implicit joins, we also consider attributes appearing in SELECT clauses to discover other possible key-foreign key joins. This may lead to false-positive joins in our discovery of partitioning strategies, but we will later use the workload trace to eliminate such joins.

EXAMPLE 4. *Figure 3 shows the join graph for the* Customer-Position *transaction type in TPC-E workload. In the graph, tables with blue attributes, which are the table's primary keys, are non-replicated tables.*

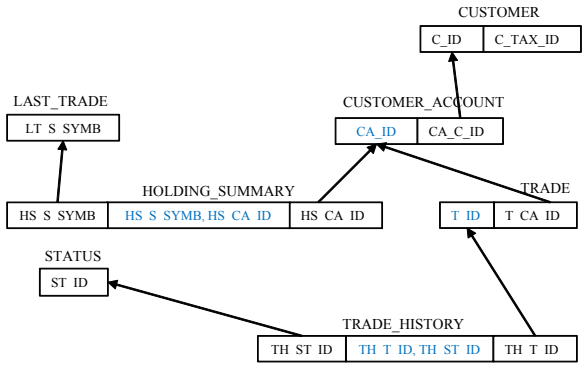

Figure 3: The join graph for the customer-position transaction type in TPC-E. Blue nodes are primary keys of partitioned tables.

5.2 Step 2: Enumerating all join trees

From the join graph obtained from the previous step, we look for *root attributes*, which are attributes reachable from all primary keys of non-replicated tables via join paths in the graph. There are two cases.

Case 1: Such root attributes exist. Now, for each root attribute found, the combination of all possible join paths from each table's primary key to that root attribute comprises all possible join trees. We expect that this is a common case if the workload respects the database schema (all joins are key-foreign key joins).

EXAMPLE 5. *In the join graph of Figure 3, there are four root attributes: CA_ID, CA_C_ID, C_ID, and C_TAX_ID. For each partitioned table, we see that there is a unique join path to each of these attributes. Since CA_C_ID and C_ID refer to the same attribute, we end up with total three join trees for the* Customer-Position *transaction type with the root at CA_ID, CA_C_ID, and C_TAX_ID, respectively.*

Case 2: Such root attributes do not exist. In this case, no total solution for the transaction class exists, since no join trees can be constructed for the workload. Instead, we look for partial solutions by separating the graphs into smaller subgraphs such that join trees can be constructed for each subgraph. The original graph will be split as follows:

- If the graph contains multiple connected components, each connected component will be a new subgraph.
- In each connected component, if there exists an m-to-n relationship, i.e., a non-replicated table has two edges, say a left edge and right edge, that point to other non-replicated tables, then the subgraph is split into two new subgraphs where the first subgraph contains the table and the part with the left edge, and the second subgraph contains the table and the part with the right edge.

EXAMPLE 6. *Suppose in the join graph in Figure 3, LAST_TRADE were also a non-replicated table (it is actually the read-mostly table in TPC-E.) There are now no root attributes among those five non-replicated tables because of the m-to-n relationship in HOLDING_SUMMARY. In this case, we split the graph into two parts: the first part with HOLDING_SUMMARY and LAST_TRADE and the second part will all tables except LAST_TRADE.*

5.3 Step 3: Join tree and mapping function

In this step, we find the best partitioning solution from the set of join trees obtained in the previous step. We look for join trees that map every tuple in each transaction to a unique value of the root attribute. For example, the join tree in Figure 2 maps every tuple of a *CustInfo* transaction in Figure 1 to a unique value of CA_C_ID. Such join trees reflect how tuples in transactions are connected to each other. Also, since the quality of the partitioning solution does not depend on the selection of the mapping function (as long as it does not replicate tuples), we call them *mapping independent* solution. We provide the intuitive meaning of a mapping independent solution following the formal definition.

DEFINITION 7. *Given a homogeneous workload W, a partitioning solution, $P_W = (Tree(W, X), f_{k,X})$ is a* mapping independent solution *for W if for any transaction τ of W, P_W maps all tuples of τ to a unique value x of X.*

Intuitively, mapping independent solutions arise when all tuples accessed by a given transaction are associated with a single value of X. If this is not true, that is, the tuples of the transaction are associated with multiple values of X, then the transactions will be single site only if the multiple values of X associated with the transaction are mapped to the same partition. Hence, the partitioning solution, if one can be found, will depend on the details of the function mapping values of X to partitions. Different functions may succeed at this task to different degrees, so in this case there is a notion of better or worse mapping functions. We call a mapping function that incurs the lowest cost an *optimal mapping function*, and it is more formally defined as follows:

DEFINITION 8. *Given a homogeneous workload W and a join tree $Tree(W, X)$, $f^*_{k,X}$ is an* optimal mapping function *on W and $Tree(W, X)$ if it incurs the lowest cost among all possible mapping functions:*

$$f^*_{k,X}(Tree(W, X), W) = min_f(cost(f_{k,X}, Tree(W, X), W))$$

We observe that for each homogeneous workload, there may be more than one join tree, thus more than one mapping independent solution. Since selecting the global partitioning solution will now be a search on all local partitioning solutions and for workloads consisting of many transaction classes, we need to keep a minimal number of solutions of each transaction class to reduce the search space in the next phase. We do that by merging *compatible* solutions from a single homogeneous workload, which is formally defined as follows.

DEFINITION 9. *Given a homogeneous workload W, the join tree $Tree(W, Y)$ is* compatible *with (and called* coarser *than) the join tree $Tree(W, X)$ if $Tree(W, Y)$ is the combination of $Tree(W, X)$ and a join path from X to Y: $Tree(W, Y) = Tree(W, X) + p(X, Y)$.*

Intuitively, two join trees are compatible if the partitionings specified by one join tree is a refinement of that specified by the other. When a workload has two mapping independent solutions with join trees compatible with each other, the following property will help us to decide which one to choose for the next phase.

PROPERTY 1. *If $Tree(W, Y)$ is coarser than $Tree(W, X)$, and $Tree(W, X)$ is a mapping independent solution of W, then $Tree(W, Y)$ is a mapping independent solution of W.*

This means the mapping-independent property is preserved when moving from finer to the coarser trees.

Now, for each join tree, we use Definition 7 to identify if it is a mapping independent solution. If there are two compatible mapping independent solutions, we eliminate the coarser one. The rationale for this is that, choosing the finer one provides finer partitioning solutions, and also increases the chances of the solution being compatible with solutions from other workloads, as will be required for merging in phase 3.). Finally, we add all non-compatible mapping independent solutions to the set of total solutions.

If the set of total solution is not empty, we also consider all possible partial solutions for each solution from its sub-join trees, which are one or more subtrees obtained by removing the root attribute from the original join tree. For each sub-join tree, if it is still mapping independent over the workload that contains only tuples of tables belonging to the sub-join tree, we add it to the set of partial solutions and continue finding possible solutions from its smaller sub-join trees.

EXAMPLE 7. *Among three join trees for the Customer-Position transaction type, shown in Figure 3, there are two join trees that are mapping independent. These are the trees with roots at CA_C_ID and C_TAX_ID. Since these two trees are compatible, there is a total solution, which is the join tree with the root at CA_C_ID. In this example, there is no partial solution since the closest subtree of the total solution is the join tree with the root at CA_ID, which is not mapping independent. (To see why this is not mapping independent consider two different customer accounts for the same customer. Now, if the root of the tree is the customer account number (CA_ID), it is possible that the two accounts which belong to the same customer are split across different partitions, and hence not mapping independent.)*

If there are no mapping independent total solutions, we need to find the best mapping function that captures clusters of root attribute values appearing within transactions. Our approach to build such a mapping is by using the statistic-based method similar to the one used in Schism [11]. Specifically, the best mapping function for each join tree is built as follows:

- Use the workload trace and join paths to compute the set of values of the root attribute that each transaction accesses.
- Build a graph where each node is a value of the attribute, and edges are used to connect nodes accessed together within transactions. The weight of an edge is the number of transactions that co-access the two nodes.
- Apply a min-cut graph partitioning algorithm to obtain the mapping that maps each attribute value to a partition label.
- The mapping found is considered to be meaningful only if its cost on a new trace is smaller than the costs of both hashing and range-based mapping functions. If there are no join trees with meaningful mapping functions, the transaction class is considered as non-partitonable.

As mentioned earlier, an issue with statistics-based approaches is the lack of scalability in the database size, and we briefly discuss this in Section 7.3. However, JECB may scale better than approaches that scale proportionally to the total number of tuples since JECB scales propotionally to the number of distinct values of the root attributes, which will be no larger and could be much smaller than the total number of tuples.

6. PHASE 3: COMBINING SOLUTIONS

This phase selects a final solution for each non-replicated table such that the global solution produces the lowest cost. A table can be accessed by multiple homogeneous workloads, thus, each table might have multiple associated partitioning solutions.

DEFINITION 10. *A* partitioning solution *for table T on a partitioning attribute X is a pair of components:*

- *A join path from T to X, $p(key(T), X)$;*
- *A mapping function over X, denoted $f_{k,X}$, that maps each value x of X to an integer $i \in [0 \dots k]$, where k is the number of partitions, and $i = 0$ means all tuples associated with x are replicated.*

A partitioning solution for a table is denoted as $P_T = (p(key(T), X), f_{k,X})$.

The set of partitioning solutions for a table consists of all possible solutions found from all transaction classes accessing the table and also the full replication solution. The partitioning solution for a database consists of the partitioning solutions for all its tables.

DEFINITION 11. *A partitioning solution for a database D, denoted P_D, is a collection of partitioning solutions for all of its tables: $P_D = \{P_T, T \in D\}$.*

A naive approach to find the best global solution is exhaustive search of all possible solutions for each table. However, as mentioned in Section 5, exhaustive search does not scale. Accordingly, to reduce the search space, we use the concept of *compatibility* and merge solutions. In Phase 2, we merged compatible join trees for each homogeneous workload, and in this phase we merge compatible solutions for each table in the workload.

We now define the conditions under which two solutions for a table are compatible. These conditions must eliminate one of the two solutions without affecting the quality of the search result, or in other words, one must "include" the other. If two partitioning solutions are compatible, their join paths and partitioning attributes need to be compatible. We first formally define the compatibility of attributes join paths.

DEFINITION 12. *Two attributes X and Y are compatible if either*

- *They participate in a key-foreign key relationship. In this case, they have the same level of granularity, denoted as $X \equiv Y$, or*
- *There exists a join path connecting them. If the join path goes from X to Y, then Y is* coarser *than X, denoted as $Y > X$, and vice versa.*

EXAMPLE 8. *In Figure 2, CA_ID has the same level of granularity as T_CA_ID and HS_CA_ID; CA_C_ID is coarser than T_ID; T_QTY is not compatible with CA_C_ID.*

PROPERTY 2.
- *If $X \equiv Y$ and $Y \equiv Z$, then $X \equiv Z$*
- *If $X > Y$ and $Y > Z$, then $X > Z$*
- *If $X > Y$ and $Y \equiv Z$, then $X > Z$*
- *If $X \equiv Y$ and $Y > Z$, then $X > Z$*

DEFINITION 13. *Given table T and two join paths $p_1(key(T), X)$ and $p_2(key(T), Y)$, suppose that p_2 is not shorter than p_1. These two join paths are called compatible if one of the below conditions hold:*

1. *p_1 is a prefix of p_2.*
2. *$p_1 - X$ is a prefix of p_2 and the attributes X and Y are compatible.*

EXAMPLE 9. *Suppose we have three tables:*

1. *$R1(X, A)$ where X is the key*
2. *$R2(X1, X2, B)$ where $X1, X2$ is the key and they both refer to $R1.X$*
3. *$R3(X1, X2, Y, C)$ where $X1, X2, Y$ is the key, and $X1, X2$ refer to $R2.X1, R2.X2$*

and the following join paths
p_1 = {{R3.X1, R3.X2, R3.Y}, {R3.X1, R3.X2}, {R2.X1, R2.X2}, R2.X1, R1.X, R1.A}
p_2 = {{R3.X1, R3.X2, R3.Y}, {R3.X1, R3.X2}, {R2.X1, R2.X2}, R2.X2, R1.X, R1.A}
p_3 = {{R3.X1, R3.X2, R3.Y}, {R3.X1, R3.X2}, {R2.X1, R2.X2}, R2.X1}
p_4 = {{R3.X1, R3.X2, R3.Y}, R3.X1}
p_5 = {{R3.X1, R3.X2, R3.Y}, R3.X2}

Using the definition, we have:

- *p_1 is not compatible with p_2 because neither condition 1 nor condition 2 holds ($R2.X1 \neq R2.X2$).*
- *$p_1 > p_3$ since condition 1 is satisfied.*
- *$p_4 \equiv p_3$ since condition 2 is satisfied with $R2.X1 \equiv R3.X1$.*
- *p_5 is not compatible with p_1, p_3, or p_4 because neither of the two conditions is satisfied.*

PROPERTY 3. *Given a table T and two compatible join paths $p_1(key(T), X)$ and $p_2(key(T), Y)$ where $p_2 > p_1$ or $p_2 \equiv p_1$, for every pair of tuples $t_1, t_2 \in T$, if $p_1(t_1) = p_1(t_2)$ then $p_2(t_1) = p_2(t_2)$*

We can now define the compatibility of two partitioning solutions and how to merge them:

DEFINITION 14. *Consider a table T and two partitioning solutions on T, $P_1 = ((p_1(key(T), X), f^1_{k,X})$ and P_2=$((p_2(key(T), Y), f^2_{k,Y})$. P_1 and P_2 are compatible with each other if they satisfy both of following conditions:*

- *p_1 and p_2 are compatible.*
- *If $p_2 \equiv p_1$, then one of two solutions needs to be mapping independent. Otherwise, the solution with the finer join path needs to be mapping independent.*

When P_1 and P_2 are compatible, we can merge them into solution P, defined as follows:

- *If $p_2 \equiv p_1$ and suppose that P_1 is mapping independent, then $P = P_2$.*
- *Otherwise, the merged solution P is the one with coarser join path. (We cannot choose the solution with the finer join path as the merged solution, since it would result in the coarser join path not having a meaningful root attribute for partitioning. However, choosing the coarser join path will always result in a meaningful partitioning root attribute for the finer join path, since these 2 paths are compatible.)*

Combining Definition 13 and Definition 14, we have the following property:

PROPERTY 4. *Let T be a table with two compatible partitioning solutions, $P_1 = ((p_1(key(T), X), f^1_{k,X})$ and P_2= $(p_2(key(T), Y), f^2_{k,Y})$. Furthermore, let the merged solution $P = P_2$. Then P_1 is mapping independent, and there exists a specific mapping function for P_1 such that for every tuple t: $P_1(t) = P_2(t) = P(t)$.*

Proof: From Definition 14, we can see that P_1 needs to be mapping independent because if it is not, the two solutions will not be compatible. Using Definition 13, we define the specific mapping function $f^1_{k,X}$ for P_1 as follows:

1. If $p_2 \equiv p_1$, then $f^1_{k,X} = f^2_{k,Y}$.
2. If $p_2 > p_1$, then $f^1_{k,X} = p(X, Y) \circ f^2_{k,Y}$ where $p(X, Y)$ is a part of p_2 that connects X (or X') to Y.

For every tuple t we have:

1. If $p_2 \equiv p_1$, then $P_1(t) = f^1_{k,X}(p_1(t)) = f^2_{k,Y}(p_1(t)) = f^2_{k,Y}(p_2(t)) = P_2(t)$.
2. If $p_2 > p_1$, then $P_1(t) = f^1_{k,X}(p_1(t)) = (p(X, Y) \circ f^2_{k,Y})(p_1(t)) = f^2_{k,Y}(p(X, Y) \circ p_1(t)) = f^2_{k,Y}(p_2(t)) = P_2(t)$

This property implies that we can use compatible solutions interchangeably without affecting the quality of the partitioning.

After merging compatible solutions, we apply another technique of *searching only around compatible attributes* to arrive quickly at the partitioning solution. This technique can be illustrated by the following example. Suppose we have two tables *A* and *B*, and for table *A*, we already selected a solution with partitioning attribute *X*. We observe that, among solutions of *B*, solutions that have partitioning attributes compatible with *X* produce better results when combined with the selected solution of *A* than solutions whose partitioning attributes are not compatible with *X*.

Choosing solutions with compatible attributes is desirable because it produces join trees that are compatible; that is, it does not give rise to two contradictory partitioning solutions.

Using the two described techniques, we our search algorithm works as follows.

Step 1: Finding partitioning attributes From solution sets of tables, we find all partitioning attributes that are not compatible with each other. If two attributes are compatible, we select the coarser one.

Step 2: Enumerating all global solutions associated with each attribute Given a partitioning attribute X, we first find the solution set associated with X, called the *reduced solution set*, for each non-replicated table. Then we enumerate all possible combinations associated with X from these reduced solution sets.

To find the reduced solution set associated with X for table T, we add to the reduced solution set all solutions that have the partitioning attribute compatible with X. Among these solutions, if any two of them are compatible with each other, we merge them using Definition 14. After that, if there is any solution with a partitioning attribute other than X, we use the shortest join path to extend its join path to X. If the solution set is empty, we add the full replication solution.

Step 3: Searching for the best global solution We evaluate the *cost* of a partitioning solution for a specific workload by the percentage of *distributed transactions* that would result from executing the workload with the given partitioning (Defined in Section 5.)

For all partitioning solutions obtained from step 2, we apply a global trace, i.e., a trace consists of all types of transactions, to get their costs, and select the one with the lowest cost.

EXAMPLE 10. *In TPC-E benchmark, after replicating all read-only and read-mostly tables, we have ten non-replicated tables accessed by ten transaction types. After running the first phase, we get a search space of about 2.6 million combinations. However, our two heuristics help to reduce the search space to only twelve combinations over four partitioning attributes: C_ID, B_ID, T_S_SYMB, and T_DTS. Among them, the solution of partitioning all tables via attribute C_ID incurs the lowest cost of 21% distributed transactions for eight partitions.*

7. EXPERIMENTAL EVALUATION

In this section, we evaluate and compare the performance and scalability of our approach, JECB, to Schism [11] and Horticulture [17].

7.1 Data partitioning evaluation framework

To evaluate the scalability and performance of a data partition approach, we built a general framework, shown in Figure 4.

In this framework, we used a *trace collector* to collect the workload trace. As described in Section 4, we injected a SQL statement right after each query in stored procedures to capture the tuples accessed in the query. For each tuple, we capture its table name, its primary key, the transaction id, and whether it is read or updated. Compared to the approach of rerunning transactions from the query log, this method lessen errors arising due to stale data. The collected trace is then split into two parts: a training trace and a testing trace. The training trace is fed into a *partitioner* along with the database schema and the transaction's SQL code. The partitioner uses this information depending on the given data partitioning algorithm to find the best partitioning strategy for the database. The testing trace is used to measure the performance of a given data partitioning algorithm. We use a *partitioning evaluator* that applies the partitioning solution produced from the partitioner on

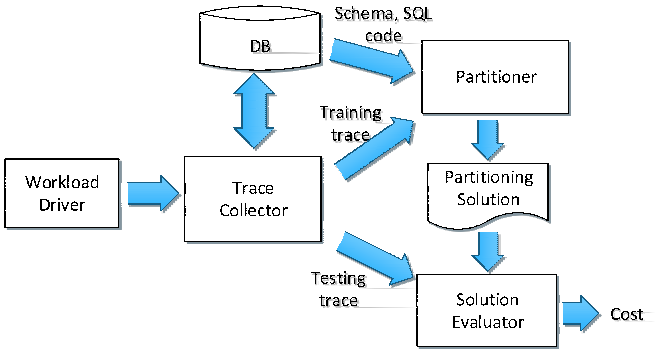

Figure 4: Data partitioning evaluation framework

the testing trace to compute the cost, as given in Definitions 5 and 6.

In our experiments, we implemented two different data partitioning algorithms for the partitioner: JECB and Schism. We did not implement Horticulture since the partitioning solution obtained by Horticulture for each workload is deterministic, i.e., it does not depend on the size and the state of the database. It provides the partitioning column and the partitioning function (usually hashing) for each table; thus, for each workload, we directly apply the partitioning solution found in [17].

7.2 Experimental settings

We implemented and executed the data partitioning evaluation framework on an Intel Xeon L5630, 2.13GHz machine with 32GB of RAM, running Microsoft SQL-Server.

As mentioned in Section 1, one of the main problems with tuple-based approaches is scalability with database size. Experiments in [11] show good results only for small database sizes with limited numbers of partitions, and the authors of that paper also qualitatively state that the performance of Schism depends on workload complexity, database size, and number of partitions. Therefore, in the first experiment, we quantitatively examine the performance and scalability of Schism and JECB on bigger databases with more partitions. In this experiment, we use TPC-C since it is known to be best partitioned by *warehouse_id* for all non-replicated tables.

In the next experiment, we compare the quality of partitioning solutions obtained by JECB, Schism, and Horticulture on a variety of benchmarks, including TPC-C, TPC-E, TATP and two others in the OLTP benchmark suite [10]: SEATS and AuctionMark. We then explore in more detail one of the more complex benchmarks, TPC-E. Finally, we discuss the performance of JECB on synthetic workloads where our assumption that all joins are key-foreign key joins does not hold.

7.3 Scalability in database size

In this experiment, we first applied Schism to partition 128 and 1024 warehouse TPC-C databases into various numbers of partitions. In Figure 5, we present the result of running Schism with training sets of 30K, 180K, and 400K transactions, which cover 1%, 5%, and 10% of the initial 128-warehouse database. Also, in Figure 6, we see the result of running Schism with training sets of 40K and 110K transactions, which covers 0.1% and 0.2% of the initial 1024-warehouse database.

From these graphs we can see that JECB is independent of the database size and the number of partitions and always produces solutions matching the best partitioning solutions (partitioning on warehouse ID). On the other hand, Schism produces good partitioning solutions only for small number of partitions when the training set is big enough.

Approach	RAM (MB)	CPU (seconds)
schism 1%	692	232
schism 5%	4442	577
schism 10%	9774	1870
JECB	30	35

Table 1: Resource consumption for partitioning TPC-C 128-warehouse database

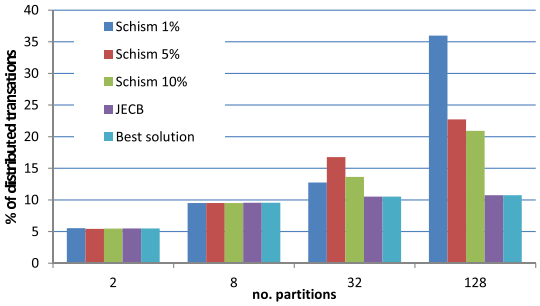

Figure 5: TPC-C 128 warehouses

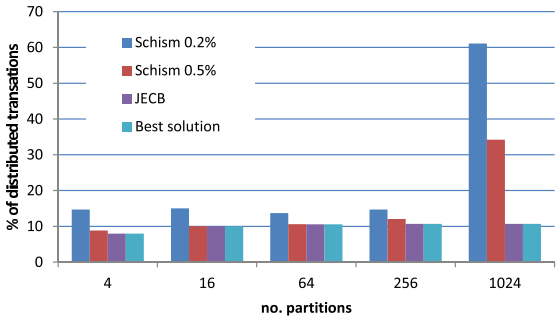

Figure 6: TPC-C 1024 warehouses

Both graphs show that Schism's partitioning quality increases as the size of the training set increases. In the extreme case, where the number of partitions equals the number of warehouses, the errors in Schism's partitioning come from the approximate nature of the min-cut graph partitioning algorithm used.

In our experiments, Schism could not find a good partitioning solution in extreme cases for both databases due to restrictions on our machine resources. Tables 1 and 2 show the resource consumption for experiments with 128- and 1024-warehouse databases, respectively. For the 128-warehouse database, the graph partitioning phase is the major source of resource consumption while for 1024-warehouse database, the explanation phase is the major source of resource consumption. On the other hand, the resource consumption of JECB does not depend on the size of the database or the number of partitions.

Scalability of JECB.

As discussed in Section 5.3, in the event of not finding any mapping independent total solutions, we fall-back to the statistics-based approach. Of course, the problem of lack of scalability in the database size for statistics based approaches applies here too. But, it is less severe in our context since the number of distinct values of the root attribute is often smaller than the number of tuples

Approach	RAM (MB)	CPU (seconds)
schism 0.1%	5285	1250
schism 0.2%	30252	3870
JECB	30	36

Table 2: Resource consumption for partitioning TPC-C 1024-warehouse database

in the corresponding table, which in turn is often much smaller than the total number of tuples in the whole database. However, it is important to note that in all five common OLTP benchmarks we experimented with, most transaction classes had mapping independent partitioning solutions and the remaining classes were non-partitionable.

To summarize, one can view the results of the section as highlighting the penalty Schism pays for generality. There is a great deal of information available in the database schema and transaction source code, and Schism ignores this information. Of course, for scenarios in which the schema and source code are not available, Schism has the advantage, as JECB does not apply.

7.4 Partitioning quality on benchmarks

In the next experiment, we applied JECB on various benchmarks and compared the quality of our partitioning solutions with those obtained by Schism and Horticulture. We used 5 benchmarks: TPC-C, TPC-E, TATP, SEATS, and AuctionMark. This first three benchmarks are popular OLTP benchmarks while the last two are from the OLTP benchmark suite [10].

In each benchmark, we obtained partitioning solutions for JECB and Schism with 10% coverage. Then we applied these solutions along with the solution from Horticulture, which was supplied by authors of [17], on the testing trace to obtain the partitioning quality for each approach. Since the performance of Schism depends on the number of partitions and the training set coverage, we fixed the training set to 10% and the partition number to 8 for all benchmarks. The workload details and experiment results for each benchmark are described as follows.

TPC-C: This is an OLTP standard benchmark for simulating an order processing system [2]. Figure 7 shows that solutions from all three approaches have the same quality. These solutions are actually the same; all tables except the *item* table are partitioned by *warehouse id*, which is also the popular way of partitioning TPC-C.

TATP: This is an OLTP benchmark that simulates a typical telecommunication systems [16]. Similar to TPC-C, this workload is easy to partition since most tables in TATP are accessed via a common attribute, *subscriber id*. Figure 7 shows both JECB and Horticulture are able to obtain this solution. However, Schism 10% produces a solution with 22.6% of distributed transactions. We observe that in this case the graph partitioning algorithm worked well, producing a total edge cut of 0, and that the actual reason for errors is from the high cardinality of the classification attribute. Since the workload trace of 10% coverage, consisting of 70K transactions, does not cover the 100K distinct values of the classification attribute, *subscriber id*, it cannot produce the correct partitioning rule for each individual value.

SEATS: This is an OLTP benchmark for modeling an airline ticketing system [1]. Looking at Figure 7, we see that there is a significant difference between the partitioning quality of JECB and that of Horticulture. Unlike the two previous benchmarks, there is no common attribute among non-replicated tables, which means different transaction types have different optimal partitioning attributes. JECB uses join paths to connect all non-replicated tables to a common attribute, and is able make the workload completely partitionable. Schism's performance can be understood for the same reasons as its performance on TATP.

Transaction class	Mix	Total solutions	Partial solutions
Broker-Volume	4.9%	No	No
Customer-Position	13%	CA_C_ID	No
Market-Feed	1%	No	No
Market-Watch	18%	HS_CA_ID	No
Security-Detail	14%	Read-only	Read-only
Trade-Lookup Frame1	2.4%	No	No
Trade-Lookup Frame2	2.4%	CA_ID	No
Trade-Lookup Frame3	2.4%	T_S_SYMB or T_DTS	No
Trade-Lookup Frame4	0.8%	CA_ID or T_DTS	No
Trade-Order	10.1%	B_ID	CA_ID
Trade-Result	10.0%	B_ID	CA_ID
Trade-Status	19.0%	B_ID	CA_ID
Trade-Update Frame1	0.66%	No	No
Trade-Update Frame2	0.67%	CA_ID or T_DTS	No
Trade-Update Frame3	0.67%	T_S_SYMB or T_DTS	No

Table 3: Transaction classes and solutions found by JECB for TPC-E (after Phase 1)

AuctionMark: This is an OLTP benchmark that models an Internet auction system [5]. Similar to TPC-C and TATP, non-replicated tables in AuctionMark are often accessed by the common attribute of *user id*. However, there are two types of users, *sellers* and *buyers*, and there exist m-to-n relationships among them in some transaction types, which makes the workload not completely partitionable. We can see in Figure 7 that the partitioning provided by JECB is better than Schism 10% and approximately the same as Horticulture.

TPC-E: This benchmark is more complicated than TPC-C and is designed to model modern OLTP applications [3]. The database schema consists of 33 tables with a total of 188 columns and 50 foreign keys, and its workload features 10 types of complex transactional activities, which are often composed of multiple SQL queries that access multiple tables. We observe in Figure 7 that both Schism and Horticulture perform badly on TPC-E, while JECB is sill able to produce a solution in less than two minutes with only 21% distributed transactions. The benchmark will be presented in more detail in Section 7.5.

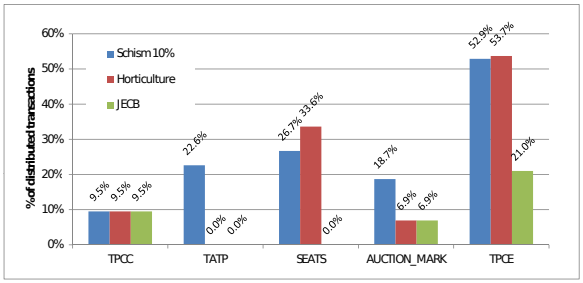

Figure 7: Partitioning qualities on different benchmarks

We now explore the details of the TPC-E benchmark and explain the relative performance between JECB and Horticulture.

7.5 TPCE details

The TPC-E benchmark emulates the activities of a brokerage firm, which include managing customer accounts, executing customer trade orders, and managing the interaction between customers and financial markets. There are a total of 10 types of activities, which are decomposed into 15 classes of transactions, as summarized in Table 3.

In this table, the "mix percentage" of a transaction class is the frequency that these transactions are executed in the workload. Among 15 transaction classes, there are 6 transaction classes updating the database: *Market-Feed, Trade-Order, Trade-Result*, and three *Trade-*

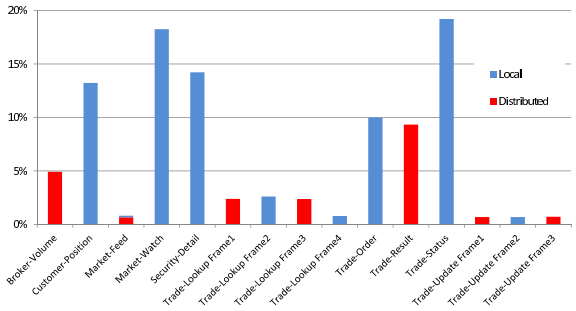

Figure 8: Join-extension on TPCE

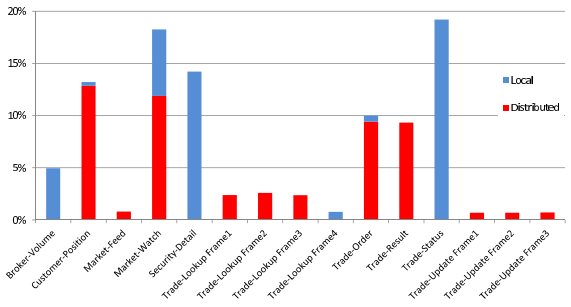

Figure 9: Horticulture on TPCE

Table	HC	Join-extension
ACCOUNT_PERMISSION	AP_CA_ID	replicated
CUSTOMER_TAXRATE	CX_C_ID	replicated
DAILY_MARKET	DM_DATE	replicated
WATCH_LIST	WL_C_ID	replicated
CASH_TRANSACTION	CT_T_ID	CT_T_ID → T_ID → T_CA_ID → CA_ID → CA_C_ID
CUSTOMER_ACCOUNT	replicated	CA_C_ID
HOLDING	H_CA_ID	H_T_ID → T_ID → T_CA_ID → CA_ID → CA_C_ID
HOLDING_HISTORY	HH_T_ID	HH_T_ID → T_ID → T_CA_ID → CA_ID → CA_C_ID
HOLDING_SUMMARY	HS_CA_ID	(HS_CA_ID, HS_S_SYMB)→ HS_CA_ID→CA_ID→CA_C_ID
SETTLEMENT	SE_T_ID	SE_T_ID → T_ID → T_CA_ID → CA_ID → CA_C_ID
TRADE	T_CA_ID	T_ID → T_CA_ID → CA_ID → CA_C_ID
TRADE_HISTORY	TH_T_ID	TH_T_ID → T_ID → T_CA_ID → CA_ID → CA_C_ID
TRADE_REQUEST	replicated	TR_T_ID → T_ID → T_CA_ID → CA_ID → CA_C_ID

Table 4: Partitioning solutions for TPC-E. All columns are partitioned by hashing.

Update transaction classes; the remaining transactions are read-only.

Table 4 shows the partitioning solutions obtained by Horticulture and by JECB. Among 33 tables, the first 23 tables are read-only or read-mostly. For the remaining 10 tables, our solution replicates the BROKER table and partitions all remaining tables by the C_ID attribute (the primary key of the CUSTOMER table).

The performance of our partitioning solution for each transaction class is shown in Figure 8. In the graph, there are seven transaction classes on which our solution does not perform well. To understand why our solution does not perform well on these classes, we list all total and partial solutions found in the first phase of our algorithm for each transaction class in Table 3 (for brevity, we do not show the whole join trees, but only their root attributes instead.) Using this table, we divide these seven transaction classes into two groups:

1. The first group, including *Broker-Volume*, *Market-Feed*, *Trade-Lookup Frame1*, and *Trade-Update Frame1*, are transaction classes that are not partitionable (we do not consider the trivial solution of replicating all tables since that would make transactions updating them distributed). The common property of all transactions in this group is that each transaction is always mapped into a set of values of the partitioning attribute, which are actually values of stored procedure input parameters, thus there are no mapping-independent solutions for such transactions. In addition, because these values are randomly chosen, there are no patterns or clusters among these values. So, there is effectively no way to partition such transactions.

2. The second group, including *Trade-Lookup Frame3*, *Trade-Result*, and *Trade-Update Frame3*, are transaction classes that each have their own mapping independent solutions, but their partitioning attributes are incompatible with C_ID, which is the attribute chosen in the second phase of our algorithm as the one that incurs the lowest cost. In this group, *Trade-Lookup Frame3* and *Trade-Update Frame3* are both partitioned by S_SYMB (primary key of SECURITY table), but since the mix percentages of these transaction classes are small, the main candidates are C_ID and B_ID (primary key of BROKER table). Because the mix percentage of *Customer-Position* is greater than that of *Trade-Result*, C_ID is the final winner.

Comparing to the performance of the Horticulture solution, shown in Figure 9, our solution is better on most transaction classes. The transaction class where Horticulture performs better is on *Broker-Volume*, but their solution is to replicate both non-read-only tables of the class: BROKER and TRADE_REQUEST. Replicating TRADE_REQUEST makes *Trade-Order* transactions, which update TRADE_REQUEST, also distributed, and as the mix percentage of *Trade-Order* is greater than that of *Broker-Volume*, this replication increases the overall rate of distributed transactions. Also, Horticulture solution does not perform well on *Customer-Position*, *Market-Watch*, *Trade-Lookup Frame2*, and *Trade-Update Frame2* transaction classes, which are completely partitionable by our solution.

7.6 Synthetic workloads

In this subsection we briefly touch upon the performance of our approach on workloads that do not respect the database schema (that is, some joins are not key-foreign key joins). To do so, we experimented with synthetic workloads on a simple database with 1-to-n relationships in the schema. Our workload consists of two transactions classes, one which respects the database schema, and the other which has implicit joins. We vary the mix percentages of these two transaction type and fix the number of partitions at 100 in all runs. Our observation is that the join-extension approach performs well when transactions that respect the database schema dominate in the workload. On the other hand, the quality of the column-based solutions decrease as the percentage of transactions with implicit joins decreases and the column-based approaches only perform well when these transactions dominate in the workload.

8. CONCLUSION

We have presented JECB, an approach for automatically partitioning OLTP databases. We leverage join-paths, made of a sequence of key-foreign key joins, to enlarge the search space to capture partitioning solutions that might be missed if they were not considered. We then use a divide-and-conquer strategy, the database schema, and the transaction SQL code to direct our search algorithm to reach a good partitioning solution. Our experimental results suggest that JECB is able to produce better partitioning so-

lutions more quickly than previous approaches on many popular benchmarks, especially on more complicated benchmarks including TPC-E.

Substantial room for future work remains. One issue that always arises with parallel performance is skew. Alleviating the effects of "hot" and "cold" partitions on performance will be important in some cases. A promising idea is to partition the database into many more partitions than processing elements; thus, each processing element (a core, or node) can have different numbers of partitions mapped to it. A heuristic bin packing that does so while considering the heat of partitions might alleviate the impact of skew on performance, but of course this needs to be fully developed and evaluated.

Another direction for future work is the exploration of more complex cost models. Our cost model in this work was extraordinarily simple: minimize the fraction of distributed transactions. Of course, many more complex cost models can be and should be considered, including models that take into account the number of sites over which a transaction is distributed, the relative running times of local vs. distributed transacitons, and so forth. It would be interesting to consider a spectrum of increasingly complex cost functions and to see under what scenarios they produce significantly better results.

Finally, as we have alluded to in the introduction, JECB can be viewed as trading generality for performance — by requiring knowledge about the source code, schema, and workload, JECB can do better on the benchmarks that we considered than SCHISM (which only needs the workload) and Horticulture (which needs only the workload and schema). An empirical study of how often each type of partitioning approach is applicable in practice, as well as the identification of other solutions that are perhaps even less general (can we improve partitioning by analyzing the application program in which the transactions are embedded?) would be interesting.

Acknowledgement

This work was supported by a grant from the Microsoft Gray Systems Lab.

9. REFERENCES

[1] The SEATS Airline Ticketing Systems Benchmark. *http://hstore.cs.brown.edu/projects/seats/*.

[2] The Transaction Processing Council. TPC-C Benchmark (version 5.11). *http://www.tpc.org/tpcc/*.

[3] The Transaction Processing Council. TPC-E Benchmark (version 1.12). *http://www.tpc.org/tpce/*.

[4] Partitioning with Oracle Database 11g Release 2 *http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-partitioning-11gr2-2011-12-1392415.pdf*. *Oracle White Paper*, Dec. 2011.

[5] AuctionMark: An OLTP Benchmark for Shared-Nothing Database Management Systems. *http://hstore.cs.brown.edu/projects/auctionmark/*. Nov. 2012.

[6] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '11, pages 1165–1176, New York, NY, USA, 2011. ACM.

[7] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):4–24, Mar. 1990.

[8] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *Proc. of the 1982 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '82, pages 128–136, New York, NY, USA, 1982. ACM.

[9] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of data*, SIGMOD '98, pages 367–378, New York, NY, USA, 1998. ACM.

[10] C. Curino, D. E. Difallah, A. Pavlo, P. Cudre-Maroux, E. Jones, Y. Zhang, and S. Madden. OLTP Benchmarks. *http://www.oltpbenchmark.com*.

[11] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, Sept. 2010.

[12] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

[13] S. Ghandeharizadeh and D. J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proc. of the 16th Intl. Conf. on Very Large Data Bases*, VLDB '90, pages 481–492, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[14] S. Ghandeharizadeh and D. J. DeWitt. Magic: A multiattribute declustering mechanism for multiprocessor database machines. *IEEE Trans. Parallel Distrib. Syst.*, 5(5):509–524, May 1994.

[15] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of data*, SIGMOD '11, pages 1137–1148, New York, NY, USA, 2011. ACM.

[16] S. Neuvonen, A. Wolski, M. manner, and V. Raatikka. Telecom Application Transaction Processing Benchmark. *http://http://tatpbenchmark.sourceforge.net/*.

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

[18] A. Quamar, K. A. Kumar, and A. Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proc. of the 16th Intl. Conf. on Extending Database Technology*, EDBT '13, pages 430–441, New York, NY, USA, 2013. ACM.

[19] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of data*, SIGMOD '02, pages 558–569, New York, NY, USA, 2002. ACM.

[20] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '02, pages 558–569, New York, NY, USA, 2002. ACM.

[21] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, Aug. 2013.

[22] A. L. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *Proc. of the 2012 IEEE 28th Intl. Conf. on Data Engineering*, ICDE '12, pages 102–113, Washington, DC, USA, 2012. IEEE Computer Society.

[23] D. C. Zilio. Physical database design decision algorithms and concurrent reorganization for parallel database systems. Technical report, 1997.