Sharing-Aware Horizontal Partitioning for Exploiting Correlations During Query Processing

Kostas Tzoumas
Aalborg University
Denmark
kostas@cs.aau.dk

Amol Deshpande
University of Maryland
College Park, MD, USA
amol@cs.umd.edu

Christian S. Jensen
Aarhus University
Denmark
csj@cs.au.dk

ABSTRACT

Optimization of join queries based on average selectivities is suboptimal in highly correlated databases. In such databases, relations are naturally divided into partitions, each partition having substantially different statistical characteristics. It is very compelling to discover such data partitions during query optimization and create multiple plans for a given query, one plan being optimal for a particular combination of data partitions. This scenario calls for the sharing of state among plans, so that common intermediate results are not recomputed. We study this problem in a setting with a routing-based query execution engine based on eddies [1]. Eddies naturally encapsulate horizontal partitioning and maximal state sharing across multiple plans. We define the notion of a *conditional join plan*, a novel representation of the search space that enables us to address the problem in a principled way. We present a low-overhead greedy algorithm that uses statistical summaries based on *graphical models*. Experimental results suggest an order of magnitude faster execution time over traditional optimization for high correlations, while maintaining the same performance for low correlations.

1. INTRODUCTION

Traditional query optimizers pick one execution plan per query, based on first-order statistics about the underlying data. In particular, a join order is determined based on join selectivities that are computed over a relation as a whole. However, real-world databases often contain skewed data with complex correlations, and first-order statistics are not sufficiently powerful to capture the underlying statistical properties of the data. Indeed, one can get better join selectivity estimates by modeling data correlations [6, 13]. However, the presence of data correlations does not only make selectivity estimation harder–it also offers opportunities for more effective query optimization.

When data correlations are present, the input relations are naturally divided into partitions, each partition having completely different statistical characteristics. It is then very attractive to create *multiple plans* per query, each plan being optimized for a different combination of data partitions. Consider for example the join query $R \bowtie S \bowtie T \bowtie U$. Assume that S is naturally partitioned into two partitions, $S = S_1 \cup S_2$, where S_1 (similarly, S_2) has a low selectivity when it joins with R (T), and a high selectivity when it joins with T (R). A possible optimization process may decide to partition S into S_1 and S_2, and pick the plans $(R \bowtie S_1) \bowtie (T \bowtie U)$ and $((T \bowtie U) \bowtie S_2) \bowtie R$. The combined cost of the two resulting plans can be smaller than the cost of any possible monolithic plan.

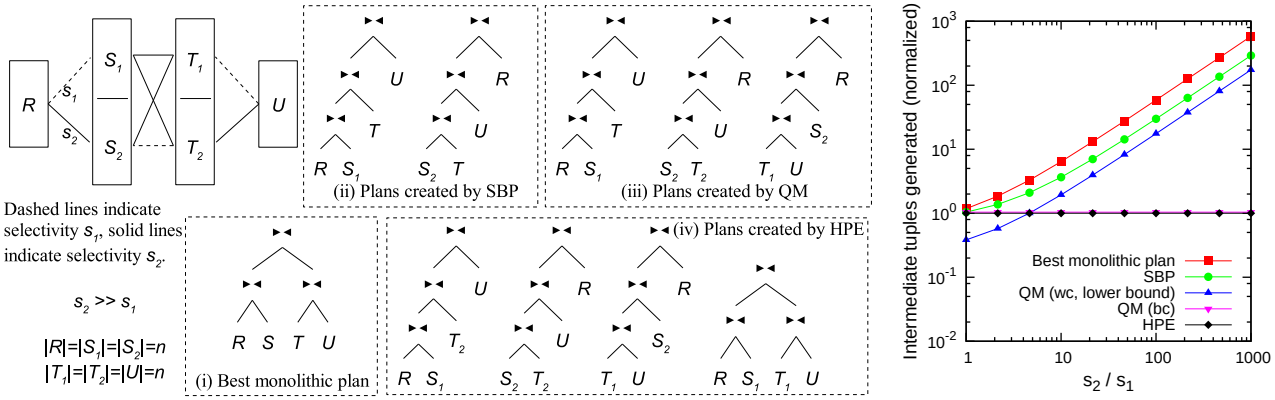
With the introduction of partitioning, query optimization consists of two tasks: Determining the partitions of the input relations, and creating a plan for each combination of partitions. Unfortunately, the two problems are inter-dependent. A partitioning of the relations is optimal only with respect to already chosen join plans. A partitioning is query-plan specific, because it is evaluated against the selectivities of the joins in the join plans; it is not merely a set of clusters based on the statistical properties of the data. Conversely, a collection of join plans is optimal only with respect to a certain partitioning. This inter-dependence yields a much larger optimization space than the one considered by traditional query optimizers.

Further, an optimization process that results in multiple plans per query naturally raises the issue of sharing state among the constituent plans at execution time. Identical intermediate tuples should not be constructed multiple times from different plans during query execution. In the example above, the intermediate relation $T \bowtie U$ is required in both plans. This relation should not be constructed twice; rather, it should be shared between the two plans.

This paper presents the first study of horizontal partitioning during query processing with *maximal* sharing of intermediate results. In particular, the contributions of this paper are the following: First, we offer a more formal study of the general problem than hitherto. We introduce the notion of *conditional join plans* (CJPs), a representation of the search space resulting from horizontal partitioning that captures both the partitioning and join order aspects. We define recursive cost formulas for CJPs, and are thus able to define query optimization as a search problem in a suitable space. In addition, we show how to estimate correlated join selectivities using low-overhead summaries based on graphical models. Then, we focus on the case of query execution with eddies [1] and symmetric hash joins. This case is particularly interesting, because sharing is maximal; an intermediate tuple that is used by different join plans is computed only once. We show how query execution with eddies restricts the search space, and we provide a low-overhead greedy algorithm for this space. Our algorithm can achieve an order of magnitude better execution time than the best monolithic plans in databases with high correlations, while being on par with traditional query optimization for uniform data.

The rest of this paper is organized as follows. Section 2 reviews related work and eddies. Section 3 defines conditional join plans and how to estimate their cost, including the estimation of corre-

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



(a) Running example query and the join plans chosen by a (i) a traditional query optimizer, (ii) selectivity-based partitioning [18], (iii) query mesh [17], and (iv) our approach.

(b) Intermediate tuples generated (normalized).

Figure 1: A 3-join query $R \bowtie S \bowtie T \bowtie U$ used as a running example throughout the paper.

lated selectivities. Section 4 describes how maximal sharing of intermediate results restricts the search space, and provides a greedy search algorithm. Section 5 presents experimental results. Finally, Section 6 concludes and offers research directions.

2. BACKGROUND

2.1 Related work

Horizontal partitioning of relations has been considered in many settings, especially in parallel and distributed databases [10]. Prior work has addressed horizontal partitioning for selection queries [7], and explored heuristic solutions in the adaptive setting [2]. The works most relevant to ours are selectivity-based partitioning [18] and query mesh [16, 17]. We discuss the points that differentiate our work from these.

In selectivity-based partitioning [18], an iterative algorithm can partition *one* relation of a join query into k partitions and construct k *left-deep* join plans, one for each partition. Our work does not have these limitations. In particular, we allow for bushy plans and the partitioning of multiple, both base and intermediate, relations. In addition, state sharing among the resulting plans is not considered in [18]. The plans are executed independently, so intermediate results that are common are computed multiple times, as opposed to exactly once which is the case in our work.

Query mesh [16, 17] allows the partitioning of multiple relations. However, intermediate results are always recomputed and never stored, in a similar manner to SteMs [19]. This has two implications: First, bushy plans are not allowed, resulting in missed opportunities for certain queries. Second and more important, the join plan followed depends on the arrival order of the input tuples (see [9], Section 6.2). In some cases, although query mesh will try to partition the input relations, the chosen partitioning is not obeyed by the execution engine, due to the tuple arrival order. Our work does not suffer from these limitations, as intermediate results are stored and shared. A discussion of the benefits of storing intermediate results as opposed to recomputing them can be found in the literature [8].

We illustrate the advantages of our more general problem setting with an example. Assume the following schema and join query:

$$R(A, \mathsf{X}), S(A, \mathsf{Y}, B), T(B, \mathsf{Z}, C), U(\mathsf{W}, C)$$

```
select * from R,S,T,U
where R.A = S.A and S.B = T.B and T.C = U.C
```

Figure 1(a) shows this query, which is used as a running example throughout the paper. Relations S and T are naturally divided into two partitions each. S_1 has low selectivity, s_1, when joining with R and high selectivity, s_2, when joining with T. T_1 has low selectivity when joining with U, and high selectivity when joining with S. It is then attractive to first join S_1 with R, and first join T_1 with U.

The plans generated by the methods discussed above are shown in Figure 1(a). **(i)** The best monolithic plan is the bushy plan that first joins R with S, and T with U. **(ii)** Selectivity-based partitioning (SBP) can only partition one relation and is forced to use left-deep plans. Since the query is symmetric, the choice between S and T does not matter. So, assuming that S is partitioned, the resulting plans are $((R \bowtie S_1) \bowtie T) \bowtie U$ and $((S_2 \bowtie T) \bowtie U) \bowtie R$. **(iii)** Query mesh (QM) can partition both relations, but can only use left-deep plans. This results in three plans $((R \bowtie S_1) \bowtie T) \bowtie U$, $((S_2 \bowtie T_2) \bowtie U) \bowtie R$, and $((T_1 \bowtie U) \bowtie S_2) \bowtie R$. This partitioning is only possible if the order of arrival of the input relations can be fully controlled. Unfortunately, this assumption is impractical, both in a streaming setting, and in a more traditional setting. If we assume that the relations arrive at equal rates, the accumulated state at the joins forces query mesh to follow the sub-optimal plan $((R \bowtie S) \bowtie T) \bowtie U$ for a large subset of subsequent input tuples, regardless the partition in which they belong (see [9], Section 6.2). **(iv)** Finally, our approach (called HPE—Horizontal Partitioning with Eddies) allows bushy trees and can partition multiple relations. It results in the four plans shown in the lower-right part of Figure 1(a).

Figure 1(b) shows the number of intermediate tuples generated by the various methods when the selectivity ratio, s_2/s_1, varies from 1 to 1000. The large selectivity, s_2 is fixed at 0.01. All numbers are normalized by the number of intermediate tuples generated by our approach (HPE). Selectivity-based partitioning can achieve only a modest benefit compared to the best monolithic plan, because it can only partition one relation and is forced to use left-deep trees. For query mesh, we present numbers for the best (bc) and the worst case (wc). In the best case, it is assumed that the order of arrival of the input tuples can be controlled so that the chosen partitioning can be enforced. This yields the same number of intermediate tuples as our approach. In the worst case, query mesh is forced by the arrival order to follow one sub-optimal plan for a large subset of input tuples. Note that Figure 1(b) shows a lower bound of the intermediate tuples generated by query mesh in this case. Our approach yields the lowest number of intermediate tuples because it does not suffer from the limitations described above.

2.2 Partitioning with eddies

Eddies with symmetric hash joins [1, 8] provide a framework that naturally encapsulates horizontal partitioning and state sharing, making it an ideal framework for exploiting data correlations through horizontal partitioning. With eddies, fixed query plans are no longer constructed. Instead, the operators that are involved in the query are connected with a central router (the eddy), and query execution proceeds by routing the tuples through the operators. The eddy makes a routing decision for each individual tuple. This enables multiple plans to be executed simultaneously for the same query, each plan operating on a different subset of (base or intermediate) tuples. These multiple plans are not created explicitly; rather, they are implied by the *eddy routing policy*. Note that although eddies were introduced as a way to achieve adaptivity in a streaming environment, we do not use them as such. We assume a more traditional setting, where the data is static. This eliminates the adaptivity overhead of eddies.

Consider the join query $R \bowtie S \bowtie T \bowtie U$ and the execution of the query using an eddy, as shown in Figure 2. Tuples from relations R and U each have only one possible destination: $R \bowtie S$ and $T \bowtie U$, respectively. However, S tuples can be routed to either $R \bowtie S$ or $S \bowtie T$, and T tuples can be routed to either $T \bowtie U$ or $S \bowtie T$. The eddy can use a predicate on one of the relation attributes to distinguish the routing destinations. In Figure 2, the eddy uses the predicate ϕ_S (e.g., $\phi_S = (S.\mathsf{Y} > 5)$) to route S tuples. Tuples from S that satisfy ϕ_S are routed to $R \bowtie S$, yielding partition S_1. Tuples from S that do not satisfy ϕ_S are routed to $S \bowtie T$, yielding partition S_2.

In Figure 2 the intermediate results, as stored in the hash tables of the symmetric hash joins, are shown. While all R (U) tuples are stored in the join $R \bowtie S$ ($T \bowtie U$), the relations S and T are partitioned. The S_1 (T_1) partition is stored in $R \bowtie S$ ($T \bowtie U$), and the S_2 and T_2 are stored in $S \bowtie T$. Thus, the intermediate results created are RS_1, S_2T_2, and T_1U. The RS_1 and T_1U tuples are stored in $S \bowtie T$ (their only routing destination). The state of $S \bowtie T$ is then as shown in Figure 2. The subsequent routing of intermediate results in the combined execution of the four plans shown in Figure 1(a). The state captured in the joins at the end of query execution is shown in Figure 2. Note that the relations RS_1 and T_1U that are common in multiple plans are computed only once. Eddies provide maximal sharing of intermediate results at execution time, with no extra optimization time overhead.

3. CONDITIONAL JOIN PLANS

Traditional query optimization is realized as search for the "best" join plan in a suitable search space. The search space can be constrained (e.g., to exclude bushy trees), and the search algorithm can be either exhaustive or greedy, among other possibilities. To achieve a similar search framework for our optimization problem, we need a new representation of the search space that is capable of capturing both the partitioning and the join orders for each partition combination. Conditional join plans offer such a representation. We begin with defining CJPs, and then discuss how to estimate their cost.

3.1 Definition of CJPs

Before defining CJPs formally, we provide an example CJP using the running example. The *query graph* of a query has the relations as its nodes and the eligible joins as its edges. We annotate each edge with the predicate of the corresponding join. For our example, the query graph is $\mathcal{Q}(\{R, S, T, U\}, \mathcal{J})$, where the set of edges is $\mathcal{J} = \{\bowtie_{RS}, \bowtie_{ST}, \bowtie_{TU}\}$. The Cartesian product $\mathcal{U} = R \times S \times T \times U$ in our example has the following schema:

$$\mathcal{U}(R.A, R.\mathsf{X}, S.A, S.\mathsf{Y}, S.B, T.B, T.\mathsf{Z}, T.C, U.C, U.\mathsf{W}).$$

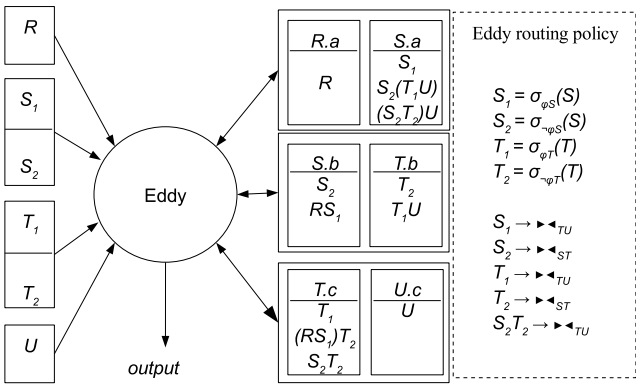


Figure 2: Query execution using an eddy with symmetric hash joins. The routing policy directs tuples from partition S_1 to $R \bowtie S$, tuples from S_2 to $S \bowtie T$, tuples from T_1 to $T \bowtie U$, and tuples from T_2 to $S \bowtie T$. The result is four different plans that execute simultaneously sharing all common state.

Let us *conceptually* view the query $\mathcal{Q}$ as a selection query over the Cartesian product $\mathcal{U}$. Then a join between two relations is a predicate defined over $\mathcal{U}$. For example, the join predicate $\bowtie_{RS}$ is the predicate $\bowtie_{RS} = (R.A = S.A)$ defined over the relation $\mathcal{U}$. Besides join predicates, we also define decision predicates. An example decision predicate is $\phi_S = (S.\mathsf{Y} > 5)$, where we subscript the name of the predicate with the name of the relation that contains the attribute. A CJP is a directed, rooted tree that contains join and decision predicates as its nodes. Decision nodes model relation splits, and the orders of the join nodes at each path from the root to a leaf model the join plans for each combination of partitions. One possible CJP for the running example is depicted in Figure 3. The CJP can be interpreted as a conditional selection plan [7] on the Cartesian product $\mathcal{U}$. Tuples from $\mathcal{U}$ flow from the root to the leaves of the CJP. A tuple $rstu$ first visits the node ϕ_S. If $\phi_S(rstu) = \mathbf{T}$, it follows the upper outgoing edge of the node, otherwise it follows the lower edge. Let $S_1 = \sigma_{\phi_S=\mathbf{T}}(S)$ and $S_2 = \sigma_{\phi_S=\mathbf{F}}(S)$. Then tuples from $R \times S_1 \times T \times U$ follow the upper edge of ϕ_S, and tuples from $R \times S_2 \times T \times U$ follow the lower edge of ϕ_S. After all the decision nodes have been visited, the four resulting partitions of $\mathcal{U}$ are $\mathcal{U}_1 = R \times S_1 \times T_1 \times U$, $\mathcal{U}_2 = R \times S_1 \times T_2 \times U$, $\mathcal{U}_3 = R \times S_2 \times T_1 \times U$, and $\mathcal{U}_4 = R \times S_2 \times T_2 \times U$. Tuples from different partitions follow different orders of the join nodes. For example, tuples from $\mathcal{U}_1$ follow the order $\bowtie_{RS}, \bowtie_{TU}, \bowtie_{ST}$ (subplan P_{11}). When a tuple of $\mathcal{U}$ visits a join node, it either satisfies the predicate and continues to the next node, or it does not satisfy

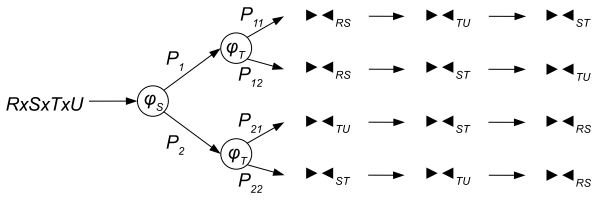


Figure 3: A CJP for the query $R \bowtie S \bowtie T \bowtie U$ with two decision predicates ϕ_S, ϕ_T. The two ϕ_T predicates in the subplans P_1 and P_2 can have different values ϕ_{T_1} and ϕ_{T_2}.

the predicate and is discarded. Consider a tuple of the partition $\mathcal{U}_1$, $rs_1\,t_1\,u$ that visits the join node $\bowtie_{RS}$ in the sub-plan P_{11}. The tuple is evaluated against the predicate $\bowtie_{RS}= (R.A = S.A)$, and it will continue to the next node, $\bowtie_{TU}$ only if $\bowtie_{RS}\ (rs_1\,t_1\,u) = \mathbf{T}$. Observe that the tuples that pass the predicate $\bowtie_{RS}$ are the tuples of the relation $R \bowtie S_1 \times T_1 \times U$, so the join $R \bowtie S$ is executed first. The join predicate order followed by $\mathcal{U}_1$ tuples is $(R \bowtie S_1) \bowtie (T_1 \bowtie U)$. In fact, the CJP models the partitioning and the forest of join trees corresponding to HPE as shown in Figure 1(a).

More formally, given a query graph $\mathcal{Q}(\{R_1, \ldots, R_n\}, \mathcal{J})$ and a set $\mathcal{F}$ of decision predicate values, a conditional join plan $P(\mathcal{Q})$ is a directed, rooted tree that contains two kinds of nodes. A *decision node* $\rightarrow \phi_X \begin{matrix}\nearrow\\ \rightarrow \\ \searrow\end{matrix}$ contains an n-ary predicate $\phi_X \in \mathcal{F}$, defined over the Cartesian product $\mathcal{U} = R_1 \times \cdots \times R_n$. A decision predicate splits the relation $\mathcal{U}$ into n disjoint partitions that cover the relation. A *join node* $\rightarrow\bowtie_{XY}\rightarrow$ contains a predicate from $\mathcal{J}$, defined over $\mathcal{U}$, with only one outgoing edge. A join predicate discards the tuples of the Cartesian product that do not satisfy it. A CJP $P(\mathcal{Q})$ is valid for the query $\mathcal{Q}$ if every path from the root to a leaf contains every join predicate in $\mathcal{J}$ exactly once. This means that the correct query result is produced. To simplify our discussion, we also require that a join node does not precede a decision node in any path from the root to a leaf.

A CJP $P(\mathcal{Q})$ can easily be converted to a forest of join plans, each join plan operating on a certain combination of data partitions. Consider the tree formed by the decision predicates of $\mathcal{F}$ (e.g., the full binary tree formed by ϕ_S and ϕ_T in Figure 3). Each leaf of this tree defines a particular combination of relation partitions that can be discovered with a tree traversal, and leads to a particular sub-plan (P_{11}–P_{22} in the figure) that contains only join nodes. For example, the sub-plan P_{11} uses the relation partitions R, S_1, T_1, and U. After discovering the partitions, each order of join predicates in a sub-plan can be de-linearized to form the corresponding join plan. This transformation defines the "semantics" of a CJP, i.e., the way it is executed by a query processor.

Finally, we define a *CJP structure* $P(\mathcal{Q}, \mathcal{F})$ as a CJP whose decision predicates are not assigned values, but are viewed as variables of the CJP. A CJP can be derived from a CJP structure when we assign values to all the predicates in $\mathcal{F}$. These values can be normal predicate values or one of the following two special values (for chain query graphs): the always-true predicate ϕ_{true} and the always-false predicate ϕ_{false}. If the predicate of a decision node $\phi = \phi_{true}$, all $\mathcal{U}$ tuples follow the upper sub-plan of the node, and if $\phi = \phi_{false}$, all $\mathcal{U}$ tuples follow the lower sub-plan. For general query graphs, we can similarly define special predicates that direct the incoming tuples to exactly one particular outgoing edge. We denote by $\mathcal{F}(P)$ the assigned values of the decision predicates of the concrete CJP P.

3.2 Cost estimation basics

In order to define the cost of a conditional join plan, we need to make several decisions. First, to be able to formally analyze the query optimization problem, and to keep the cost formulas tractable, we use the number of intermediate tuples as the cost metric. Although simple, this metric is known to be quite effective, and it mirrors disk or CPU-based cost functions in many scenarios [4].

Second, we need to incorporate the cost of partitioning into the cost metric. To simplify our cost formulas, we ignore the partitioning cost for now. Instead, we impose a constraint on the number of predicates that can be used, termed the *partitioning budget*. Taking into account the cost of partitioning is an easy extension to our cost model.

Third, we need to decide whether the cost of a CJP includes the size of an intermediate result common to multiple plans once or multiple times. This in turn depends on the query processor that will execute the CJP. As discussed previously, a CJP $P(\mathcal{Q})$ can be transformed into a forest of join plans. Let us denote by $\|P\|_{NS}$ the total number of intermediate tuples generated by these plans and by $\|P\|_S$ the number of intermediate tuples when duplicate tuples are counted only once.[1] Then $\|P\|_{NS}$ is the cost of the CJP P when intermediate results are not shared during query execution, and $\|P\|_S$ is the cost of P when maximal sharing of intermediate results occurs.

It is easy to compute $\|P\|_{NS}$ in a naive way. The CJP is transformed to a forest of join plans, and the cardinalities of the intermediate results they produce are computed as usual. In order to compute $\|P\|_S$ in a similar way, we need to note the intermediate results that are added to the total cost, and only count them once. For example, consider the cost estimation of the CJP in Figure 3. A traversal of the binary tree of decision predicates finds the predicate assignments[2] that hold in each leaf of the tree (the sub-plans P_{11}–P_{22}). These assignments define the relation partitions. For example, in P_{11}, the assignment of the decision predicates is $\Phi = (\phi_S = \mathbf{T}, \phi_T = \mathbf{T})$, which defines the partitions $S_1 = \sigma_{\phi_S}(S)$ and $T_1 = \sigma_{\phi_T}(T)$. The corresponding join plan is $(R \bowtie S_1) \bowtie (T_1 \bowtie U)$, whose cost is $|RS_1| + |T_1U|$. Using our notation for join and decision predicates, the cost $|RS_1|$ can be written as $\Pr(\bowtie_{RS}, \phi_S)|R||S|$, where

$$\Pr(\bowtie_{RS}, \phi_S) = \frac{|\sigma_{\bowtie_{RS} \wedge \phi_S}(R \times S \times T \times U)|}{|R||S||T||U|} = \frac{|RS_1|}{|RS|}. \quad (1)$$

Apart from being costly, this cost estimation procedure defeats the purpose of constructing CJPs in the first place. Since the CJP needs to be translated to a forest of join plans when its cost needs to be computed, search could as well proceed in the space of join plan forests. Instead, we propose a cost function that is recursive in the structure of CJPs. Before covering recursive cost estimation in Section 3.4, we show how the required probabilities as the one in Equation 1 can be estimated.

3.3 Estimation of joint selectivities

To estimate joint selectivities, we need a statistical model of the database that captures correlations. Existing techniques [6, 13] enable trade-offs between the storage requirements and the correlations that are captured, typically using the notion of graphical models [14]. Unfortunately, these techniques cannot be used unmodified in our setting, as they do not support arbitrary joins. One proposal [6] is designed with only selection queries in mind, and another [13] supports only key-foreign key joins. However, it is not hard to extend the notion of statistical relational models described in [12] to work correctly with arbitrary joins. The downside is that all possible joins must be known prior to building the statistical model.

We use an undirected graphical model (also called a Markov network) to estimate joint selectivities. A Markov network is defined by its structure and the probability distributions that need to be kept.

[1] The subscript *NS* stands for "no sharing," and the subscript *S* stands for "sharing."

[2] We distinguish between a predicate value and a predicate assignment. The former is a function $\mathcal{U} \rightarrow \{\mathbf{T}, \mathbf{F}\}$, while the latter is a value from $\{\mathbf{T}, \mathbf{F}\}$. Further, when there is no confusion, we abbreviate the assignment $\phi = \mathbf{T}$ as ϕ, and the assignment $\phi = \mathbf{F}$ as $\neg\phi$.

The structure is an undirected graph with random variables as its nodes. The edge set of the graph encodes the conditional independencies that the model implies. Although these can be discovered automatically using a training set, we have chosen a fixed structure that captures the necessary correlations in minimal space. After the structure is defined, a probability distribution for each clique of the graph has to be computed and stored.

The model construction algorithm takes as argument the *universal query graph*, the query graph that captures all possible joins in the database. The random variables that serve as the nodes of the Markov network are (1) the *descriptive attributes* of the database and (2) the *join indicators*. The descriptive attributes are the attributes that are used to partition the relations during query processing. The join indicators are binary random variables that capture the events that two relations join. A join indicator exists for each edge in the universal query graph. We choose to place an edge in the graph only between a join indicator and the descriptive attributes of the relations whose join it represents. The model is represented internally as a junction tree [14], which allows for efficient computation of joint probabilities.

Assume that our example query defines the universal query graph, and that the descriptive attributes are $R.\mathsf{X}$, $S.\mathsf{Y}$, $T.\mathsf{Z}$, $U.\mathsf{W}$. The join indicators are $\mathcal{J}_{RS}$, $\mathcal{J}_{ST}$, $\mathcal{J}_{TU}$, defined as $\mathcal{J}_{RS} = (R.A = S.A)$, $\mathcal{J}_{ST} = (S.B = T.B)$, and $\mathcal{J}_{TU} = (T.C = U.C)$. Figure 4 shows the Markov network.

Figure 4: Markov network for the example database.

The probability distributions that need to be stored are the maximal cliques of the Markov network. In our example, we need the distributions $P(\mathsf{X}, \mathcal{J}_{RS})$, $P(\mathsf{Y}, \mathcal{J}_{RS})$, $P(\mathsf{Y}, \mathcal{J}_{ST})$, etc. All of these distributions can be stored as one-dimensional distributions, and they can be computed without first constructing the Cartesian product. For example, the probability distribution $P(\mathsf{Y}, \mathcal{J}_{RS})$ can be maintained as two one-dimensional distributions: $P(\mathsf{Y}, \mathcal{J}_{RS} = \mathbf{T})$ and $P(\mathsf{Y}, \mathcal{J}_{RS} = \mathbf{F})$. We can compute $P(\mathsf{Y}, \mathcal{J}_{RS} = \mathbf{T})$ as $\frac{C(\mathsf{Y}, \mathcal{J}_{RS}=\mathbf{T})}{|R||S|}$, where $C(\mathsf{Y}, \mathcal{J}_{RS} = \mathbf{T})$ is the result of the query

```
select Y, count(*) from R,S
where R.A = S.A group by Y.
```

Then, we can compute $P(\mathsf{Y}, \mathcal{J}_{RS} = \mathbf{F})$ as

$$P(\mathsf{Y}, \mathcal{J}_{RS} = \mathbf{F}) = \frac{|R|C(\mathsf{Y}) - C(\mathsf{Y}, \mathcal{J}_{RS} = \mathbf{T})}{|R||S|},$$

where $C(\mathsf{Y})$ is the result of the query

```
select Y, count(*) from S group by Y.
```

Given the constructed Markov network in a form of a junction tree, we can efficiently compute arbitrary joint probabilities of decision and join predicates. For example, the probability of Equation 1 is the probability that both predicates $\bowtie_{RS}$, and ϕ_S are true. To compute it, we need to form the marginal distribution of $\mathcal{J}_{RS}$ and X which can be done with standard inference algorithms like message passing [6, 14].

3.4 Recursive cost estimation

We define a recursive cost function, $\textsc{Cost}_{NS}$, for $\|P\|_{NS}$. For $\|P\|_S$, we have a recursive cost function that is correct for a restricted space of CJPs, discussed in Section 4. $\textsc{Cost}_{NS}$ takes two sets as arguments, both initially empty: Φ, a set of predicate assignments that hold in the current node under evaluation, and $\mathcal{M}$, the set of the intermediate results that have already been produced. The set $\mathcal{M}$ contains elements of the form $m = (\mathcal{R}, \Xi)$, where $\mathcal{R}$ is a set of relations, and Ξ is a set of join predicates. The cost of a plan P is defined as the cost of its root node:

$$\textsc{Cost}_{NS}(P) = \textsc{Cost}_{NS}(\mathit{root}(P), \emptyset, \emptyset)$$

To define the cost of a node $\textsc{Cost}_{NS}(n, \Phi, \mathcal{M})$, we need to distinguish between decision and join nodes. The cost of a decision node $n =\!\!\rightarrow \phi \begin{smallmatrix} \nearrow & n' \\ \searrow & n'' \end{smallmatrix}$ is

$$\textsc{Cost}_{NS}(n, \Phi, \mathcal{M}) = \textsc{Cost}_{NS}(n', \Phi \cup \{\phi\}, \mathcal{M}) + \textsc{Cost}_{NS}(n'', \Phi \cup \{\neg\phi\}, \mathcal{M}). \quad (2)$$

Observe that a decision node does not add to the cost since we ignore the cost of partitioning. A decision node is used only to update the set Φ. When the cost function recurses to the upper (lower) node n' (n''), the assignment $\phi = \mathbf{T}$ ($\phi = \mathbf{F}$) has been added to the set Φ.

To compute the cost of a join node $n =\!\!\rightarrow\!\bowtie_{ij}\!\rightarrow n'$, we need to consider four cases:

1. The join node represents a join between the two base relations R_i and R_j.
2. The join node represents a join between an intermediate relation that contains R_i and the base relation R_j.
3. The join node represents a join between an intermediate relation that contains R_j and the base relation R_i.
4. The join node represents a join between two intermediate relation, one containing R_i and the other containing R_j.

The argument set $\mathcal{M}$ is used to make the distinction. Assume two elements of $\mathcal{M}$, $m_1 = (\mathcal{R}_1, \Xi_1)$ and $m_2 = (\mathcal{R}_2, \Xi_2)$, such that $R_i \in \mathcal{R}_1$ and $R_j \in \mathcal{R}_2$. If both these elements exist, we are in case 4 above; if only m_1 exists, we are in case 2; if only m_2 exists, we are in case 3; if none of m_1 and m_2 exist, we are in case 1. In case 1, we calculate the cost of a join node that represents the join between two base relations R_i,R_j:

$$\textsc{Cost}_{NS}(n, \Phi, \mathcal{M}) = \Pr(\bowtie_{ij}, \bigwedge_{\phi \in \Phi^{\downarrow\{R_i, R_j\}}} \phi)|R_i||R_j| + \textsc{Cost}_{NS}(n', \Phi, \mathcal{M} \cup \{(\{R_i, R_j\}, \{\bowtie_{ij}\})\}) \quad (3)$$

The node adds to the total cost the cardinality of the Cartesian product $|R_i \times R_j|$ weighted by the probability of the conjunction of $\bowtie_{ij}$, the join predicate under evaluation, and the decision predicate assignments in the set $\Psi = \Phi^{\downarrow\{R_i, R_j\}}$. The set Ψ contains the decision predicate assignments that have been seen so far, Φ, restricted to those that involve attributes of R_i and R_j. In general, if $\mathcal{X}$ is a set of relations, $\Phi^{\downarrow\mathcal{X}}$ denotes the restriction of Φ to $\mathcal{X}$, i.e., the predicate assignments in Φ that contain only attributes of relations in $\mathcal{X}$. The cost contribution of the node n is exactly the number of intermediate tuples of the join $R_i \bowtie R_j$, when R_i and R_j are partitioned by the decision predicate assignments in $\Phi^{\downarrow\{R_i, R_j\}}$.

For example, consider the cost of the node $\bowtie_{RS}$ in P_{12} of Figure 3. Since we have seen the decision nodes ϕ_S, ϕ_T and followed the edges that lead to P_{12}, the assignment of decision predicates currently valid, as set by the cost calculation of the decision nodes, is $\Phi = \{\phi_S = \mathbf{T}, \phi_T = \mathbf{F}\}$. Decision nodes do not add elements to the set $\mathcal{M}$, so $\mathcal{M} = \emptyset$ and we are in case 1. The set $\Phi^{\downarrow R,S}$ is $\Phi^{\downarrow R,S} = \{\phi_S, \neg\phi_T\}^{\downarrow R,S} = \{\phi_S\}$. The contribution to the total cost is then $\Pr(\bowtie_{RS}, \phi_S)|R||S|$. In addition to adding the size of an intermediate result to the total cost, the node n in Equation 3 adds to the set $\mathcal{M}$ the element $m = (\{R_i, R_j\}, \{\bowtie_{ij}\})$, which represents the intermediate relation $R_i \bowtie R_j$.

In case 2, we join R_j with an intermediate result $\mathcal{R}_1$ that contains R_i and the join predicates in Ξ_1:

$$\text{COST}_{NS}(n, \Phi, \mathcal{M}) =$$
$$\Pr(\bowtie_{ij}, \bigwedge_{\xi \in \Xi_1} \xi, \bigwedge_{\phi \in \Phi^{\downarrow \mathcal{R}_1 \cup \{R_j\}}} \phi) |R_j| \prod_{R \in \mathcal{R}_1} |R| + \quad (4)$$
$$\text{COST}_{NS}(n', \Phi, \mathcal{M} - \{m_1\} \cup \{(\mathcal{R}_1 \cup \{R_j\}, \Xi_1 \cup \{\bowtie_{ij}\})\})$$

The cost contribution of the node n is now the size of the intermediate result $\mathcal{R}_1 \bowtie R_j$, when the relations are partitioned by the predicates in $\Phi^{\downarrow \mathcal{R}_1 \cup \{R_j\}}$.

Consider the calculation of the cost of $\bowtie_{ST}$ in P_{12}. The arguments of the cost function are $\Phi = \{\phi_S, \neg\phi_T\}$ and $\mathcal{M} = \{(\{R, S\}, \{\bowtie_{RS}\})\}$. Since there is an element in $\mathcal{M}$ that contains relation S, but there is no element that contains relation T, we are in case 2. The cost contribution of $\bowtie_{ST}$ is $\Pr(\bowtie_{ST}, \bowtie_{RS}, \phi_S, \neg\phi_T)|T||R||S| = |RS_1T_2|$. In addition, an element for the intermediate relation RST will be added to $\mathcal{M}$. Case 3 is symmetric to case 2 and can be dealt with similarly. Finally, in case 4 we join the intermediate relations $\mathcal{R}_1$ and $\mathcal{R}_2$:

$$\text{COST}_{NS}(n, \Phi, \mathcal{M}) =$$
$$\Pr(\bowtie_{ij}, \bigwedge_{\xi \in \Xi_1 \cup \Xi_2} \xi, \bigwedge_{\phi \in \Phi^{\downarrow \mathcal{R}_1 \cup \mathcal{R}_2}} \phi) \prod_{R \in \mathcal{R}_1 \cup \mathcal{R}_2} |R| + \quad (5)$$
$$\text{COST}_{NS}(n', \Phi, \mathcal{M} - \{m_1, m_2\} \cup \{(\mathcal{R}_1 \cup \mathcal{R}_2, \Xi_1 \cup \Xi_2 \cup \{\bowtie_{ij}\})\})$$

For example, consider the cost computation for the node $\bowtie_{ST}$ in P_{11}. This is the sub-plan that represents the bushy join tree. The cost function has already evaluated the nodes $\bowtie_{RS}$ and $\bowtie_{TU}$. When it reaches $\bowtie_{ST}$ it is called with arguments $\Phi = \{\phi_S, \phi_T\}$ and $\mathcal{M} = \{(\{R, S\}, \{\bowtie_{RS}\}), (\{T, U\}, \{\bowtie_{TU}\})\}$. The cost contribution of $\bowtie_{ST}$ is $\Pr(\bowtie_{ST}, \bowtie_{RS}, \bowtie_{TU}, \phi_S, \phi_T)|R||S||T||U| = |RS_1T_1U|$. The recursion of the cost function ends at the leaves of the CJP, or at the second-last join node if we do not want to include the size of the query result in the cost estimate. By the definition of the cost function COST_{NS}, the following lemma holds.

Lemma 1. *For any valid $P(\mathcal{Q})$, $\text{COST}_{NS}(P) = \|P\|_{NS}$.*

4. THE EDDY CJP SPACE

4.1 Eddy restrictions

The routing nature of query execution with eddies imposes constraints on the possible partitions as well as on the join plans that can be executed. This in turn imposes restrictions on the CJPs that can be considered during query optimization. Consider for example the valid CJP for our example query in Figure 6.

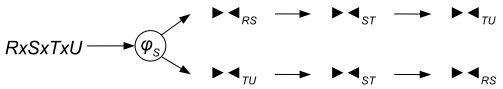


Figure 6: A CJP that is not eddy-compliant.

This CJP is equivalent to the join plans $((R \bowtie S_1) \bowtie T) \bowtie U$, $((T \bowtie U) \bowtie S_2) \bowtie R$. If we were to execute this query with an eddy, we need to make a routing decision for T tuples using a predicate, ϕ_S, on relation S. If $\phi_S = \mathbf{T}$, T needs to be joined with $R \bowtie S_1$, while if $\phi_S = \mathbf{F}$, T needs to be joined with U. There is no possible routing that can achieve this. The routing decisions for T tuples can only be made using a predicate on T, ϕ_T. The restrictions on the possible CJPs is the price paid for state sharing provided by eddies.

The constraints imposed by eddies affect the CJP search space as follows. Given a query $\mathcal{Q}$, we can construct a *unique* CJP structure $P_e(\mathcal{Q}, \mathcal{F}_e)$, called the *eddy CJP structure*. Any CJP valid for $\mathcal{Q}$ that can be executed using an eddy (called an *eddy-compliant* CJP) can be derived from the eddy CJP structure by assigning values to the predicates in $\mathcal{F}_e$. Hence, the eddy CJP structure determines the *eddy CJP space* for this query. Appendix A details an algorithm that, given a query, constructs the unique eddy CJP structure. Figure 7 shows the eddy CJP structure for our example.

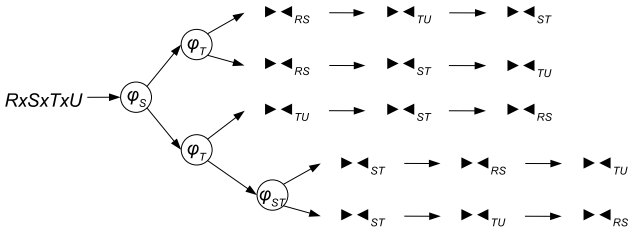


Figure 7: The eddy CJP structure for our running example. Note that the predicate ϕ_T must have a unique value.

For example, assume that we are given a partitioning budget of $c = 1$ and we decide to use the predicate $\phi_S = (S.\text{Y} > 5)$. Then, all the eddy-compliant CJPs can be derived from the eddy CJP structure of Figure 7 by assigning values to ϕ_T and ϕ_{ST} from the set $\{\phi_{true}, \phi_{false}\}$, as defined in Section 3.1. These values must be honored across sub-plans. For example, assume that we choose $\phi_T = \phi_{true}$. Then we must use the join order $(R \bowtie S_1) \bowtie (T \bowtie U)$ for partition S_1 and the join order $((T \bowtie U) \bowtie S_2) \bowtie R$ for partition S_2. Note that ϕ_{ST} is not defined in the $\phi_T = \phi_{true}$ sub-plans because the intermediate result ST is never formed in these sub-plans.

Since eddies provide maximal sharing, the recursive cost function COST_{NS} from Section 3.4 does not estimate the cost of an eddy-compliant CJP correctly. Fortunately, we can define a recursive cost function COST_{Eddy} that estimates the cost of an eddy-compliant CJP with sharing accounted for. Only one change is needed to COST_{NS}: Instead of including the decision predicates of the set $\Phi^{\downarrow \mathcal{X}}$, where $\mathcal{X}$ is the set of relations relevant to the join node under consideration, in Equations 3- 5, we simply include all the decision predicates in Φ. The following holds.

Lemma 2. *If P is eddy-compliant, $\text{COST}_{Eddy}(P) = \|P\|_S$.*

PROOF. See Appendix B.

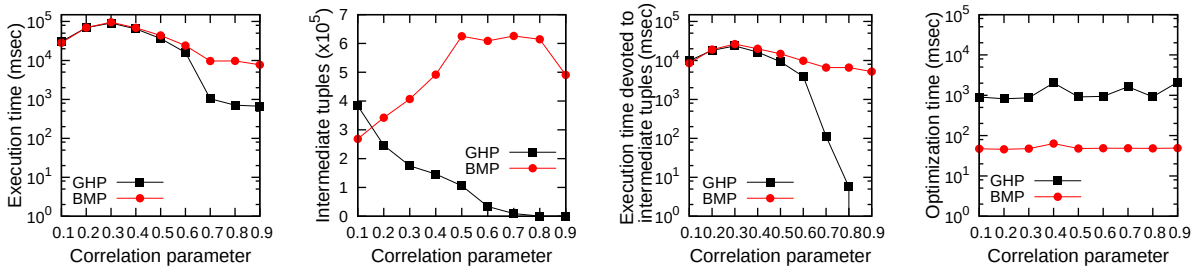
Denote the eddy CJP structure for a query Q by $P_e(\mathcal{Q}, \mathcal{F}_e)$. We can now formally state the problem we are solving.

Horizontal paritioning with eddies. *Given a query $\mathcal{Q}$ and a partitioning budget c, find the plan*

$$P^*(\mathcal{Q}) = \underset{|\mathcal{F}(P)| \leq c, \mathcal{F}(P) \subset \mathcal{F}_e}{\arg\min} [\text{COST}_{Eddy}(P(\mathcal{Q}))]$$

that is valid for $\mathcal{Q}$ and is eddy-compliant.

Put differently, query optimization has to partition the predicate variables $\mathcal{F}_e$ of the eddy CJP structure into two disjoint sets: The first set of size at most c contains predicates that are assigned normal predicate values (e.g., $S.\text{Y} > 5$), and the second set, of size at least $|\mathcal{F}_e| - c$ contains predicates that are assigned values from the set $\{\phi_{true}, \phi_{false}\}$. The choice of the two sets and the choice of values should yield the minimum cost. Once the predicates in $\mathcal{F}_e$ have been assigned values, it is trivial to construct an eddy routing policy that executes the resulting concrete CJP.



(a) Total execution time. (b) Intermediate tuples generated. (c) Execution time for intermediate tuples. (d) Optimization time.

Figure 5: The effect of varying the correlation parameter r in a 3-join query when all the joins have the same selectivity.

4.2 Greedy search

While possible, it is computationally infeasible to exhaustively search the eddy space. We propose an algorithm that starts from the best monolithic plan for a query and gradually builds an eddy-compliant CJP. At each step, the algorithm cycles over all the decision predicates on attributes that have not been used yet, and picks the one that yields the best cost when used to split the plan into two sub-plans. This is done greedily: when the algorithm introduces a split, it assumes that no future splits will occur, but rather that the best monolithic plans (under the eddy constraints) will be used for the sub-plans. The algorithm stops if it has introduced the maximum number c of decision predicates allowed, or if no further cost improvement can be achieved.

The gradual construction of the CJP has three advantages. First, the complete eddy CJP structure does not need to be stored. Second, the sizes of the CJPs whose costs will be evaluated are controllable; a CJP with more than c decision predicates is never generated. Finally, the cost of the final CJP is guaranteed to be less or equal to the cost of the best monolithic plan. However, since there is no backtracking, the algorithm can obviously get stuck in local minima; an initial choice for a locally optimal decision predicate can lead the algorithm to assume that no cost improvement can be made by further splitting. Appendix C provides the details and pseudocode for the greedy search algorithm, as well a discussion of its cost as compared to the cost of exhaustive search.

5. EXPERIMENTAL RESULTS

We have implemented eddies, symmetric hash joins and the greedy horizontal partitioning scheme in PostgreSQL, reusing the eddies code from the TelegraphCQ project [3] (see Appendix D for additional details). We compare our greedy horizontal partitioning algorithm (termed GHP) with the best monolithic plan (BMP), found by an exhaustive enumeration of all possible plans. Both methods use the same junction tree-based selectivity estimation code. To ensure fairness of comparison, the resulting plans of both methods are translated into an eddy routing policy and executed with eddies and SHJs. Note that executing a monolithic plan using eddies takes at most double the time than executing it using the vanilla PostgreSQL executor in our experiments.

The quality of the best monolithic plan depends only on the selectivities of the participating joins, while the quality of a CJP depends on both the average selectivities and the correlations. We study the effect of these parameters in Section 5.1. The optimization time is affected by the size of the statistical model, which is studied in Section 5.2. Finally, we study how our greedy algorithm scales in Section 5.3. We use synthetic data in order to be able to control three parameters: the number of tuples, the selectivity of the joins, and the data correlation. Appendix E covers the data generation in detail.

5.1 The effect of data correlation

We start with varying the data correlation parameter r in a 4-relation join query. Each relation has 10^4 tuples, and all the descriptive attributes (one per relation) take values from the domain $\{0, 1, \ldots, 9\}$. The data is generated using a correlation coefficient r which takes values in $[0.1, 0.9]$. The selectivities of all joins are fixed at 0.001. This means that there are no substantial optimization opportunities for this query in the traditional sense. However, when the data is naturally partitioned into subsets due to high correlation, the execution time of the query can be reduced using horizontal partitioning. This means that horizontal partitioning can be beneficial in situations where traditional optimization falls short.

Figure 5(a) shows the total execution time in msecs (in logarithmic scale) of the best monolithic plan of and the CJP found by the greedy horizontal partitioning algorithm. In non-correlated data ($r \in [0.1, 0.4]$), the two plans yield the same execution time. As the data becomes more correlated ($r \geq 0.5$), the CJP becomes faster than the monolithic plan. At $r = 0.9$, we observe a 90% reduction of the total execution time.

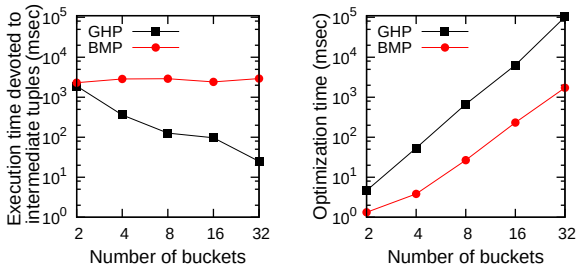
To understand the performance difference better, Figure 5(b) examines the number of intermediate tuples generated. While high correlations cause the best monolithic plan to produce more intermediate tuples, the opposite is true for horizontal partitioning. The number of intermediate tuples is zero when $r = 0.9$. Figure 5(c) shows the portion of the execution time devoted to intermediate tuples only. The benefits of horizontal partitioning are apparent: the execution time quickly drops to zero after r reaches 0.4, whereas the execution time of the best monolithic plan stays fairly constant.

The execution time savings of the query come at the cost of an increased optimization time. The greedy horizontal partitioning algorithm finished after 2 iterations in all cases. Even so, there is an order of magnitude increase compared to the optimization time of exhaustive monolithic optimization. However, the benefits of partitioning during query processing outweigh the increase in optimization time. The optimization time can be reduced if we reduce the buckets of the descriptive attribute histograms (currently 10), but with an expected degradation in accuracy, and thus a possible increase of query execution time.

We also experimented with a query with different join selectivities, set to $10^{-2}, 10^{-3}$, and 10^{-4}. As expected, the total execution cost is lower than in the previous case for both methods. Traditional query optimization can produce less intermediate tuples than before, but the effect of the data correlation remains the same. We omit the graphs due to lack of space.

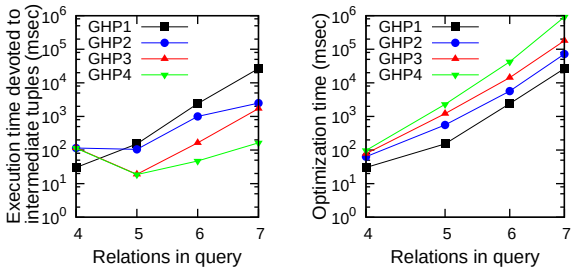
5.2 The effect of the number of buckets

We vary the domain size of the descriptive attributes for the same



(a) Execution time for intermediate tuples. (b) Optimization time.

Figure 8: The effect of the number of histogram buckets.



(a) Execution time for intermediate tuples. (b) Optimization time.

Figure 9: Varying the number of relations and GHP iterations.

query, fixing the correlation parameter at 0.6. Since we have one bucket per domain value, this is equivalent to varying the number of buckets in the constructed histograms. A large domain size means more detailed statistics and thus more opportunities for horizontal partitioning, but it makes the search in the junction tree more expensive. Figure 8(a) shows the execution time for intermediate tuples for GHP and BMP when the number of buckets varies from 2 to 32 (both axes in logarithmic scale). The execution time for intermediate tuples of GHP is equal to that of the best monolithic plan when there is only 2 buckets, and it improves over the best monolithic plan by two orders of magnitude when there 32 buckets. Figure 8(b) shows the optimization time. Both methods are affected equally by the increase in the number of buckets. The optimization time of GHP is one order of magnitude worse than that of traditional query optimization.

Note that with more than 16 buckets, the optimization time exceeds the execution time for intermediate tuples in our setting. However, if the size of the database was larger (e.g., each relation contains 10^8 tuples), the execution time would vary from 10^4 to 10^8 in Figure 8(a), while the optimization time would not be affected. In most settings, achieving up to two orders of magnitude faster execution is more significant than an one order of magnitude slower optimization. However, a wrong choice of parameters can lead to high optimization times for horizontal partitioning.

5.3 Scaling the number of relations

Finally, we study the performance of GHP when the number of relations in the query varies from 4 to 7. The correlation parameter is fixed at 0.5, the selectivity of all the joins is fixed at 0.001, and the domain size of the descriptive attributes is fixed at 4. At the same time, we vary the number of iterations of the GHP algorithm from 1 to 4. The execution time devoted to intermediate tuples is shown in Figure 9(a), and the optimization time is shown in Figure 9(b). As the number of joins in the query grows, it is very beneficial to increase the number of iterations of the greedy algorithm. In particular, for seven relations, four iterations of the greedy algorithm can reduce the execution time of intermediate tuples by two orders of magnitude when compared to one iteration.

6. CONCLUSIONS AND FUTURE WORK

Data correlations provide opportunities for more effective query optimization by partitioning relations. We first present a principled way to approach the problem of horizontal partitioning as search in the space of conditional join plans. CJPs provide an intuitive way to think about the problem, and recursive cost formulas for CJPs can be defined. Further, we show how to efficiently estimate correlated selectivities using a statistical model with low storage overhead. Then, we show how the sharing of intermediate results that is inherent in eddies restricts the space of possible CJPs. A greedy search with controlled iterations in this space is proposed that can achieve an one order of magnitude better execution time for highly correlated databases, while performing on par with the best monolithic plan at low correlations.

This work opens several lines of research that we plan to pursue. First, a problem that remains open is whether shared computation is always beneficial. Second, it would be interesting to explore multi-query optimization in this environment, where multiple queries are optimized together to produce many join plans that share computation. Finally, we would like to explore the parallel query processing case where the optimization metric is throughput..

7. REFERENCES

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pp. 261–272, 2000.

[2] P. Bizarro, S. Babu, D. J. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In *VLDB*, 2005.

[3] S. Chandrasekaran, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[4] S. Cluet, and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products In *ICDT*, 1995.

[5] A. Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 33(1):44–49, 2004.

[6] A. Deshpande, M. N. Garofalakis, and R. Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. In *SIGMOD*, pp. 199–210, 2001.

[7] A. Deshpande, C. Guestrin, W. Hong, and S. Madden. Exploiting correlated attributes in acquisitional query processing. In *ICDE*, pp. 143–154, 2005.

[8] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pp. 948–959, 2004.

[9] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[10] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6):85–98, 1992.

[11] P. L. Fackler. Generating correlated multidimensional variates. Available at http://www4.ncsu.edu/~pfackler/randcorr.ps.

[12] L. Getoor. *Learning Statistical Models from Relational Data*. PhD thesis, Stanford University, 2001.

[13] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *SIGMOD*, pp. 461–472, 2001.

[14] D. Koller, and N. Friedman. Probabilistic graphical models. MIT Press, 2009.

[15] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, pp. 128–137, 1986.

[16] R. V. Nehme, E. A. Rundensteiner, and E. Bertino. Self-tuning query mesh for adaptive multi-route query processing. In *EDBT*, 2009.

[17] R. V. Nehme, K. Works, E. A. Rundensteiner, and E. Bertino. Query mesh: Multi-route query processing technology. *PVLDB*, 2(2), 2009.

[18] N. Polyzotis. Selectivity-based partitioning: a divide-and-union paradigm for effective query optimization. In *CIKM*, 2005.

[19] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, pp. 353–, 2003.

APPENDIX

A. THE EDDY CJP STRUCTURE

We show how, given a query $\mathcal{Q}$, we can construct a unique eddy CJP structure $P_e(\mathcal{Q}, \mathcal{F}_e)$. We assume the chain query graph:

$$\mathcal{Q}(\{R_1, \ldots, R_n\}, \{\bowtie_{i,i+1} | i = 1, \ldots, n-1\})$$

The generalization to tree query graphs is straightforward if we consider n-ary instead of binary decision predicates. The construction proceeds in two steps. First, the *eddy skeleton routing policy* $\pi_e(\mathcal{Q}, \mathcal{F}_e)$ of a query is constructed. Then, using π_e, the eddy CJP structure $P_e(\mathcal{Q}, \mathcal{F}_e)$ is constructed.

An eddy routing policy π is a map from relation schemas to join operators. The eddy skeleton routing policy is the most general routing policy possible, and it is unique for a given query $\mathcal{Q}$. For its construction, we start with $\pi_e(\mathcal{Q}, \mathcal{F}_e) = \emptyset$. For each base relation $R_i, i = 2, \ldots, n-1$ in the query, we add to $\pi_e(\mathcal{Q}, \mathcal{F}_e)$ the element

$$R_i \mapsto \phi_i \begin{array}{l} \nearrow \ \bowtie_{i-1,i} \\ \searrow \ \bowtie_{i,i+1} \end{array} .$$

For each intermediate relation $R_i R_{i+1} \cdots R_{i+k}$, we add to $\pi_e(\mathcal{Q}, \mathcal{F}_e)$ the element

$$R_i R_{i+1} \cdots R_{i+k} \mapsto \phi_{i,\ldots,i+k} \begin{array}{l} \nearrow \ \bowtie_{i-1,i} \\ \searrow \ \bowtie_{i+k,i+k+1} \end{array} ,$$

until no intermediate relation is left. Note that the assumption we follow in this section is that if a tuple satisfies a decision predicate, it will be routed to the leftmost possible join in the chain query graph. The eddy skeleton routing policy for our running example is

$$\pi_e(\mathcal{Q}, \mathcal{F}_e) = \{S \mapsto \phi_S \begin{array}{l} \nearrow \ \bowtie_{RS} \\ \searrow \ \bowtie_{ST} \end{array} , T \mapsto \phi_T \begin{array}{l} \nearrow \ \bowtie_{ST} \\ \searrow \ \bowtie_{TU} \end{array} ,$$
$$ST \mapsto \phi_{ST} \begin{array}{l} \nearrow \ \bowtie_{RS} \\ \searrow \ \bowtie_{TU} \end{array} \}.$$

Given the eddy skeleton routing policy $\pi_e(\mathcal{Q}, \mathcal{F}_e)$, we construct the eddy CJP structure $P_e(\mathcal{Q}, \mathcal{F}_e)$. The algorithm first produces the full binary tree of base predicates ϕ_i. This will result in a binary tree with 2^{n-2} leaves. Each leaf of this tree corresponds to a full $\{\mathbf{T}, \mathbf{F}\}$-assignment of all the base predicates $\phi_2, \ldots, \phi_{n-1}$, and leads to a sub-plan containing join nodes and possibly decision nodes that contain predicates on intermediate relations. Denote each of these sub-plans by $P(\phi_2, \ldots, \phi_{n-1})$. The CJP structure for our example query is shown again in Figure 10. It is rotated (rotations do not affect correctness or the results of cost calculation), in order to conform to the assumption we made above. The sub-plans are denoted by $P(\phi_S, \phi_T)$, and $P_1 = P(\mathbf{T}, \mathbf{T})$, $P_2 = P(\mathbf{T}, \mathbf{F})$, etc.

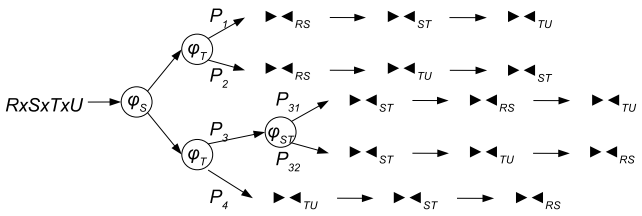


Figure 10: The eddy CJP structure for our running example.

The first step of the algorithm is to decide which decision nodes over intermediate relations will be placed in which sub-plans. For example, in Figure 10, the decision node ϕ_{ST} is placed only in the sub-plan P_3. Essentially, a decision node $\phi_{i,i+1}$ should only be placed in the sub-plans in which the intermediate result $R_i R_{i+1}$ is formed. These are the sub-plans in which both R_i and R_{i+1} are routed to $\bowtie_{i,i+1}$ and produce $R_i R_{i+1}$. Under our formalization, routing R_i to $\bowtie_{i,i+1}$ means that $\phi_i = \mathbf{F}$, and routing R_{i+1} to $\bowtie_{i,i+1}$ means that $\phi_{i+1} = \mathbf{T}$. So, the decision node $\phi_{i,i+1}$ is placed in the sub-plans $P(\phi_2, \ldots, \phi_i = \mathbf{F}, \phi_{i+1} = \mathbf{T}, \ldots, \phi_{n-1})$.

Having placed the level-2 decision nodes, the same procedure can place the level-3 nodes, and continue until all the decision nodes present in the eddy skeleton routing policy have been placed. The generalized rule is: The decision predicate $\phi_{i,\ldots,i+k}$ is included in a sub-plan P iff

1. $\phi_i = \mathbf{F}$ and $\phi_{i+1,\ldots,i+k}$ is included in P and $\phi_{i+1,\ldots,i+k} = \mathbf{T}$, or
2. $\phi_{i,\ldots,i+k-1}$ is included in P and $\phi_{i,\ldots,i+k-1} = \mathbf{F}$ and $\phi_{i+k} = \mathbf{T}$.

The final step is to place all the join nodes at every leaf of the formed tree of the decision nodes. A partial order of the join nodes is defined by the predicate assignments that hold at each leaf.

1. If $\phi_i = \mathbf{T}$, then $\bowtie_{i-1,i} \prec \bowtie_{i,i+1}$, else $\bowtie_{i,i+1} \prec \bowtie_{i-1,i}$.
2. If $\phi_{i,\ldots,i+k} = \mathbf{T}$, then $\bowtie_{i-1,i} \prec \bowtie_{k,k+1}$, else $\bowtie_{k,k+1} \prec \bowtie_{i-1,i}$.

The partial order produced is transformed to *any* equivalent total order, and the join nodes are placed using the resulting total order. For example, consider the sub-plan P_1 of Figure 10. Since $\phi_S = \phi_T = \mathbf{T}$, the partial order is $\bowtie_{RS} \prec \bowtie_{ST} \prec \bowtie_{TU}$ which is a total order. For the sub-plan P_{31}, $\phi_S = \mathbf{F}$ and $\phi_T = \mathbf{T}$. This will produce the partial order $\bowtie_{ST} \prec \bowtie_{RS}, \bowtie_{ST} \prec \bowtie_{TU}$. Hence, the assignments of ϕ_S and ϕ_T alone cannot produce a total order. In the this sub-plan, S and T tuples are routed to $\bowtie_{ST}$ producing the intermediate result ST. This intermediate result can be routed to either $\bowtie_{RS}$ or $\bowtie_{TU}$. One more predicate, the intermediate predicate ϕ_{ST}, is needed to make the routing deterministic. Since in P_{31}, $\phi_{ST} = \mathbf{T}$, using the second rule, we get the constraint $\bowtie_{RS} \prec \bowtie_{TU}$, which makes the partial order total. Finally, consider the sub-plan P_2. Here, the constraints imposed by ϕ_S and ϕ_T are $\bowtie_{RS} \prec \bowtie_{ST}$ and $\bowtie_{TU} \prec \bowtie_{ST}$, which do not form a total order. However, this does not matter. P_3 represents the bushy plan $(R \bowtie S) \bowtie (T \bowtie U)$, and the cost estimation function will produce the same result regardless of the relative order of $\bowtie_{RS}$ and $\bowtie_{TU}$.

B. PROOF OF LEMMA 2

We need to prove that if P is an eddy-compliant CJP, then the cost function $\text{COST}_{Eddy}(P)$ computes $\|P\|_S$ correctly. $\|P\|_S$ is the sum of sizes of intermediate relations, where each intermediate relation is added to the cost only once. Consider the cost $\|P\|_S$ of the CJP of Figure 3

$$\|P\|_S = |RS_1| + |T_1U| + |S_2T_2| + |RS_1T_2| + |S_2T_1U| + |S_2T_2U|,$$

and the cost calculated by the cost function $\text{COST}_{NS}(P)$:

$$\text{COST}_{NS}(P) = 2|RS_1| + 2|T_1U| + |S_2T_2| + |RS_1T_2| + |S_2T_1U| + |S_2T_2U|.$$

While the terms added by $\text{COST}_{NS}(P)$ are correct, some terms are added multiple times. The cost function COST_{Eddy} is derived from the cost function COST_{NS} with a simple modification: In every joint probability, instead of including only the decision predicate assignments that are relevant to the join predicates, we include all the decision predicate assignments from the root to the current node. The

cost formula for decision nodes in Equation 2 remains the same. For completeness, the cost function for a join node $n = \rightarrow \bowtie_{ij} \rightarrow n'$ is presented below for cases 1, 2, and 4:

$$\textsc{Cost}_{\textit{Eddy}}(n, \Phi, \mathcal{M}) = \Pr(\bowtie_{ij}, \bigwedge_{\phi \in \Phi} \phi)|R_i||R_j| + \textsc{Cost}_{\textit{Eddy}}(n', \Phi, \mathcal{M} \cup \{(\{R_i, R_j\}, \{\bowtie_{ij}\})\}).$$

$$\textsc{Cost}_{\textit{Eddy}}(n, \Phi, \mathcal{M}) = \Pr(\bowtie_{ij}, \bigwedge_{\xi \in \Xi_1} \xi, \bigwedge_{\phi \in \Phi} \phi)|R_j| \prod_{R \in \mathcal{R}_1} |R| + \textsc{Cost}_{\textit{Eddy}}(n', \Phi, \mathcal{M} - \{m_1\} \cup \{(\mathcal{R}_1 \cup \{R_j\}, \Xi_1 \cup \{\bowtie_{ij}\})\}).$$

$$\textsc{Cost}_{\textit{Eddy}}(n, \Phi, \mathcal{M}) = \Pr(\bowtie_{ij}, \bigwedge_{\xi \in \Xi_1 \cup \Xi_2} \xi, \bigwedge_{\phi \in \Phi} \phi) \prod_{R \in \mathcal{R}_1 \cup \mathcal{R}_2} |R| + \textsc{Cost}_{\textit{Eddy}}(n', \Phi, \mathcal{M} - \{m_1, m_2\} \cup \{(\mathcal{R}_1 \cup \mathcal{R}_2, \Xi_1 \cup \Xi_2 \cup \{\bowtie_{ij}\})\}).$$

This modification does not alter the relation partitions. Since all the decision predicates that are relevant to the join predicates are included in the probabilities in both cost functions, the partition sizes that are added as terms to the cost are the same. What is different is the weight of these terms. The extra decision predicates included in $\textsc{Cost}_{\textit{Eddy}}(P)$ weight the partition sizes. For example, consider the cost contribution of the node $\bowtie_{RS}$ of the sub-plan P_{11} in Figure 3:

$$\Pr(\bowtie_{RS} = \mathbf{T}, \phi_S = \mathbf{T}, \phi_T = \mathbf{T})|R||S| = |RS_1|\frac{|T_1|}{|T|}.$$

While the cost function $\textsc{Cost}_{\textit{NS}}(P)$ adds to the total cost the whole size of the intermediate result $|RS_1|$, the cost function $\textsc{Cost}_{\textit{Eddy}}(P)$ weights it by $\frac{|T_1|}{|T|}$. When the cost function $\textsc{Cost}_{\textit{Eddy}}(P)$ evaluates the cost of the node $\bowtie_{RS}$ in P_{12}, it adds to the cost the term

$$\Pr(\bowtie_{RS} = \mathbf{T}, \phi_S = \mathbf{T}, \phi_T = \mathbf{F})|R||S| = |RS_1|\frac{|T_2|}{|T|}.$$

Added together, these terms result to the size $|RS_1|$ being included only once to the total cost.

The total cost of a CJP consists of sizes of intermediate relations $|\mathcal{X}|$. An intermediate relation $\mathcal{X}$ encompasses both a certain schema, and certain partitions of relations. The schema is produced by the join predicates, and the partitions are produced by the decision predicates. It is obvious that the terms produced by $\textsc{Cost}_{\textit{Eddy}}(P)$ are the same as the ones produced by $\textsc{Cost}_{\textit{NS}}(P)$, since the extra decision predicates added are exactly the ones that do not change the partitions. The difference is that in $\textsc{Cost}_{\textit{Eddy}}(P)$ some terms are weighted. We only need to show that these weights in multiple appearances of the same term will add to one:

Lemma 3. *If P is an eddy-compliant CJP, then if the cost of an intermediate relation $\mathcal{X}$ appears in $\textsc{Cost}_{\textit{Eddy}}(P)$, it appears exactly one time.*

PROOF. We assume the chain query graph

$$\mathcal{Q}(\{R_1, \ldots, R_n\}, \{\bowtie_{i,i+1} | i = 1, \ldots, n-1\}).$$

A generalization to tree query graphs is straightforward if we assume n-ary instead of binary decision predicates. Consider an intermediate relation $\mathcal{X}$ that involves the relations $R_i, \ldots, R_{i+k}$ in some partitioned form. Our proof is constructed as an induction over the cardinality of the intermediate relation $\text{card}(\mathcal{X}) = k + 1$. Specifically, we will show the following:

Basis An intermediate result $\mathcal{X}$ with $\text{card}(\mathcal{X}) = 2$ is included in the cost one or zero times.

Induction step If all intermediate results $\mathcal{X}$ with $\text{card}(\mathcal{X}) \leq k$ are included in the cost one or zero times, then all intermediate results $\mathcal{Y}$ with $\text{card}(\mathcal{Y}) = k + 1$ are included in the cost one or zero times.

We begin with the basis of the induction. Consider an intermediate relation $\mathcal{X}$ of cardinality 2, that involves the relations R_i and R_{i+1}, possibly in some partitioned form. We distinguish between three cases: (a) $i = 1$, (b), $i = n - 1$, and (c) $i \neq 1 \wedge i \neq n - 1$. Consider first case (a). Then, $\mathcal{X}$ involves the relations R_1 and R_2 and is formed only when the join $\bowtie_{1,2}$ is at the first level of the join tree. For the eddy CJP, this means that $\mathcal{X}$ is only formed in the sub-plans with $\phi_2 = \mathbf{T}$ (see Appendix A). As already discussed, a leaf of the tree of decision predicates is equivalent with an assignment of $\{\mathbf{T}, \mathbf{F}\}$-values to all the decision predicates $\mathcal{F}(P)$ in the CJP. The join $\bowtie_{1,2}$ is at the first level only at the sub-plans with $\phi_2 = \mathbf{T}$, regardless of the values of the rest of the decision predicates. However, note that the CJP $P(\mathcal{Q})$ is derived from the eddy CJP structure P_e, and may contain only a subset of the free predicates in $\mathcal{F}_e$ ($\mathcal{F}(P) \subset \mathcal{F}_e$). Hence some predicates from $\mathcal{F}_e$ (including ϕ_2) may not appear in the decision node tree. So, we distinguish between three sub-cases: (i) $\phi_2 \in \mathcal{F}(P)$ (meaning that ϕ_2 is a part of P), (ii) $\phi_2 = \phi_{\textit{true}}$ (meaning that only the upper sub-plan of ϕ_2 in the eddy CJP structure is included in P), and (iii) $\phi_2 = \phi_{\textit{false}}$ (meaning that only the lower sub-plan of ϕ_2 in the eddy CJP structure is part of P). In case (iii), $\mathcal{X}$ will never be formed (R_2 is routed to $\bowtie_{2,3}$), so $|\mathcal{X}|$ will not be included in the final cost. In case (ii) $\mathcal{X}$ is formed at every sub-plan. Hence, the total cost contribution will be

$$\sum_{\phi \in \{\mathbf{T}, \mathbf{F}\}} \Pr(\bowtie_{1,2}, \bigwedge_{\phi \in \mathcal{F}(P)} \phi)|R_1||R_2| = |R_1 \bowtie R_2|.$$

Finally, in case (i), $\mathcal{X}$ is formed only in the sub-plans with $\phi_2 = \mathbf{T}$. Then, the cost contribution will be

$$\sum_{\phi \in \{\mathbf{T}, \mathbf{F}\}} \Pr(\bowtie_{1,2}, \phi_2, \bigwedge_{\phi \in \mathcal{F}(P) - \{\phi_2\}} \phi)|R_1||R_2| = |R_1 \bowtie \sigma_{\phi_2}(R_2)|.$$

So, in any case, $|\mathcal{X}|$ is included with weight 1 in the cost, ether in the form $|R_1 R_2|$ or in the form $|R_1 R_2^1|$. This concludes case (a) of the induction basis. Case (b) is handled similarly. For case (c) ($i \neq 1, n-1$), an intermediate relation $\mathcal{X}$ that contains R_i and R_{i+1} will be created only in the sub-plans with $\phi_i = \mathbf{F}$ and $\phi_{i+1} = \mathbf{T}$. Again, we distinguish between three cases:

1. $\phi_i, \phi_{i+1} \notin \mathcal{F}(P)$ which has the following subcases
   (a) $\phi_i = \phi_{\textit{false}}$ and $\phi_{i+1} = \phi_{\textit{false}}$. Then, there are not sub-plans in which $\mathcal{X}$ is formed, so $|\mathcal{X}|$ does not appear in the total cost.
   (b) $\phi_i = \phi_{\textit{false}}$ and $\phi_{i+1} = \phi_{\textit{true}}$.
   (c) $\phi_i = \phi_{\textit{true}}$ and $\phi_{i+1} = \phi_{\textit{false}}$. Then, $|\mathcal{X}|$ does not appear in the total cost.
   (d) $\phi_i = \phi_{\textit{true}}$ and $\phi_{i+1} = \phi_{\textit{true}}$. Then, $|\mathcal{X}|$ does not appear in the total cost.
2. $\phi_i \in \mathcal{F}(P), \phi_{i+1} \notin \mathcal{F}(P)$. The case $\phi_i \notin \mathcal{F}(P), \phi_{i+1} \in \mathcal{F}(P)$ is dual. We have the following subcases:
   (a) $\phi_{i+1} = \phi_{\textit{false}}$. Then, $|\mathcal{X}|$ does not appear in the total cost.
   (b) $\phi_{i+1} = \phi_{\textit{true}}$.
3. $\phi_i, \phi_{i+1} \in \mathcal{F}(P)$. Then, $\mathcal{X}$ is formed only in the sub-plans with $\phi_i = \mathbf{F}$ and $\phi_{i+1} = \mathbf{T}$.

We need to prove that in cases 1(b), 2(b), and 3, $|\mathcal{X}|$ appears exactly once in the total cost. In case 1(b), $\mathcal{X}$ is formed in every sub-plan.

The contribution to the total cost is

$$\sum_{\phi\in\{\mathbf{T},\mathbf{F}\}} \Pr(\bowtie_{i,i+1}, \bigwedge_{\phi\in\mathcal{F}(P)} \phi)|R_i||R_{i+1}| = |R_i \bowtie R_{i+1}|$$

In case 2(b), $\mathcal{X}$ is only formed in the sub-plans with $\phi_i = \mathbf{F}$. The contribution to the total cost is

$$\sum_{\phi\in\{\mathbf{T},\mathbf{F}\}} \Pr(\bowtie_{i,i+1}, \neg\phi_i, \bigwedge_{\mathcal{F}(P)-\{\phi_i\}} \phi)|R_i||R_{i+1}| = |\sigma_{\neg\phi_i}(R_i) \bowtie R_{i+1}|.$$

In case and 3, $\mathcal{X}$ is only formed in the sub-plans with $\phi_i = \mathbf{F}$ and $\phi_{i+1} = \mathbf{T}$. The contribution to the total cost is

$$\sum_{\phi\in\{\mathbf{T},\mathbf{F}\}} \Pr(\bowtie_{i,i+1}, \neg\phi_i, \phi_{i+1}, \bigwedge_{\mathcal{F}(P)-\{\phi_i,\phi_{i+1}\}} \phi)|R_i||R_{i+1}| = |\sigma_{\neg\phi_i}(R_i) \bowtie \sigma_{\phi_{i+1}}(R_{i+1})|$$

So, $|\mathcal{X}|$ is always weighted by a factor of 0 or 1 in the total cost. This concludes the basis step.

For the induction step, consider the intermediate relation $\mathcal{X}$ of cardinality $k+1$, that contains the relations $R_i, \ldots, R_{i+k}$. This intermediate relation will be formed from a join node $\bowtie_{j,j+1}$ that will join the intermediate relations $\mathcal{X}_1$, $\mathcal{X}_2$, which contain the relations $R_i, \ldots, R_j$ and $R_{j+1}, \ldots, R_{i+k}$ respectively. Since card($\mathcal{X}_1$) $\leq k$ and card($\mathcal{X}_2$) $\leq k$, we know that $|\mathcal{X}_1|$ and $|\mathcal{X}_2|$ either do not appear in the total cost, or they appear with a factor of 1. Note that multiple intermediate results that contain the relations $R_i, \ldots, R_{i+k}$ can appear in the total cost, produced by all the joins $\bowtie_{j,j+1}$ with $j = i, \ldots, i+k-1$. However, these intermediate relations contain different partitions of the relations, and are thus different for the purposes of this proof. We can therefore focus on a particular value of j. Similarly to the base step, we have the following cases:

1. Either $|\mathcal{X}_1|$ or $|\mathcal{X}_2|$ do not appear in the total cost. Then, $|\mathcal{X}|$ does not appear in the total cost.
2. Both $|\mathcal{X}_1|$ and $|\mathcal{X}_2|$ appear in the total cost. Then, $\mathcal{X}$ is formed at the sub-plans with $\phi_1 = \phi_{i,\ldots,j} = \mathbf{F}$, and $\phi_2 = \phi_{j+1,\ldots,i+k} = \mathbf{T}$. This has again the following sub-cases:
   (a) $\phi_1, \phi_2 \notin \mathcal{F}(P)$, $\phi_1 = \phi_{\mathit{false}}$, and $\phi_2 = \phi_{\mathit{true}}$.
   (b) $\phi_1, \phi_2 \in \mathcal{F}(P)$.
   (c) $\phi_1 \in \mathcal{F}(P)$, $\phi_2 \notin \mathcal{F}(P)$, and $\phi_2 = \phi_{\mathit{true}}$.
   (d) $\phi_2 \in \mathcal{F}(P)$, $\phi_1 \notin \mathcal{F}(P)$, and $\phi_1 = \phi_{\mathit{false}}$.

We need to prove that $|\mathcal{X}|$ appears with a factor of 1 in the total cost in cases 2(a)–2(c) (case 2(d) is dual to case 2(c)). The fact that $\mathcal{X}_1$ and $\mathcal{X}_2$ are formed implies an assignment of all the base decision predicates $\phi_i, \ldots, \phi_{i+k}$, as well as an assignment of all the decision predicates on the intermediate relations that form $\mathcal{X}_1$ and $\mathcal{X}_2$. Let us call $\mathcal{F}_j$ the set of these assigned decision predicates and $\mathcal{A}_j$ the particular assignment. This assignment creates the partitions of the relations $R_i, \ldots, R_{i+k}$ particular to $\mathcal{X}_1$ and $\mathcal{X}_2$, but does not introduce factors in their cost (from the induction hypothesis). Let us denote by $\Xi_1(\Xi_2)$ the conjunction of join predicates in $\mathcal{X}_1(\mathcal{X}_2)$, and by $|\mathcal{R}|$ the size of the Cartesian product $R_i \times \cdots \times R_{i+k}$. For the case 2(a) above, the cost contribution is

$$\sum_{\phi\in\{\mathbf{T},\mathbf{F}\}} \Pr(\bowtie_{j,j+1}, \Xi_1, \Xi_2, \bigwedge_{\phi\in\mathcal{F}(P)-\mathcal{F}_j} \phi, \mathcal{A}_j)|\mathcal{R}| = |\mathcal{X}_1 \bowtie \mathcal{X}_2|.$$

In case 2(b), $\mathcal{X}$ appears only in the subplans with $\phi_1 = \mathbf{F}$ and $\phi_2 = \mathbf{T}$. The cost contribution is

$$\sum_{\phi\in\{\mathbf{T},\mathbf{F}\}} \Pr(\bowtie_{j,j+1}, \Xi_1, \Xi_2, \bigwedge_{\phi\in\mathcal{F}(P)-\mathcal{F}_j-\{\phi_1,\phi_2\}} \phi, \mathcal{A}_j, \neg\phi_1, \phi_2)|\mathcal{R}| = |\sigma_{\neg\phi_1}(\mathcal{X}_1) \bowtie \sigma_{\phi_2}(\mathcal{X}_2)|.$$

Finally, for case 2(c), the cost contribution is

$$\sum_{\phi\in\{\mathbf{T},\mathbf{F}\}} \Pr(\bowtie_{j,j+1}, \Xi_1, \Xi_2, \bigwedge_{\phi\in\mathcal{F}(P)-\mathcal{F}_j-\{\phi_1\}} \phi, \mathcal{A}_j, \neg\phi_1)|\mathcal{R}| = |\sigma_{\neg\phi_1}(\mathcal{X}_1) \bowtie \mathcal{X}_2|.$$

So, $|\mathcal{X}|$ will appear in the total cost with a factor of 0 or 1. □

C. GREEDY ALGORITHM DETAILS

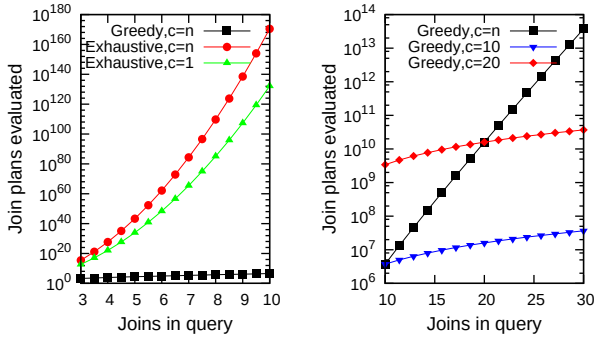
Pseudocode for our greedy algorithm is given in Algorithm 1. Initially (in function GREEDY-HPE), a traditional query optimizer is invoked to find the best monolithic plan for the query. We use the KBZ polynomial-time algorithm in our implementation [15]. This plan corresponds to a $\{\phi_{\mathit{false}}, \phi_{\mathit{true}}\}$-value for each predicate in the set of predicates of the eddy skeleton routing policy, $\mathcal{F}_e$. Then, the recursive function GREEDY-HPE-REC is called. It takes the following arguments: $\mathcal{F}$ is the set of predicates from $\mathcal{F}_e$ that have not been given values; $\mathcal{B}$ is the set of bound predicates by the algorithm (i.e., predicates that are already part of the CJP and are assigned a value not from $\{\phi_{\mathit{true}}, \phi_{\mathit{false}}\}$); $\mathcal{A}$ is an set of predicate values from $\{\phi_{\mathit{true}}, \phi_{\mathit{false}}\}$ for all the predicates in $\mathcal{F}$; $\mathcal{C}$ is a $\{\mathbf{T}, \mathbf{F}\}$ assignment for the predicates in $\mathcal{B}$, valid in the *current sub-plan*; C_{min} is the cost of the best CJP found so far; c is the partitioning budget. The initial values for these arguments can be seen in line 5 of Algorithm 1.

The function GREEDY-HPE-REC examines all the free predicates in $\mathcal{F}$. For each possible value of every free predicate ϕ, it evaluates the cost of the CJP that uses only ϕ as a decision predicate, and the upper and lower join orders honor the eddy restrictions. If it finds that such a predicate ϕ^* improves the total cost, it introduces it to the CJP, and recurses to the two sub-plans. The recursion finishes if such a predicate was not found, or if c decision predicates have already been used.

Algorithm 1 Greedy Horizontal Partitioning with Eddies, initialization and main algorithm

```
1: function GREEDY-HPE(Q)
2:     Convert Q to the eddy skeleton routing policy π_e(Q, F)
3:     Find the optimal plan P* for the query, and its cost C*
4:     Find the assignment A*(F_e) that corresponds to the plan P*
5:     return GREEDY-HPE-REC(F, ∅, A*, ∅, C*, c)
6: function GREEDY-HPE-REC(F,B,A,C,C_min,c)
7:     φ* = null; C* = C_min; P* =null
8:     for φ ∈ F do
9:         for all possible values A ≤ v for φ do
10:            A' = A[F − {φ}]
11:            A_φ = A' ∪ {φ = φ_true} ∪ C
12:            A_¬φ = A' ∪ {φ = φ_false} ∪ C
13:            if LEGAL(A_φ) ∧ LEGAL(A_¬φ) then
14:                P_φ = PLAN(A_φ)
15:                P_¬φ = PLAN(A_¬φ)
16:                P = → φ ↗ P_φ
                           ↘ P_¬φ
17:                C = COST(P)
18:                if C ≤ C* then
19:                    C* = C; P* = P; φ* = φ
20:    if φ* = null ∨ c = 0 then return PLAN(A ∪ C)
21:    P_1 = GREEDY-HPE-REC(F − {φ*}, B ∪ {φ*}, A[F − {φ*}], C ∪ {φ* = T}, C*, c − 1)
22:    P_2 = GREEDY-HPE-REC(F − {φ*}, B ∪ {φ*}, A[F − {φ*}], C ∪ {φ* = F}, C*, c − 1)
23:    return → φ* ↗ P_1
                   ↘ P_2
```

We can measure the cost of the greedy and exhaustive algorithms as the number of *join plans* that will be evaluated. For simplicity,



(a) Cost of exhaustive search. (b) Cost of greedy search.

Figure 11: Number of join plans considered by the exhaustive and greedy search in the eddy CJP space.

we assume that the decision predicates in a CJP form a complete binary tree. This is true for chain queries when no decision predicates on intermediate results are used, but not true in the general case. Under this assumption, if a CJP contains k decision predicates, we need to pay $C(k) = 2^k$ in order to evaluate its cost. Assume that the eddy CJP structure contains $|\mathcal{F}_e| = m$ binary decision predicates. This number can be easily calculated using the query graph and the number of joins in the query n. For chain queries, $m = \frac{n(n-1)}{2} = \Theta(n^2)$, while for star queries $m = \Theta(n^3)$.

The exhaustive algorithm needs to partition the set $\mathcal{F}_e$ of m decision predicates into two sets: a set $\mathcal{F}_1$ of predicates that will be a part of the CJP, and a set $\mathcal{F}_2$ of predicates that will be assigned values from $\phi_{true}, \phi_{false}$. The sizes of the sets, given the partitioning budget c are $|\mathcal{F}_1| = c$ and $|\mathcal{F}_2| = m - c$. For each combination of these sets, and for each combination of values of the decision predicates, the algorithm needs to evaluate the cost of a CJP of size c. Assuming that the domain size of all descriptive attributes is d, the cost of the exhaustive algorithm is

$$\textsc{Exhaustive}(m, c) = \left(\begin{array}{c} m \\ c \end{array} \right) d^c 2^{m-c} C(c)$$

For the greedy algorithm, consider the case where $c-i$ iterations have already been executed, and there are i iterations left. Then, the size of the argument $\mathcal{F}$ is $|\mathcal{F}| = m - c + i$. The greedy algorithm will for every predicate and every predicate value in that set, evaluate the cost of a CJP with one decision predicate. Then, it will recurse with $i - 1$ iterations left:

$$\textsc{Greedy}(m, i) = d(m - c + i)C(1) + 2\textsc{Greedy}(m, i - 1)$$

Since initially there are c iterations left, we are interested in the cost $\textsc{Greedy}(m, c)$. When there is no restriction in the number of partitions ($c = m$), the cost of both algorithms grows super-exponentially with respect to n. However, consider the case where we restrict the partitions to the number of joins in the query, $c = n$, allowing essentially one decision predicate for each base relation. The number of join plans evaluated by the exhaustive and greedy algorithms are shown in Figure 11(a) for $n = 3$ to $n = 10$ joins in the query. While the cost of exhaustive grows as $\Theta(2^{n^2})$, the cost of the greedy search grows as $\Theta(2^n)$. Even restricting to a fixed number of partitions ($c = 1$) cannot alleviate the super-exponential growth of the exhaustive algorithm. The cost of the greedy algorithm for fixed iterations and iterations equal to the number of joins is shown in Figure 11(b) for $n = 10$ to $n = 30$ joins. While the growth is exponential, it can be kept reasonable with fixed iterations at the cost of CJPs of reduced quality.

D. IMPLEMENTATION DETAILS

Our prototype is based on the PostgreSQL codebase, and uses the eddy implementation described in [5]. Specifically, we have created two new PostgreSQL operators: the eddy and the SteM operator. The SteM is a main memory hash table that stores base or intermediate tuples and has an insert/probe interface. A join is executed using two SteMs. The eddy operator performs the routing, via an internal routing policy structure. The routing policy structure is a mapping from a tuple signature to zero or more operators. We are using the routing policy as the search space representation, which is equivalent to using CJPs. All our additions in the PostgreSQL code are in the execution engine, and in fact we have completely bypassed the PostgreSQL optimizer. The construction of the junction tree is done outside the PostgreSQL code, with a simple JDBC program. The junction tree is read into memory when PostgreSQL starts. A more careful implementation would reuse the PostgreSQL catalog and optimizer, and subsequently have better optimization performance.

E. DATA GENERATION

We generate synthetic data for a join query $R_1 \bowtie \cdots \bowtie R_n$, simultaneously controlling three parameters: the number of tuples in each relation, N, the selectivities of the join operators $s_1, \ldots, s_n$, and the Pearson correlation coefficient, r. The latter controls the degree of correlation in the database and takes values in $[-1, 1]$: if $r = 0$, the database should be fairly uniform; if $|r| = 1$, an almost perfect partitioning scheme of the relations should exist. The data generation algorithm takes as input the desired parameter values $N, \{s_1, \ldots, s_n\}, r$, as well as the Markov network of the database.

Consider the example query and its Markov network shown in Figure 4. The variables that are connected in the Markov network (e.g., X and $\mathcal{J}_{RS}$) have a correlation coefficent with absolute value equal to r. In order to simulate a perfect paritioning for high correlations, we flip the sign of the correlation coefficient at every edge of the graph. For example, the correlation coefficient between X and $\mathcal{J}_{RS}$ is equal to r, the correlation coefficient between $\mathcal{J}_{RS}$ and Y is equal to $-r$, etc. Variables that are not adjacent in the graph are correlated only indirectly. Under certain assumptions, the correlation coefficient between two variables A and B is equal to $(-r)^k$, where k is the number of edges in the path between A and B. For example, the correlation coefficient of X and J_{ST} is $-r^3$. This procedure gives the correlation matrix $R = [r_{ij}]$ whose entries are the correlation coefficients between every pair of random variables. Then, the method described by Fackler [11] is used to create a $N \times 7$, $[0, 1]$-valued matrix $D = [d_{ij}]$ that conforms to the correlation matrix R. Each column of the matrix D is used to generate a random variable. For descriptive attributes, the $[0, 1]$ value d_{ij} is scaled accordingly to the attribute's domain. For join indicators, the value **T** is chosen if $d_{ij} > 0.5$, and the value **F** is chosen otherwise. Finally, each value of a join indicator j_{ij} is used to create two values of the join attributes for the relations. The values should be equal if $j_{ij} = \mathbf{T}$, and not equal otherwise. In addition, the domain of the join attributes is chosen so that the selectivity of the join is equal to the input given to the data generation algorithm.

Acknowledgements

This research was conducted when C. S. Jensen was a full-time Professor at Aalborg University. C. S. Jensen is an Adjunct Professor at University of Agder, Norway.