MONSOON: Multi-Step Optimization and Execution of Queries with Partially Obscured Predicates

Sourav Sikdar
Rice University
ss107@rice.edu

Chris Jermaine
Rice University
cmj4@rice.edu

ABSTRACT

User-defined functions (UDFs) in modern SQL database systems and Big Data processing systems such as Spark—that offer API bindings in high-level languages such as Python or Scala—make automatic optimization challenging. The foundation of modern database query optimization is the collection of statistics describing the data to be processed, but when a database or Big Data computation is partially obscured by UDFs, good statistics are often unavailable.

In this paper, we describe a query optimizer called the Monsoon optimizer. In the presence of UDFs, the Monsoon optimizer may choose to collect statistics on the UDFs, and then run the computation. Or, it may optimize and execute part of the plan, collecting statistics on the result of the partial plan, followed by a re-optimization step, with the process repeated as needed. Monsoon decides how to interleave execution and statistics collection in a principled fashion by formalizing the problem as a Markov decision process.

ACM Reference Format:
Sourav Sikdar and Chris Jermaine. 2020. MONSOON: Multi-Step Optimization and Execution of Queries with Partially Obscured Predicates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389728

1 INTRODUCTION

User-defined functions (UDFs) in modern SQL and Big Data processing systems such as Spark [46]—that offer API bindings in high-level languages such as Python or Scala—pose many challenges [37]. One major challenge is that it is difficult to perform cost-based optimization over UDFs. For example, consider the following PySpark code:

```
docNameAndText = validLines.map
    (lambda x : (x[x.index('id="') + 4 :
     x.index('" url=')], x]))
docInfo = docNameAndText.join (docInfo)
docInfoWithAuthor = docInfo.map
     (lambda x : (x[1][1][x[1][1].index(
      'author="') + 7 : x[1][1].index(
      '" id=')], (x[1][0], x[1][1])))
docAndAuthorInfo = docInfoWithAuthor.join
     (authorInfo)
```

This code first extracts a document name from each document in a large text corpus (stored in the RDD `validLines`), and then joins `validLines` (on the document name) with the `docInfo` data set. The code then extracts each document's author and joins the result (on the author name) with the `authorInfo` data set. Because the system is aware that there are two equi-joins but is unaware what the arguments to the joins are due to the presence of UDFs, these joins are said to make use of *partially obscured predicates*.

Modern cost-based optimizers require statistics on the underlying data, such as the number of distinct document names in `validLines` and `docInfo`, so that they can estimate the size of the resulting join. If these statistics depend upon non-trivial code that includes API calls (like `x.index()`), obtaining these statistics from a static analysis of the code can be quite difficult.

One way to optimize a computation that contains such black-box UDF functions is to execute the UDFs over the data, and count the statistics on the resulting data. If the UDF is applied to the input data in one pass, then the result can be pipelined into a probabilistic counting [44] or sketching algorithm [22] to compute required statistics, such as distinct value counts. However, there are some problems with this. First, executing the UDF and counting the results can add a significant overhead to the computation. For some computations—for example, a join of a huge table with a small table where the output from the join cannot be much larger than the input—the statistics may not really matter. It may be best to simply run the computation rather than paying the cost of first computing accurate statistics.

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389728

A second problem is that collecting statistics before execution is not a universal solution to the problem, as it may be the case that some UDFs do not even take the base data as input. For example, consider an SQL query of the form:

```
SELECT  *
FROM  R, S, T, ...
WHERE  𝓕1(R, S) = 𝓕2(T)  AND  ...
```

Here, it is not possible to compute statistics for $\mathcal{F}_1$ until *after* a join of R and S has been performed because $\mathcal{F}_1$ takes attributes from both sets as input.

Balancing Exploration and Execution. In this paper, we describe an optimizer called Monsoon (Multi-Step Optimization and executiON) that is able to decide when it is better to be safe and collect statistics on the number of distinct values returned by a UDF—and pay the cost to materialize and scan an intermediate result (or one of the inputs)—and when it is better to be bold and simply guess at and run a reasonable plan. Monsoon is able to execute a computation in a series of steps, running part of the computation, observing some statistics, then re-planning and executing once again.

To interleave execution and statistics collection in a principled fashion, we model the problem of iteratively executing a partial plan and collecting statistics as a Markov decision process (MDP) [23, 34]. The MDP models the uncertainty in the output of a UDF using a prior on the number of distinct values. Query optimization is performed not by dynamic programming, as in a classical relational system, but using Monte Carlo tree search (MCTS) [9, 27].

Classically, interleaved query planning and execution has been used to correct or avoid errors made by the optimizer cost model [4, 6, 15, 25]. In contrast, Monsoon assumes a correct cost model and uses interleaved execution in order to collect statistics that are unknown at optimization time. Ideas related to Monsoon have been explored before; Microsoft SQLServer, for example, can partially execute a query to materialize and collect accurate statistics on table-valued functions [1], and subsequently resume optimization and execution of the remaining operations. Our goal is to design a principled framework within which such iterative optimization and execution can take place.

Our Contributions. Specific contributions are:

- We introduce the idea of using an MDP and MDP solver as a method for handling UDFs in Big Data and relational computations. Our methods provide a framework for optimizing computations that contain UDF functions whose statistics are unknown.
- We carefully examine the empirical effect of different priors, and propose a general purpose, "spike and slab" prior that works well on a variety of benchmarks.
- While our focus is on data processing in the presence of UDFs, the ideas described in this paper could, in principal, be applied to any scenario where statistics are missing at optimization time [13, 14], or when cardinality estimation is challenging due to the presence of correlations or heavy skew.

2 EXAMPLE AND MOTIVATION

2.1 UDFs and Logical Query Optimization

We are interested in the optimization of multi-table (or multi-set) queries where statistics over one (and sometimes all) of the attributes referenced in the underlying computation are unavailable, typically because the predicate(s) evaluated by the computation are partially obscured by the presence of UDFs.

For example, imagine a Spark SQL query over two Python DataSets. One of the DataSets describes customer orders, the second describes online sessions. The order DataSet has a data structure containing all of the items purchased in the order. The goal is to find potentially fraudulent orders by finding pairs of identical orders placed on a specific date by pairs of customers who logged on from the same city (the city is extracted from the sessions' IP address). The corresponding SQL query is:

```
SELECT c1.name, c2.name
FROM order o1, order o2, sess s1, sess s2
  Intersection (o1.items, o2.items) =
  Union (o1.items, o2.items) AND
  ExtractDate (o1.when) = '1/11/19' AND
  ExtractDate (o2.when) = '1/11/19' AND
  o1.cID = s1.cID AND o2.cID = s2.cID AND
  o1.cID <> o2.cID AND
  City (s1.ipAdd) = City (s2.ipAdd)
```

Any reasonable plan will probably first filter each instance of the order DataSet on the specified date. The next action is unclear. We could compute a cross product over the two instances of order, filtering and accepting only those pairs with matching sets of items, followed by a pair of joins with sess. This gives the plan $((\texttt{o1} \times \texttt{o2}) \bowtie \texttt{s1}) \bowtie \texttt{s2}$, with a final filter for City (s1.ipAdd) = City (s2.ipAdd). Or, one could filter o1 and o2, then join each with sess, giving $(\texttt{o1} \bowtie \texttt{s1}) \bowtie (\texttt{o2} \bowtie \texttt{s2})$. The plan would finish with a final filter checking for the equality of o1.items and o2.items.

If a customer has many sessions, and the predicate City (s1.ipAdd) = City (s2.ipAdd) is not selective, it may be much better to run the cross product first (before the joins with sess have a chance to increase the number of order pairs to check) meaning the first plan is preferred. If things are reversed (the check for equality is not selective, but the check for the same city is), the second plan is preferred.

The presence of UDFs clearly makes it very difficult to choose between these options. If all UDFs present in a query are over a single set, then a viable solution is to evaluate each UDF during a pre-processing step, and then compute the number of distinct values (and the heavy hitters i.e., most common values with their frequencies, if needed [2]). Then, the query can be optimized and executed in the traditional fashion. This can be expensive, rendering an overall execution time that is higher than necessary. Sometimes, it may be better to simply run the query using a (possibly) poor plan, and "take our chances", rather than undertaking the expense of scanning the sets and applying the user-defined functions beforehand. Furthermore, this simple tactic becomes problematic in the case of multi-table UDFs, as it requires sampling from a join or a cross-product.

2.2 A Bayesian Approach

The problem is that it is not clear whether it is better to collect the statistics before (or during) query execution, or to just execute and try our luck. We aim to build a query optimizer that is able to intelligently choose among such options. Our approach takes a Bayesian view of the problem, modeling uncertainty as probability. That is, we allow for a prior on the number of distinct values output by each user-defined function, and given such a prior, we choose a sequence of actions so as to minimize the expected overall execution time. In practice, we find that choosing a prior that works well in most cases does not require much engineering effort.

Given the various priors, the simplest sequence of actions would be to simply choose a logical plan for the query, and then execute the plan, even if many of the distinct values are unknown. However, this most likely will be a sub-optimal strategy. Thus, we also allow the system to materialize the result of subcomputations needed to perform the overall computation, and then compute statistics over those intermediate results, using the computed statistics to optimize the rest of the computation. The simplest example of this is performing a scan over one or more of the input sets to collect statistics before the computation begins. Under certain circumstances (such as when a user-defined function is defined over multiple input sets), such statistics collection will happen later in the computation. But in other situations, the best execution plan may involve executing a series of subcomputations, collecting statistics over each and re-optimizing after each statistics collection.

Note that this approach resembles classical mid-query re-optimization [6, 25] in that a query is run as a series of planning and execution steps. However, this classical work assumes that all parameters (cardinalities, distributions, histograms, etc.) required for query optimization are known before compilation, and a classical optimizer can be invoked as is to generate a query plan. The execution of the query plan is subsequently monitored to correct or avoid errors made by the optimizer cost model. In contrast, our goal is to use interleaved execution in order to collect statistics that are unknown at optimization time.

$d(\mathcal{F}_2, \mathsf{S})$	$d(\mathcal{F}_4, \mathsf{T})$	Optimal Plan	Int. Tuples
1	1	Both	10 million
1	10000	$((\mathsf{R} \bowtie \mathsf{T}) \bowtie \mathsf{S})$	1 million
10000	1	$((\mathsf{R} \bowtie \mathsf{S}) \bowtie \mathsf{T})$	1 million
10000	10000	Both	1 million

Table 1: Enumerating attribute cardinalities.

2.3 Optimal Multi-Step Execution

For an example of how one may choose whether to collect statistics early on, or to just guess at a reasonable join order and see what happens, consider the following SQL query.

```
SELECT  SUM(R.a)
FROM  R, S, T
WHERE  𝓕1(R) = 𝓕2(S)   AND   𝓕3(R) = 𝓕4(T)
```

We have the following statistics on the size of the inputs:

- $c(\mathsf{R}) = 1000000$
- $c(\mathsf{S}) = 10000$
- $c(\mathsf{T}) = 10000$

And we have the following priors on the number of distinct values output by each UDF:

- $\Pr(d(\mathcal{F}_1, \mathsf{R}) = 1000) = 1.0$
- $\Pr(d(\mathcal{F}_2, \mathsf{S}) = 1) = 0.5$
  $\Pr(d(\mathcal{F}_2, \mathsf{S}) = 10000) = 0.5$
- $\Pr(d(\mathcal{F}_3, \mathsf{R}) = 1000) = 1.0$
- $\Pr(d(\mathcal{F}_4, \mathsf{T}) = 1) = 0.5$
  $\Pr(d(\mathcal{F}_4, \mathsf{T}) = 10000) = 0.5$

Without considering physical implementations, the optimizer has three equivalent logical plans to choose from: $((\mathsf{R} \bowtie \mathsf{S}) \bowtie \mathsf{T})$, $((\mathsf{R} \bowtie \mathsf{T}) \bowtie \mathsf{S})$ and $((\mathsf{S} \times \mathsf{T}) \bowtie \mathsf{R})$.

Most optimizers would avoid the plan $((\mathsf{S} \times \mathsf{T}) \bowtie \mathsf{R})$, involving a cross product between S and T. The plan is likely to be sub-optimal as the cross product produces 100 million objects. Hence, the optimizer is left with choice between one of the two plans $((\mathsf{R} \bowtie \mathsf{S}) \bowtie \mathsf{T})$ and $((\mathsf{R} \bowtie \mathsf{T}) \bowtie \mathsf{S})$.

Given the prior described above, Table 1 shows the different possibilities for the domain cardinalities, along with the corresponding optimal plan and the number of intermediate objects generated by the first join in each of those plans, assuming the classical cost model for join result size holds [19, 38]. Subsequently, we assume that the cost of a plan is the number of intermediate objects produced, though other, more sophisticated metrics are certainly possible.

We want our optimizer to evaluate the merits of various possibilities, whether it is better to collect statistics on some sets, or to just guess at a reasonable join order based on the priors. Imagine that the optimizer simply guesses the join order $((\mathsf{R} \bowtie \mathsf{S}) \bowtie \mathsf{T})$ based on priors, without additional statistics. In three out of the four aforementioned scenarios, it turns out to be the optimal plan. However, in one of the cases (row 2 of Table 1), it is a sub-optimal plan. Again assuming that the classical join cost model, it will produce 10 million intermediate objects, which is 10× the number of objects produced by the optimal plan. Similarly, choosing the join order $((\mathsf{R} \bowtie \mathsf{T}) \bowtie \mathsf{S})$, can produce 10× the number of objects produced by the optimal plan if we fall in row 3 of Table 1.

Note that the classical notion of least-expected cost optimization [13, 14] is not particularly helpful here. Both the second and third rows in the table have the same expected number of intermediate objects.

As an alternative, imagine the optimizer chooses to scan the set S and collect statistics on $d(\mathcal{F}_2, \mathsf{S})$. If the domain cardinality is revealed to be 10000, then the optimizer can choose an optimal join order $((\mathsf{R} \bowtie \mathsf{S}) \bowtie \mathsf{T})$ with certainty. In contrast, if the domain cardinality is revealed to be 1, the optimizer can again choose an optimal join order $((\mathsf{R} \bowtie \mathsf{T}) \bowtie \mathsf{S})$ with certainty. By paying an additional cost of scanning and collecting additional statistics, we can ensure that the optimizer chooses the optimal plan for all four possible scenarios. A similar case can be argued in favor of scanning the set T and collecting statistics on $d(\mathcal{F}_4, \mathsf{T})$.

Scanning either S or T to collect statistics and then optimizing based upon those statistics is the strategy with the lowest *expected* cost among all query plans. Excluding the cost to write the final result, and the cost to scan the input data, the expected cost of either plan is $10^4 + 0.25(10^7) + 0.75(10^6)$ (that is, for the cost of scanning 10^4 objects and computing statistics, we are assured an optimal plan). In contrast, the best plan that does not scan either S or T has expected cost $0.5(10^7) + 0.5(10^6)$—nearly twice as large—as it has a 50% chance of picking a terrible plan.

3 BACKGROUND

3.1 Problem Scope

Consider the following simplified grammar for the Boolean expression appearing in the WHERE clause of a query:

$$\begin{aligned} \mathsf{boolExp} &\rightarrow \mathsf{boolExp\ boolOp\ boolExp} \\ &\quad \mid \mathsf{funcEval} \mid \mathsf{value\ compOp\ value} \\ \mathsf{value} &\rightarrow \mathsf{attRef} \mid \mathsf{const} \mid \mathsf{funcEval} \end{aligned}$$

Here, boolOp refers to AND or OR, funcEval refers to the evaluation of a UDF, compOp refers to >, =, etc., attRef is a reference to an attribute, and const is a constant value.

A predicate constructed using this grammar is *partially obscured* if it utilizes the production rule value → funcEval. We constrast this with a *fully obscured* predicate which utilizes the production rule boolExp → funcEval. Optimizing for such predicates likely involves semantic analysis of the predicate's code [17, 18, 35, 36] to translate them into equivalent relational algebra expressions. Such analysis can be difficult in the general case, where the code for the UDF may not even be available (for example, the UDF may be compiled into a shared library that is dynamically loaded by the system). In the remainder of the paper, without loss of generality, we assume a purely relational system with an SQL programming interface (the ideas in the paper are easily extended to systems such as Spark [46] or our own PlinyCompute [48]).

Also without loss of generality, we assume that *all* values referenced in the WHERE clause of the query are produced by UDFs. If they are not, and statistics on a referenced function are available, this can be handled within our framework by simply initializing the optimization problem so that any relevant statistics are known.

To keep the scope of the paper manageable, we consider join ordering. Physical optimization is left to future work.

3.2 QO as a Decision Process

A *sequential decision process* describes a setting in which a fixed set of choices are available to an agent, but the effects of those choices are not completely known. Choosing the best action requires planning. Often the immediate effects of an action are easy to gauge but the long term effects are not obvious. Actions that produce immediate rewards can have debilitating future effects. The goal in a decision process is to trade-off immediate gains versus future rewards, to yield the best possible solution.

Query optimization for computations with opaque UDFs can be seen as a sequential decision process, where at every step the optimizer needs to choose from a set of available actions, like composing two RA expressions to produce a more complicated expression, or choosing to execute a RA expression and materialize the result to collect its statistics, where the impact of such decisions might only be visible much later. Our goal is to build an optimizer that can weigh such different actions and plan accordingly.

We are not the first to view query optimization as a decision process [28, 31, 32, 42]. Notably, there has been a flurry of recent work that views the classical query optimization problem as a decision process where the task of choosing a join order is cast as a series of sequential decisions that build up a query plan. This recent work uses a sequential formulation of query optimization to permit the application reinforcement learning (RL) algorithms [39] to the problem of query optimization, with the hope that RL can solve some of the problems that have long plagued optimizers, such as the exponential running time of dynamic programming optimizers [38], or the inaccuracy in cost models [24].

Our own motivation (and formulation) is different. We are specifically interested in optimizing queries with partially-obscured predicates. These are queries for which we have *too little information to optimize in the traditional manner*, as opposed to learning to optimize queries in a workload.

3.3 Review of MDPs

A Markov decision processes (MDP) [23, 34] is a special type of sequential decision process. Since we cast the optimization problem over partially-obscured predicates as an MDP, we briefly review MDPs now.

An MDP is a tuple $\langle \Sigma, A, T, R, \gamma \rangle$. A state $s \in \Sigma$ is a description of the environment at a particular point in time. An agent navigates in the state space Σ by making decisions from the action set A (alternatively, from A_s, the set of actions available from state $s \in \Sigma$).

The dynamics of the underlying system are modeled by transition probabilities $T : \Sigma \times A \times \Sigma \longrightarrow [0, 1]$ and a reward structure $R : \Sigma \times A \times \Sigma \longrightarrow \mathbb{R}$, as follows: when, at time tick t, the agent chooses action $a_t \in A_s$, in state $s_t \in \Sigma$, then, with probability $T(s_t, a_t, s_{t+1})$ it makes a transition to the next state s_{t+1} and receives a reward $R(s_t, a_t, s_{t+1})$. This happens independently of the history of the states, actions, and rewards—the Markov property. Finally, the discount factor $\gamma \in [0, 1]$ balances the importance of future rewards compared to present rewards.

The solution of an MDP is a *policy* $\pi : \Sigma \longrightarrow A$: a function π that specifies the action $\pi(s)$ that the agent will choose in state s. The policy "controls" the actions of the agent as it moves from state to state. In general, the goal is to try to choose a policy that will tend to increase the rewards that the agent will receive. From start state s_0 expected reward for policy π is computed as

$$V^{\pi}(s_0) = E\Big[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \Big| \pi \Big]$$

"Solving" an MDP requires computing a policy π^* that maximizes expected long-term reward:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} V^{\pi}(s_0)$$

That is, the goal is to choose a policy π^* that, out of all possible policies, maximizes the expected reward that the agent will receive as it transitions from state to state.

4 MDP FORMULATION

To define our particular MDP, we must describe the states, actions, transitions, and rewards that constitute the decision process. In this section, we describe our MDP in detail.

4.1 States

In an MDP, a state is a description of the environment at a particular point in time. During optimization and execution of a query, the current state consists of three sets,

- A set of RA expressions (or partial plans) $\mathcal{R}_p$ that have been constructed so far, but not executed yet (the subscript p denotes "planned").
- A set of executed and materialized RA expressions $\mathcal{R}_e$ (the subscript e denotes "executed").
- A set of statistics $\mathcal{S}$ on materialized RA expression (or input data sets).

For example, re-consider the example of Section 2.3. Initially, $\mathcal{R}_p$ is empty, as we have not yet developed any plans. Initially, $\mathcal{R}_e$ is the set $\{\mathsf{R}, \mathsf{S}, \mathsf{T}\}$, as each of these input sets has presumably been materialized and is ready to be processed. And initially, $\mathcal{S}$ contains whatever statistics are available on the expressions in $\mathcal{R}_e$—we assume that the all input set sizes are available—so we may have $\mathcal{S} = \{c(\mathsf{R}) = 10^6, c(\mathsf{S}) = 10^4, c(\mathsf{T}) = 10^4\}$. If we happen to have distinct values counts (the other type of statistic available), we may also have the item $d(\mathcal{F}_1(\mathsf{R})) = 10^3$ in $\mathcal{S}$, indicating that there are a thousand distinct values for $\mathcal{F}_1(\mathsf{R})$.

Later on, imagine that we have executed and materialized RA expression $(\mathsf{R} \bowtie \mathsf{T})$. At this point, $\mathcal{R}_p$ is still empty, $\mathcal{R}_e = \{\mathsf{R}, \mathsf{S}, \mathsf{T}, (\mathsf{R} \bowtie \mathsf{T})\}$, as $(\mathsf{R} \bowtie \mathsf{T})$ has now been executed. $\mathcal{S}$ may contain additional statistics on the result of $\mathsf{R} \bowtie \mathsf{T}$.

4.2 Actions

The actions available can be categorized in two classes:

- We can modify the set of unexecuted plans $\mathcal{R}_p$. This corresponds to query planning.
- We can execute the RA expressions in $\mathcal{R}_p$.

The various modifications to $\mathcal{R}_p$ each correspond to taking another step in building a query plan. At a high level, there are two categories of modifications to $\mathcal{R}_p$: first, we can add to $\mathcal{R}_p$ a subplan that is topped with a statistics collection operator, and second, we can choose to join two existing expressions.

Here are the two options for applying a statistics collection operator to the output of an expression:

(1) We can copy a RA expression r from $\mathcal{R}_e$ to $\mathcal{R}_p$, and apply the Σ operator—the statistics collection operator–to it. When $\Sigma(r)$ is executed, it computes the number of distinct values returned by r for all UDFs that are referenced in the query to be optimized. If the optimizer makes this move, it has decided to collect statistics on (the materialized) RA expression r.
(2) We can replace an RA expression r from $\mathcal{R}_p$ with $\Sigma(r)$. If the optimizer makes this move, it has decided that it will materialize r and collect statistics on it.

And here are the three options that correspond to deciding to join two expressions:

(1) For two RA expressions r_1 and r_2 from $\mathcal{R}_e$, we add $(r_1 \bowtie r_2)$ to $\mathcal{R}_p$. This move corresponds to deciding to join two already-materialized expressions.
(2) We replace two RA expressions r_1 and r_2 from $\mathcal{R}_p$—neither of which contains the statistics collection operator Σ—with $(r_1 \bowtie r_2)$. This move corresponds to deciding to join two not-yet-materialized expressions. Note that we cannot join two expressions when one of them is topped by a statistics collection operator.
(3) Finally, for $r_1 \in \mathcal{R}_e$ and $r_2 \in \mathcal{R}_p$ where r_2 does not contain the statistics collection operator Σ, we replace $r_2 \in \mathcal{R}_p$ with $(r_1 \bowtie r_2)$. This corresponds to joining a materialized and not-yet-materialized expression.

The final move that the optimizer can make is to decide to execute and materialize each of the expressions in $\mathcal{R}_p$. As we describe in detail subsequently, this has the effect of moving all expressions in $\mathcal{R}_p$ to $\mathcal{R}_e$.

4.3 Transitions

Overview: How transitions affect the state. As described above, there are two types of actions that the optimizer can take. The first type of action simply modifies the current plan, by (for example) deciding to join two expressions. Plan modifications are always deterministic. Thus, each of the actions described above that modifies the set of RA expressions that we plan to execute initiates a deterministic state transition that simply modifies $\mathcal{R}_p$.

The second type of action—materializing each of the expressions in $\mathcal{R}_p$—results in a (partially) non-deterministic state transition. Note that even in this second case, a portion of the transition is deterministic. As each of the plans in $\mathcal{R}_p$ are executed, this has the effect of moving the plan from $\mathcal{R}_p$ to $\mathcal{R}_e$.[1] However, execution and materialization also results in a non-deterministic update to the set of statistics, as statistics regarding the executed plans are observed. Prior to execution and materialization of a join expression such as $r = r_1 \bowtie r_2$, we have some idea (quantified as a prior distribution) as to the value of the statistics describing r, but the actual execution of r "hardens" the statistics over r, non-deterministically assigning them an actual value.

There are two types of statistics that are non-deterministically added to $\mathcal{S}$ as the result of an execution:

- For each RA expression r from $\mathcal{R}_p$ where r does not have a statistics collection operator Σ at the top of the plan, $c(r)$ is added to $\mathcal{S}$.
- For each expression r with a statistics collection operator at the top, an object count for r is added to $\mathcal{S}$. Also, a distinct value count is added to $\mathcal{S}$, computed over each "useful" expression over r. This is described in more detail subsequently.

[1]Note that if a plan has a statistics collection operator at its top, this is removed before the plan is added into $\mathcal{R}_e$, as it is assumed that when an expression is executed and statistics are collected, the result of executing the expression is materialized before statistics collection.

Statistical model: distinct value counts and set sizes. We use a simple model where the size of a join is assumed to depend (deterministically) on the number of distinct values for the joined function and the size of the sets being joined. The number of objects returned from a selection predicate is assumed to depend (deterministically) on the number of distinct values for the function(s) appearing in the selection clause and the size of the input set. For example, consider an RA expression of the form

$$r = (r_1 \bowtie_{\mathcal{F}_1(r_1)=\mathcal{F}_2(r_2)} r_2) \tag{1}$$

Define the number of objects returned by r_1 and r_2 as $c(r_1)$ and $c(r_2)$, respectively. Define $d(\mathcal{F}_1, r_1|_{r_2})$ to be the number of distinct values for $\mathcal{F}_1(r_1)$ *computed with respect to the join with* r_2. Define $d(\mathcal{F}_2, r_2|_{r_1})$ similarly. Then the size of the join is assumed to be:

$$c(r) = \frac{c(r_1)c(r_2)}{\max(d(\mathcal{F}_1, r_1|_{r_2}), d(\mathcal{F}_2, r_2|_{r_1}))} \tag{2}$$

It may seem curious to the reader that we have chosen to define a number of distinct values for an RA expression "with respect to" a particular join. After all, the number of distinct values for $\mathcal{F}_1(r_1)$ is the same, regardless of which RA expression r_1 is being joined with. However, our *expectation* as to the number of distinct values may be different, depending upon what expression we are joining with. For example, we know that in practice, foreign key joins are common. If the join in question happens to be a foreign key join from r_1 to r_2, then the number of distinct values for $\mathcal{F}_1$ will be bounded by $c(r_2)$. Hence, in a WHERE clause like:

WHERE $\mathcal{F}_1$(R) = $\mathcal{F}_2$(S) AND $\mathcal{F}_1$(R) = $\mathcal{F}_3$(T)

we may have a different expectation as to the number of distinct values for $\mathcal{F}_1$(R), depending on whether we are computing R $\bowtie$ S or R $\bowtie$ T or R $\bowtie$ (S $\bowtie$ T). We could attempt to somehow reconcile these different expectations into a single value, but instead we take the simple approach of treating these two distinct value counts as being *different* quantities: $d(\mathcal{F}_1, \mathsf{R}|_{\mathsf{S}})$, $d(\mathcal{F}_1, \mathsf{R}|_{\mathsf{T}})$ and $d(\mathcal{F}_1, \mathsf{R}|_{\mathsf{S}\bowtie\mathsf{T}})$.

In our MDP, the expectation as to the number of distinct values for the expression $\mathcal{F}_1(r_1)$ computed with respect to a join like $r_1 \bowtie_{\mathcal{F}_1(r_1)=\mathcal{F}_2(r_2)} r_2$ is quantified via a *prior distribution* over $d(\mathcal{F}, r_1|_{r_2})$, taking the form:

$$f(d(\mathcal{F}_1, r_1|_{r_2}) \mid c(r_1), c(r_2))$$

Note that the distribution of the number of distinct values for $\mathcal{F}_1(r_1)$ with respect to a join with r_2 takes two parameters:

$c(r_1)$ and $c(r_2)$ (the number of tuples returned by r_1 and r_2, respectively). Why parameterize the distribution in this way? Our goal is to allow a reasonable model for the number of distinct values for $\mathcal{F}_1(r_1)$ in the join $r_1 \bowtie_{\mathcal{F}_1(r_1)=\mathcal{F}_2(r_2)} r_2$, and any model for the number of distinct values should probably take into account at least one of $c(r_1)$ or $c(r_2)$. If $\mathcal{F}_1(r_1)$ is a key for r_1, the number of distinct values for this expression is $c(r_1)$. If it is a foreign key referencing $\mathcal{F}_2(r_2)$, the number of distinct values is bounded by $c(r_2)$. The number of distinct values for $\mathcal{F}_1(r_1)$ cannot exceed $c(r_1)$. We will examine precise choices for this distribution later in the paper, but it is clear that we should allow the distribution to be parameterized on $c(r_1)$ and $c(r_2)$.

Finally, we note that distinct value counts for predicates of the form:

WHERE $\mathcal{F}$(R) = 12

are handled similarly, with a prior taking the form:

$$f(d(\mathcal{F}, \mathsf{R}) \mid c(\mathsf{R})).$$

Here, we parameterize the prior on the number of distinct values for $\mathcal{F}(r)$ on $c(r)$, as the number of distinct values is upper-bounded by $c(r)$. To estimate the selectivity of such a predicate, we use the classical formula:

$$c(\mathcal{F}(\mathsf{R}) \ = \ 12) = \frac{1}{d(\mathcal{F}, \mathsf{R})}.$$

Updating the set of statistics. Given such a prior, the question is: how to update the set of statistics $\mathcal{S}$ to facilitate a state transition? It is not always as simple as just sampling from the prior, since the state transition may be complicated, involving the execution of a non-trivial plan. For example, imagine the MDP is in the following state when the planned RA expressions are executed: $\mathcal{R}_p = \{((\mathsf{R} \bowtie \mathsf{T}) \bowtie \mathsf{S})\}$, $\mathcal{R}_e = \{\mathsf{R}, \mathsf{S}, \mathsf{T}\}$, and $\mathcal{S} = \{c(\mathsf{R}) = 10^6, c(\mathsf{S}) = 10^4, c(\mathsf{T}) = 10^4\}$. In this case, we must add the count $c((\mathsf{R} \bowtie \mathsf{T}) \bowtie \mathsf{S})$ to $\mathcal{S}$. This quantity depends on the distinct value counts associated with the join predicate linking $(\mathsf{R} \bowtie \mathsf{T})$ to S, as well as the sizes of $(\mathsf{R} \bowtie \mathsf{T})$ and S. The distinct value counts for the join predicate can be sampled from the prior provided as input to the MDP, and $c(\mathsf{S})$ is already in $\mathcal{S}$. However, the size of $(\mathsf{R} \bowtie \mathsf{T})$ is not in $\mathcal{S}$. Thus, this needs to be recursively generated before we can add $c((\mathsf{R} \bowtie \mathsf{T}) \bowtie \mathsf{S})$ to $\mathcal{S}$.

The algorithm to recursively generate $c(r)$ for an RA expression of the form $r = (r_1 \bowtie_{\mathcal{F}_1(r_1)=\mathcal{F}_2(r_2)} r_2)$ in $\mathcal{R}_p$ is as follows:

(1) If the count $c(r)$ is already in $\mathcal{S}$, return.
(2) Otherwise, if the number of objects $c(r_1)$ and $c(r_2)$ returned by r_1 or r_2, respectively, are not in $\mathcal{S}$, recursively generate them.
(3) If $d_1 = d(\mathcal{F}_1, r_1|_{r_2})$ is not in $\mathcal{S}$, then sample $d_1 \sim f(d(\mathcal{F}_1, r_1|_{r_2})|c(r_1), c(r_2))$.
(4) Likewise, if $d_2 = d(\mathcal{F}_2, r_2|_{r_1})$ is not in $\mathcal{S}$, then sample $d_2 \sim f(d(\mathcal{F}_2, r_2|_{r_1})|c(r_2), c(r_1))$.
(5) Lastly, compute $c(r)$ as a function of $c(r_1)$, $c(r_2)$, d_1, and d_2 and add to $\mathcal{S}$.

In the case that the RA expression r is topped with the statistics collection operator Σ, we first generate $c(r)$ as described above, and then sample a value for $d(\mathcal{F}, r|_s)$ from $f(d(\mathcal{F}, r|_s)|c(r), c(s))$ for each "useful" RA expression s. Here a "useful" expression s is one that could possibly be joined with r. For example, reconsider the WHERE clause with predicate $\mathcal{F}_1(\mathsf{R}) = \mathcal{F}_2(\mathsf{S})$ AND $\mathcal{F}_1(\mathsf{R}) = \mathcal{F}_3(\mathsf{T})$. With respect to $d(\mathcal{F}, \mathsf{R}|_s)$, s would be "useful" for $s = \mathsf{S}$ and $s = \mathsf{T}$ and $s = (\mathsf{S} \bowtie \mathsf{T})$.

4.4 Rewards

The reward function maps each underlying state-action-state triple onto an immediate reward. In our case, it is more intuitive to consider cost, rather than reward. The *cost* of a transition from state S_1 to state S_2 is the total number of objects that were processed in order to execute and materialize all of the RA expressions found in $\mathcal{R}_p$ in S_1. We could use a more sophisticated cost function (especially if Monsoon was used to perform simultaneous logical and physical optimization; see the Conclusion section of the paper), but we find that object counts work well for logical optimization.

More formally, to cost the state transition from S_1 to S_2, consider $\mathcal{R}_p$ and $\mathcal{R}_e$ from S_1, as well as the set of statistics $\mathcal{S}$ in S_2. For an RA expression that is not topped by the statistics collection operator Σ, compute cost(r) as:

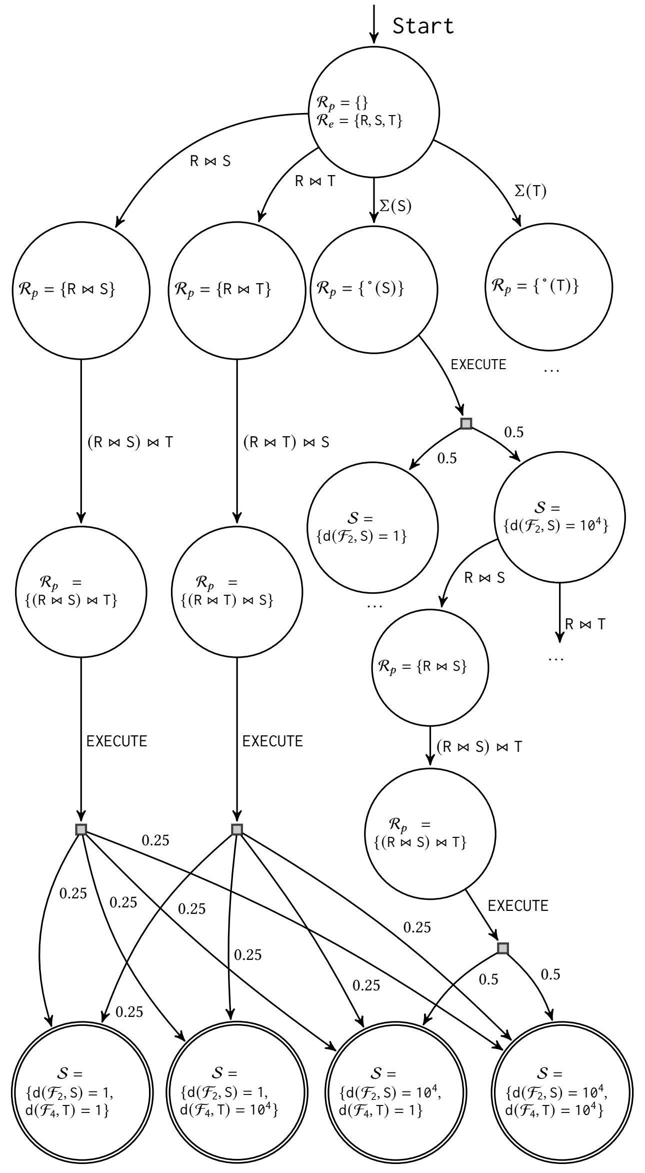
- If r in $\mathcal{R}_e$, simply return $c(r)$ from $\mathcal{S}$.
- Otherwise, we know that r must take the form $(r_1 \bowtie r_2)$. So return $c(r)$ + cost(r_1) + cost(r_2). Note that either cost(r_1) or cost(r_2) (or both) may need to be evaluated recursively.

Finally, the cost for an RA expression that is topped by the statistics collection operator Σ, cost$(\Sigma(r)) = c(r)+$ cost(r). Intuitively, we are assuming that statistics collection requires another pass through the data.

Then, the overall reward of the state transition is computed as the negation of the cost of each of the individual RA expressions that were computed in $\mathcal{R}_p$:

$$\sum_{r \in \mathcal{R}_p} -\text{cost}(r)$$

Given this MDP, the "best" query optimizer is the one that, given a state, always makes the same choice as the optimal policy π^*. That is, it always makes the choice that maximizes the long-term cumulative reward (or minimizes the long-term cumulative cost).



Start
$\mathcal{R}_p$ = {}
$\mathcal{R}_e$ = {R, S, T}
R ⋈ S
R ⋈ T
Σ(S)
Σ(T)
$\mathcal{R}_p$ = {R ⋈ S}
$\mathcal{R}_p$ = {R ⋈ T}
$\mathcal{R}_p$ = {Σ(S)}
$\mathcal{R}_p$ = {Σ(T)}
EXECUTE
0.5
0.5
(R ⋈ S) ⋈ T
(R ⋈ T) ⋈ S
$\mathcal{S}$ = {d($\mathcal{F}_2$, S) = 1}
$\mathcal{S}$ = {d($\mathcal{F}_2$, S) = 10^4}
R ⋈ S
R ⋈ T
$\mathcal{R}_p$ = {(R ⋈ S) ⋈ T}
$\mathcal{R}_p$ = {(R ⋈ T) ⋈ S}
$\mathcal{R}_p$ = {R ⋈ S}
(R ⋈ S) ⋈ T
$\mathcal{R}_p$ = {(R ⋈ S) ⋈ T}
EXECUTE
EXECUTE
EXECUTE
0.25
0.25
0.25
0.25
0.25
0.25
0.25
0.25
0.5
0.5
$\mathcal{S}$ = {d($\mathcal{F}_2$, S) = 1, d($\mathcal{F}_4$, T) = 1}
$\mathcal{S}$ = {d($\mathcal{F}_2$, S) = 1, d($\mathcal{F}_4$, T) = 10^4}
$\mathcal{S}$ = {d($\mathcal{F}_2$, S) = 10^4, d($\mathcal{F}_4$, T) = 1}
$\mathcal{S}$ = {d($\mathcal{F}_2$, S) = 10^4, d($\mathcal{F}_4$, T) = 10^4}

Figure 1: A pictorial example of an MDP.

4.5 Example MDP

We now revisit the query we discussed in Section 2.3. Figure 1 shows the (partial) MDP corresponding to this optimization problem. Note that to keep the figure readable, we have left out states, as well as some details.

We begin in a start state, where $\mathcal{R}_p$ is empty and $\mathcal{R}_e$ contains {R, S, T}. At this point, we have six options (though only four are shown); we can add any two-way join to $\mathcal{R}_p$, or we can add any of the expressions from $\mathcal{R}_e$ to $\mathcal{R}_p$, topped with a statistics collection operator.

Imagine that we add Σ(S) to $\mathcal{R}_p$. Now, we can add additional RA expressions to $\mathcal{R}_p$, or we can choose to EXECUTE the plan in $\mathcal{R}_p$. In this case, according to the prior in Section 2.3, there is a 50% chance that the number of distinct values for $\mathcal{F}_2$ evaluated over Σ(S) is 1, and a 50% chance it is 10^4.

Imagine that it is 10^4. We can choose to add (R ⋈ S) ⋈ T to $\mathcal{R}_p$, through two steps, and then EXECUTE. In this case, depending upon the statistic $d(\mathcal{F}_4, \mathsf{T})$, there are two final outcomes, either of which has a cost of one million (since one million intermediate objects were produced).

5 IMPLEMENTATION

5.1 Solving the MDP via MCTS

There are many methods to solve for π^*, the policy that maximizes the expected reward. Our query optimizer uses Monte-Carlo tree search (MCTS) [9, 27], a popular *online* planner, as its MDP solver (online means that the planner attempts to compute the optimal policy from a given state only after entering into the state). MCTS combines best-first graph traversal and Monte-Carlo evaluation of the reward associated with a particular decision. MCTS uses a simulator for the MDP to generate a sequence of state-action pairs until eventually, a goal state is reached. This is called a *rollout* or an *iteration*. Since the incurred cost (or reward) for each rollout is well-defined, the cost for a state-action pair, (s, a) can be approximated by averaging the rewards of many such rollouts. Paired with selection algorithms [27, 39] to manage the exploration/exploitation trade-off, MCTS can guarantee asymptotic convergence to the optimal policy.

To find a near optimal action from a state, MCTS constructs a search tree, starting from the current state as root. This tree incrementally grows into the direction of the most promising actions, which are determined by the rewards of the Monte-Carlo rollouts starting with these actions. MCTS works by repeating the following steps until a predefined number of iterations (or rollouts) are exhausted:

(1) Selection: Starting from the root node, a selection policy is recursively applied to traverse the tree until an *expandable* node is reached. A node is *expandable* if it is a nonterminal state and has unvisited children.
(2) Expansion: One child node is added to the selected node to expand the tree, according to the actions.
(3) Simulation: A simulation or rollout is run from the new node according to a predefined policy until a terminal or goal state is reached.
(4) Backpropagation: The cumulative reward of the rollout is back-propagated to update the value estimates of all the state-action pairs.

As soon as the predefined number of iterations are reached, the search terminates and the select the action with the highest cumulative reward is selected. Subsequently, this action is performed in the "real world", a state transition is observed, and planning begins again. Note that in our MDP, only the execution of the set of RA expressions in $\mathcal{R}_p$ has an effect in the real world; all other actions deterministically change the set of expressions in $\mathcal{R}_p$.

Balancing exploration and exploitation. To implement MCTS, a choice needs to be made regarding the selection strategy, that can balance between exploitation of actions with very high values and exploration of actions with uncertain values. In our implementation, we try out two different, state of the art selection strategies.

The first is *upper confidence bound for trees* (UCT) [27]. UCT maintains two variables per state-action pair: a counter depicting the number of times it has been visited, and the average normalized reward (normalized to the range $[0, 1]$) for this state-action pair. UCT always selects the child node c that maximizes $r_c + w \times \sqrt{\log(v_p)/v_c}$, where is r_c is the average cumulative reward for the child node, v_p and v_c are the visited counters for the parent and child node respectively, and w is a domain-specific weight factor (in our implementation, we choose, $w = \sqrt{2}$). The first term allows for exploitation, as it promotes children with high reward. The second allows exploration and promotes the children visited the least.

The second is ϵ-*greedy* [39, 40]. In an ϵ-greedy strategy, the best action is selected with probability $(1 - \epsilon)$, while a random action is selected with probability ϵ. The initial value is set to $\epsilon = 1$ (promoting exploration), which is then adaptively decreased [40] with progress in number of iterations to promote exploitation. A lower threshold is used ($\epsilon = 0.1$) to prevent purely exploitative behavior.

5.2 Designing Priors

Our query-planning MDP fundamentally depends on the prior $f(d(\mathcal{F}, r|_s)|c(r), c(s))$. The wrong prior can result in the optimizer making very poor choices. In theory, this prior can take any form, and many choices are possible. In this subsection, we design a set of simple candidate priors, that will be evaluated subsequently.

In the closest existing work, least expected-cost optimization [13, 14], priors for key statistics are constructed either from previous query workload information (an objective prior) or by a domain expert (a subjective prior). However, neither seems particularly palatable to us. Queries using UDFs are often executed only once, or when they are executed many times, it is on evolving or brand new data sets [37] and so workload information may be hard to come by. Further, we want our optimizer to applicable "out of the box", with little or no human involvement or domain expertise.

Hence, our tactic is slightly different. We design a set of simple, general-purpose priors or magic distributions [5], tune them on a set of validation queries—where "tuning" means trying different parameter values so that the prior

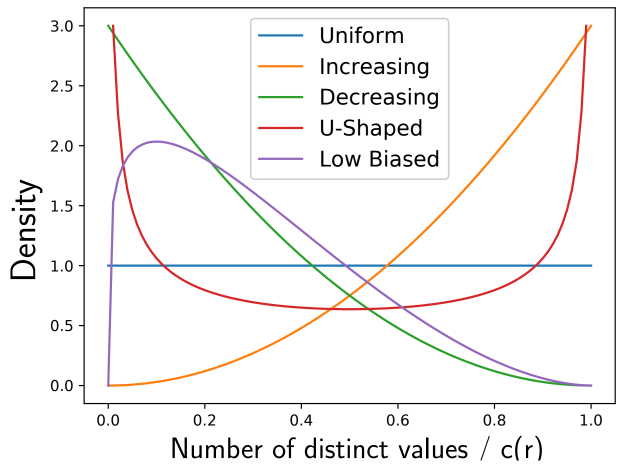


Figure 2: Five of the seven prior distributions.

gives good results during validation—and then experimentally evaluate those priors on a test set. Then, we will choose the best prior based upon those experiments, and evaluate the resulting optimizer under several different scenarios. Note that dataset specific priors [5] can be constructed for Monsoon, using pre-computed (or online samples) collected from the database. Such tailored priors would possibly outperform a generic prior or a "magic distribution".

Here are the simple, general-purpose priors we design for $f(d(\mathcal{F}, r|_s)|c(r), c(s))$:

Uniform. This is the simplest prior, and assumes that the number of distinct values is a random value chosen uniformly between 1 and $c(r)$.

Increasing. This is an optimistic prior, in the sense that it assumes that there are usually a large number of distinct values returned by $\mathcal{F}$, which tends to cut down the number of tuples returned by RA expressions that reference $\mathcal{F}$. In particular, we assume that $d(\mathcal{F}, r|_s)$ is produced by first sampling from a $\text{Beta}(3, 1)$ distribution, multiplying the result by $c(r)$, and taking the ceiling of the result. Using this prior assumes queries return few results.

Decreasing. This is a pessimistic prior, assuming that there are a small number of distinct values, tending to lead to very large query results. An optimizer using this prior assumes the worst by replacing the $\text{Beta}(3, 1)$ distribution in the Increasing prior with a $\text{Beta}(1, 3)$ distribution.

U-shaped. This prior assumes that the number of distinct values tends to be either low or high, but not in the middle. Here, we use a $\text{Beta}(.5, .5)$ distribution.

Low-biased. This is similar to the decreasing prior, but we do not want to be *too* pessimistic; we expect that the number of distinct values is not going to be *too* tiny. Here we use a $\text{Beta}(2, 10)$ distribution.

Spike and slab. Here, there is a 80% chance that $d(\mathcal{F}, r|_s)$ is sampled from a uniform prior. However, we know that in practice, foreign key joins are common. So, this prior assumes that there is a 10% chance that the number of distinct values is $c(r)$ (this is a foreign key join from s into r), and a 10% chance that the number of distinct values is $c(s)$ (this is a foreign key join from r into s).

Discrete. A discrete prior consists of a finite number of predefined values [14]. Here, we evaluate a prior that assumes, distinct count is always 10% of the row count, $d(\mathcal{F}, r|_s) = 0.1 \times c(r)$.

The first five priors are plotted in Figure 2.

5.3 Postgres Implementation

The primary application of the ideas in this paper is optimization over predicates that have been partially obscured by the presence of UDFs. Such UDFs are arguably more common in modern Big Data systems (such as Spark) compared to classical relational systems. Still, we decided to prototype the Monsoon optimizer on top of a relational engine (Postgres) rather than a Big Data system. There are two reasons for this. First, as we describe now, Monsoon works together with (rather than in place of) an existing optimizer, and the Postgres optimizer is arguably more mature and robust than the optimizer available in a system such as Spark, hence Postgres is a more attractive target. Second, an alternative approach (SkinnerDB [41, 42]) has also been implemented on top of Postgres, allowing for an apples-to-apples comparison.

One of the benefits of our proposed methodology is that it requires few (or no) changes to an existing system's optimizer and execution engine, and our on-top-of-Postgres implementation of Monsoon is in-keeping with this goal. Monsoon works with the existing Postgres optimizer and execution engine and requires few changes to either.

Given a query, our optimizer begins by using MCTS to modify the set of planned RA expressions, $\mathcal{R}_p$. Once the MCTS requests an `EXECUTE` operation, all of the RA expressions in $\mathcal{R}_p$ are executed, by Postgres, in sequence. The MCTS prescribes a join order for each RA expression that Postgres is not allowed to change, but otherwise, the Postgres optimizer is used without modification to optimize and execute the query (including the choice of a physical plan) however it chooses. If any RA expression in $\mathcal{R}_p$ has a statistics collection operator on top, Postgres' statistics collection facility is used to collect statistics on the result of the RA expression after Postgres has executed the query, and those collected statistics are added to the set of observed statistics $\mathcal{S}$, as well as to the Postgres system catalog to be used during optimization and execution of subsequent RA expressions.

After the first `EXECUTE` operation, our optimizer again performs a sequence of MCTS iterations, until once again MCTS prescribes an `EXECUTE` operation. Again, Postgres is used to execute all of the RA expressions in $\mathcal{R}_p$ and statistics are updated. This process is repeated until query completion.

6 EVALUATION

6.1 Goals and Scope

Our evaluation consists of two sets of experiments.

The goal of the first set of experiments is to determine what effect, if any, the choice of a prior on distinct value counts has on the performance of the optimizer, and to indicate the appropriate prior for use in the optimizer.

The goal of the second set of experiments is to determine whether there is an advantage to using the Monsoon optimizer to optimize and execute queries compared to a set of reasonable alternatives, when cardinality estimates are unavailable due to the presence of UDFs.

6.2 Experimental Design

6.2.1 Choice of Prior. The first set of experiments aim to determine the most appropriate prior for use in Monsoon. We use the `TPC-H` benchmark and data generator [3], restricted to only those queries that have a non-trivial join ordering problem (at least three tables). We use a scale-factor 100 TPC-H database, approximately 100GB in size.

We also create three skewed TPC-H benchmark sets [11]. The skewed TPC-H data generator provides a parameter to control the degree of Zipfian skew z. The greater the values of z, the more the skew in the generated data. We test three options; low skew ($z = 1$), high skew ($z = 4$), and mixed (the skew in each column is selected uniformly at random from the range zero through four).

Over these data sets, we evaluate Monsoon using each of the seven priors suggested in Section 5.2.

6.2.2 Comparison with Other Optimizers. The second set of experiments compare Monsoon with a set of reasonable alternatives for optimizing and executing queries in an environment where predicates are partially obscured by UDFs. These options are:

(1) Postgres/full statistics collection on demand. Simply compute a full set of required statistics after the query is issued, but before the query is optimized. To implement this, we use state of the art HyperLogLog(HLL) sketches [22] to estimate distinct value counts before optimization. Only the required statistics for the tables and attributes that participate in the predicates of a query are collected. We subsequently refer to this option as "On Demand".

(2) Postgres/Sampling. Without statistics, one may sample from the base tables before optimization, estimate statistics over the samples, and use those statistics. The closest work

to this is DYNO [26], which proposed sampling based *pilot runs* for statistics collection over UDFs defined over base tables—but did not consider selectivities for UDFs that are applied on results of joins. Designing a solution that considers selectivities for multi-table UDFs is challenging, and to the best of our knowledge, it has not been well-explored in the literature. In our implementation, at runtime we use block-based sampling to sample 2% of each base table, up to a maximum of 200,000 tuples, and use the algorithms of Charikar et. al [8] to estimate distinct value counts from the samples. For multi-table UDFs, we materialize at most one million tuples from the product of the subsamples, applying the multi-table UDF to the materialized tuples to estimate distinct value counts. For efficiency, we use block-based sampling. We refer to this option as "Sampling".

(3) Postgres/Greedy. One option is to build a left-deep query plan using only set sizes, but no other statistical information. We call this the *greedy* optimizer. Starting with the set with smallest size, greedy repeatedly joins the next largest table that does not introduce a cross product (unless necessary), until all tables are exhausted. Joining small sets first may ensure that intermediate results are small. We subsequently refer to this option as "Greedy."

(4) Postgres/Defaults. When no statistics are available, a tactic employed by optimizers is to use default values or to make reasonable guesses, such as assuming that distinct count of an attribute equals 10% of the row count. We evaluate an optimizer that uses default cardinalities and subsequently refer to this strategy as "Defaults". Many data management systems, including Postgres, resort to such ad-hoc estimations or magic constants in the absence of statistics [30].

(5) Postgres/Skinner DB. SkinnerDB [41, 42], eschews classical, cardinality based cost models and thus requires no statistics. SkinnerDB learns an optimal left-deep query plan in an online fashion during the execution of each query. There are different variants that either make the assumption that a set of self-similar queries are repeatedly issued over a database with static schema and data (Skinner-H) or require a custom execution engine (Skinner-C), and try to learn join orders over the workload. Skinner-G makes no such assumptions and is capable of handling specialized queries on ad hoc datasets, with no statistics available beforehand; hence, it is directly applicable to our problem. Since SkinnerDB relies on online query processing—not supported by modern relational engines—it runs best when implemented from scratch on top of a specialized engine. However, a Postgres-based implementation is available. [2] We subsequently refer to this option as "SkinnerDB."

(6) Postgres/Monsoon. We evaluate our Postgres-based implementation of the Monsoon optimizer; the implementation was described in Section 5.3 of the paper. We subsequently refer to this option as "Monsoon."

(7) Baseline: relational engine, full stats. As a baseline, we also run Postgres with full statistics collection done offline, and not counted on the optimizer running time. This shows us how well Postgres could do, if it had access to full statistics. We subsequently refer to this option as "Postgres".

For a fair comparison, the different optimizers use only distinct value and tuple counts, when applicable. For example, "On Demand" and "Monsoon" only collect distinct value counts using HLL sketches [22]. PostgreSQL's native statistics collection functionality is not used because it collects more sophisticated statistics like frequencies for heavy-hitters, histograms for non-heavy hitters, etc.

Benchmarks. We evaluate these seven options on three different query optimization benchmarks.

(1) IMDB. This popular query optimization benchmark is based on the `Internet Movie Data Base (IMDB)`, a real-world data-set. The benefit of using such a data set is that it contains many correlations and non-uniform data distributions. Leis et. al. proposed the IMDB Join Order Benchmark by normalizing the IMDB data-set into 21 tables and creating a suite of 113 realistic queries [29, 30]. Since the IMDB database is relatively small (3.9 GB), we create a larger database by making each table five times larger by bootstrap resampling from the underlying table. That is, for a table with n tuples, we create a new version of the table with $5 \times n$ tuples by sampling $5 \times n$ times from the original table, with replacement. The resulting database is 20 GB in size.

Note that the IMDB benchmark (and the OTT benchmark described below) do not contain partially obscured predicates or UDFs. However, if a system is forced to operate without statistics (or collect them at runtime) it is a reasonable proxy for a UDF-heavy workload where statistics are unavailable due to the presence of UDFs.

(2) OTTs. Wu et. al. proposed the correlated Optimizer Torture Tests (OTTs) in [45] – a principled approach to creating corner-case queries that are specifically difficult for cardinality-estimation-based query optimizers.

We created an OTT database [3] by augmenting a standard 100 GB TPC-H database with two additional correlated columns. We also created a suite of 20 artificial queries by following the instructions in Section 5.3 of [45]. The final result of each query is empty. However, in the worst case the optimizer can end up generating billions of intermediate tuples. It is possible to manually identify the join predicate(s)

[2]Skinner-G was generously made available by Immanuel Trummer.

[3]The OTT data generator was obtained from Wentao Wu.

that would produce an empty result. An early evaluation of such an empty join predicate is preferred; hence, as a baseline, we implement the best hand-written left deep query plans for the OTT queries.

(3) UDF. Finally, to evaluate our optimizer on a UDF-heavy benchmark, we created a set of UDFs including few multi-table UDFs that operate over strings, and designed a set of 25 queries [4] that exclusively use these UDFs for join and selection predicates. 15 of the queries were selected from the IMDB join benchmark, and are directly translated from that benchmark. The remaining 10 were queries over the TPC-H benchmark that were designed to present a difficult join order problem. The UDFs used are relatively inexpensive. We leave the study of expensive predicates [12, 21] as an important direction for future work.

Note that computing statistics over multi-table UDFs require materializing cross-products or joins. The "On Demand" option becomes prohibitively expensive. The "Postgres" option with all statistics (including multi-table UDFs) known beforehand is also unrealistic. Hence, these two options have been dropped for this benchmark.

Timeouts. We expect the various optimizers to periodically fail to produce a reasonable plan, and may take hours or days to execute (or crash the system). We set a timeout of 20 minutes. When a timeout occurs, we do not report an average time on a benchmark.

Setup. Experiments are run on an Amazon EC2 c5d.9xlarge instance which has 36 vCPU cores, 72 GB RAM and 900 GB of SSD hard disks running with Ubuntu Linux 14.04 LTS. We use Postgres 9.5.16. We follow the best practices mentioned in the literature [30] for benchmarking optimizers.

6.3 Results

Table 2 gives the results of executing the TPC-H benchmark with the Monsoon optimizer, using each of the seven prior distributions from Section 5.2. Since the "Spike and Slab" prior seems to consistently be one of the top choices, we choose this prior for the remainder of the experiments.

Tables 3 and 6 give the mean, median, and max query execution times over the IMDB and OTT benchmarks, respectively. In Table 5, we give the execution times for the 20 longest-running queries on the IMDB benchmark. In Table 4, we indicate how often each method gives a running time less than 90% or greater than 110% of Postgres on IMDB.

Table 7 gives similar results for the UDF benchmark. Figure 3 gives similar results in a plot, where the time of each of the 25 benchmark queries is depicted for each of the four optimization options. The queries are sorted from low to high time required to execute the queries using Monsoon.

[4]https://bitbucket.org/sikdarsourav/monsoonqueries

Implementation	TPC-H	Low	High	Mixed
Uniform	430.68	370.07	386.36	N/A
Increasing	N/A	442.61	398.43	376.04
Decreasing	381.25	416.97	315.02	N/A
U-Shaped	N/A	N/A	339.42	402.54
Low Biased	405.44	475.29	320.96	427.78
Spike and Slab	378.46	374.94	348.60	396.43
Discrete	408.67	422.71	326.18	429.03

Table 2: Average query execution time (in seconds) for different priors on TPC-H benchmark queries. "Low", "High", "Mixed" refer to different degrees of skew. "N/A" means that one of the queries timed out, so an average could not be computed.

Implementation	TO	Mean	Median	Max
Postgres	0	151.26	33.30	1004.72
Defaults	2	N/A	37.24	TO
Greedy	7	N/A	55.02	TO
Monsoon	0	164.83	49.29	964.41
On Demand	0	213.99	101.58	1112.35
Sampling	1	N/A	35.27	TO
SkinnerDB	35	N/A	1200	TO

Table 3: Performance (seconds) on the 20GB IMDB database. "TO" means "timeout".

Impl.	< 0.9	[0.9,1.1)	>1.1
Defaults	20%	33.33%	46.67%
Greedy	16.67%	23.33%	60%
Monsoon	21.67%	25%	53.33%
On Demand	8.33%	8.33%	83.34%
Sampling	18.33%	20%	61.67%
SkinnerDB	0%	1.67%	98.33%

Table 4: Relative performance of different implementations compared to baseline Postgres (with full statistics) on the 20GB IMDB database.

Finally, in Table 8 we show the breakdown of the average time taken by different components of the Monsoon optimizer: time required for MCTS, time required to execute statistics collection when ordered by the optimizer, and time required to implement the various relational expressions.

6.4 Discussion and Recommendations

Given all of these results, what would we recommend? First off, SkinnerDB does poorly in all of our experiments. This is not an indictment of the SkinnerDB approach. The main problem is that SkinnerDB is not a standalone optimizer; it is a complete data processing system. SkinnerDB relies

Implementation	TO	Mean	Median	Max
Postgres	0	404.36	257.77	1004.73
Defaults	2	N/A	382.83	TO
Greedy	7	N/A	860.92	TO
Monsoon	0	410.43	240.86	964.41
On Demand	0	492.99	470.21	1112.35
Sampling	1	N/A	287.41	TO
SkinnerDB	19	N/A	1200	TO

Table 5: Performance (seconds) of different implementations on the 20 most expensive IMDB Join Order Benchmark queries. "TO" means "timeout".

Implementation	TO	Mean	Median	Max
Hand-written	0	5.73	5.39	8.93
Postgres	4	N/A	164.78	TO
Defaults	8	N/A	167.44	TO
Greedy	8	N/A	5.24	TO
Monsoon	3	N/A	103.35	TO
On Demand	2	N/A	523.41	TO
Sampling	2	N/A	179.49	TO

Table 6: Performance (seconds) on the Optimizer Torture Tests. "TO" means "timeout".

Implementation	TO	Mean	Median	Max
Defaults	3	NA	40.67	TO
Greedy	4	NA	57.96	TO
Monsoon	0	46.37	26.23	195.35
Sampling	0	48.96	24.36	152.77
SkinnerDB	22	NA	1200	TO

Table 7: Performance (seconds) of different implementations on Queries with UDFs. "TO" means "timeout".

Benchmark	MCTS	Σ	Execution
IMDB (20GB)	2.77	2.05	160.01
IMDB-20 (20GB)	3.33	4.05	403.05
OTT	2.14	1.45	245.27
UDF	3.41	4.43	38.53

Table 8: Average time (seconds) taken by different components of the Monsoon optimizer.

on incremental processing, where it partially runs an operation, such as a join order, and makes a decision as to what to do next based upon the partial results. As such, it really needs to be implemented from the ground up, and not on top of an existing database (such as Postgres) that does not directly support incremental processing. As the authors of the SkinnerDB paper show in their paper, a ground-up implementation can be highly performant. But as an optimizer for

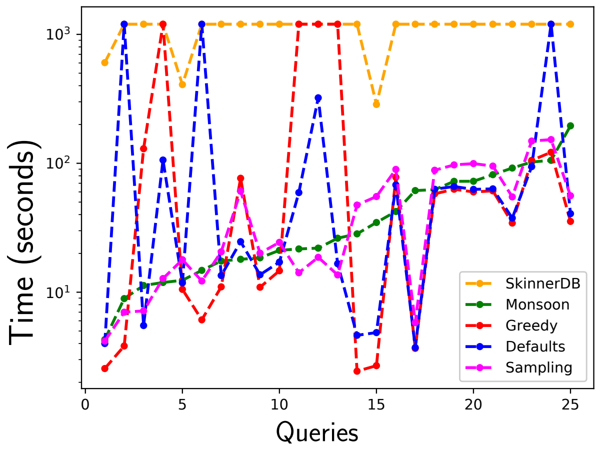


Figure 3: Performance of different implementations on queries with UDFs.

a batch data processing system (such as Postgres or Spark), it may not be the best choice.

On the IMDB queries, "classical" Postgres (with statistics) generally does the best. This is not surprising. But if UDFs are present, there are four options: Defaults, Sampling, Greedy, and Monsoon. In contrast to these four, On-Demand is not a universal option, as it cannot handle multi-table UDFs (without materializing cross-products) and regardless, it is generally dominated by Monsoon and Sampling. On-Demand results in runtimes that are (in terms of the median) 2× slower than Monsoon on the IMDB queries.

Greedy has serious problems with timeouts on all the benchmarks (it fails on eight of the OTT benchmark queries, seven IMDB queries and four of the UDF queries). As such, it is probably not a good choice. This leaves Defaults, Sampling, and Monsoon.

Defaults does surprisingly well. In terms of median time, it does an excellent job. It does best compared to Postgres in terms of matching Postgres performance on IMDB (Table 4). Probably the biggest problem with the Defaults option is that it has issues with timeouts. In particular, it fails on eight of the OTT benchmark queries (it also fails on two IMDB queries and three of the UDF queries). One may argue that the OTT benchmark is somewhat artificial, but we believe that the results call into question the robustness of the Defaults option. This matches intuition: choosing default cardinalities will usually give the right answer, but it will occasionally be very wrong. Since query optimization is most concerned with avoiding bad plans (as opposed to choosing the best plan), we argue against this choice.

Sampling or Monsoon? On IMDB, the two options had similar performance (see Table 4). Monsoon was better on the tougher queries, Sampling on the easier queries. On the

OTT and UDF benchmarks, Monsoon was somewhat faster, though not dominant. For us, the "tie-breaker" is that Monsoon is an universal option. Drawing samples from data can be difficult or impossible, particularly in a Big Data setting where input data are often saved to a huge file in JSON, CSV, or some proprietary interchange format. In such a situation, it is only feasible to process the file sequentially (or, with some difficulty, sequentially in parallel [20]) so sampling will require a full-pass reservoir algorithm [43] before optimization. Under such a circumstance, it is probably a better option to use a combination of On-Demand (using sketches for single-table distinct value counts during a full pass) and sampling (to draw samples from the cross product in the case of multi-table UDFs). But this requires a sampling operator to handle multi-table UDFs. Monsoon does not require a sampling operator, which makes it widely applicable.

7 RELATED WORK

Classical, relational cost-based query optimization [10, 38] is *offline* – the optimizer first performs an exploration among a large space of semantically equivalent plans for a given query, and chooses the plan of least cost, followed by execution. The plan chosen by an offline optimizer can be poor, primarily due to errors in cardinality estimation [24]. To alleviate these problems, *online* optimization, by interleaving query planning with execution, has been a recurrent theme in adaptive query processing [4, 6, 15, 25].

In *least expected cost* optimization [13, 14], parameters which cannot be accurately estimated at query compile time are modeled using prior distributions. The goal is to choose robust plans considering the uncertainty expressed by the priors. Similar approaches have been proposed with the goal of finding *robust* query plans [5, 33].

There has been a flurry of recent work that solves the classical, relational query optimization using reinforcement learning [28, 31, 32, 42]. While ReJOIN [32] and DQ [28] uses deep-reinforcement learning for join ordering for a given cost model, NEO [31] proposes an end-to-end, continuously learning solution without any reliance on predefined, hand-crafted cost model. SkinnerDB [42] tries to learn join orders in an online fashion in conjunction with learnt join orders from historical experience.

Our approach connects to prior works that rely on online statistics collection of some form [7, 26, 47] in big data systems. Query optimizers for big-data systems often assume availability of historical statistics [7, 47] from prior workloads to generate an initial plan. Subsequently, modifications are made to this plan online, based on statistics collected during execution. However, queries with UDFs in big-data processing systems are often run only once [37], making such approaches unbefitting. The closest work to ours is DYNO [26], which proposed sampling based statistics collection for UDFs as a prelude to optimization and execution. In relational query optimization, certain approaches completely eschew the need for prior statistics [16, 42] and we compare our approach against recently proposed SkinnerDB [42].

Under a per-tuple cost model for *fully-obscured* predicates, Chaudhuri and Shim [12] proposed an algorithm for optimal placement of such predicates.

There have been some recent efforts targeted at translating imperative UDF code to declarative SQL; this is known as `UDF algebraization` [17, 18, 35, 36]. However, it is known that not all UDFs can be translated [36]. In some general cases, the code for UDFs may not even be available for semantic analysis (for example, the UDFs may be compiled into a shared library that is dynamically loaded by the data processing system). While semantic analysis, whenever applicable, can alleviate the problem by translating some UDFs, it is not a replacement for a query optimizer. In general, we believe an UDF algebraization framework like Froid [35, 36] can co-exist within the same system as a pre-cursor to a Monsoon-style optimization framework.

8 CONCLUSIONS

We have described an optimizer called Monsoon, which handles relational and Big Data computations having predicates that are partially obscured by the presence of opaque UDFs. Monsoon views the problem of when to collect statistics on such partially-obscured predicates as a Markov decision process (MDP). By solving this MDP, Monsoon decides when it is better to be safe and collect statistics on the number of distinct values returned by a UDF, and when it is better to be bold and simply guess at a reasonable query plan.

There are several avenues for future work. Cost models are often inaccurate, and the MDP used by Monsoon could be extended to handle uncertainty in the correctness of the cost model, as well as uncertainty in statistics. We have only considered the join order problem in this paper; extending to other optimization tasks is left to future work. For example, separating logical and physical optimization can result in less efficient plans. The Markov decision process we have defined could be extended to choose physical operators as well as join orders, with the reward function measuring end-to-end running time (or CPU time, memory usage, network latency, or a combination of these).

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive suggestions. Work presented in this paper has been supported by NSF under grant Nos. 1409543 and 1910803.

REFERENCES

[1] 2017. Introducing Interleaved Execution for Multi-Statement Table Valued Functions. (2017). https://blogs.msdn.microsoft.com/sqlserverstorageengine/2017/04/19/introducing-interleaved-execution-for-multi-statement-table-valued-functions/

[2] 2019. https://www.postgresql.org/docs/9.3/view-pg-stats.html. (2019).

[3] 2019. TPC. 2013. TPC-H Benchmark. http://www.tpc.org/tpch/. (2019). http://www.tpc.org/tpch/

[4] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 261–272. https://doi.org/10.1145/342009.335420

[5] Brian Babcock and Surajit Chaudhuri. 2005. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. ACM, New York, NY, USA, 119–130. https://doi.org/10.1145/1066157.1066172

[6] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive Re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/1066157.1066171

[7] Nicolas Bruno, Sapna Jain, and Jingren Zhou. 2013. Continuous Cloud-scale Query Optimization and Processing. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 961–972. https://doi.org/10.14778/2536222.2536223

[8] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 2000. Towards Estimation Error Guarantees for Distinct Values. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/335168.335230

[9] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. 2008. Monte-Carlo Tree Search: A New Framework for Game AI. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, October 22-24, 2008, Stanford, California, USA.* http://www.aaai.org/Library/AIIDE/2008/aiide08-036.php

[10] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '98)*. ACM, New York, NY, USA, 34–43. https://doi.org/10.1145/275487.275492

[11] Surajit Chaudhuri and Vivek Narasayya. [n.d.]. Program for Generating Skewed Data Distributions for TPC-D. https://www.microsoft.com/en-us/download/confirmation.aspx?id=52430. ([n. d.]). https://www.microsoft.com/en-us/download/confirmation.aspx?id=52430

[12] Surajit Chaudhuri and Kyuseok Shim. 1999. Optimization of Queries with User-defined Predicates. *ACM Trans. Database Syst.* 24, 2 (June 1999), 177–228. https://doi.org/10.1145/320248.320249

[13] Francis Chu, Joseph Halpern, and Johannes Gehrke. 2002. Least Expected Cost Query Optimization: What Can We Expect?. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02)*. ACM, New York, NY, USA, 293–302. https://doi.org/10.1145/543613.543651

[14] Francis Chu, Joseph Y. Halpern, and Praveen Seshadri. 1999. Least Expected Cost Query Optimization: An Exercise in Utility. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '99)*. ACM, New York, NY, USA, 138–147. https://doi.org/10.1145/303976.303990

[15] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Found. Trends databases* 1, 1 (Jan. 2007), 1–140. https://doi.org/10.1561/1900000001

[16] Anshuman Dutt and Jayant R. Haritsa. 2014. Plan Bouquets: Query Processing Without Selectivity Estimation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 1039–1050. https://doi.org/10.1145/2588555.2588566

[17] K. Venkatesh Emani, Tejas Deshpande, Karthik Ramachandra, and S. Sudarshan. 2017. DBridge: Translating Imperative Code to SQL. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 1663–1666. https://doi.org/10.1145/3035918.3058747

[18] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1781–1796. https://doi.org/10.1145/2882903.2882926

[19] Hector Garcia-Molina, Jennifer Widom, and Jeffrey D. Ullman. 1999. *Database System Implementation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[20] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative Distributed CSV Data Parsing for Big Data Analytics. In *Proceedings of the 2019 International Conference on Management of Data.* ACM, 883–899.

[21] Joseph M. Hellerstein. 1998. Optimization Techniques for Queries with Expensive Methods. *ACM Trans. Database Syst.* 23, 2 (June 1998), 113–157. https://doi.org/10.1145/292481.277627

[22] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. ACM, New York, NY, USA, 683–692. https://doi.org/10.1145/2452376.2452456

[23] R. A. Howard. 1960. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA.

[24] Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the Propagation of Errors in the Size of Join Results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data (SIGMOD '91)*. ACM, New York, NY, USA, 268–277. https://doi.org/10.1145/115790.115835

[25] Navin Kabra and David J. DeWitt. 1998. Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. *SIGMOD Rec.* 27, 2 (June 1998), 106–117. https://doi.org/10.1145/276305.276315

[26] Konstantinos Karanasos, Andrey Balmin, Marcel Kutsch, Fatma Ozcan, Vuk Ercegovac, Chunyang Xia, and Jesse Jackson. 2014. Dynamically Optimizing Queries over Large Scale Data Platforms. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 943–954. https://doi.org/10.1145/2588555.2610531

[27] Levente Kocsis and Csaba Szepesvári. 2006. Bandit Based Monte-carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning (ECML'06)*. Springer-Verlag, Berlin, Heidelberg, 282–293. https://doi.org/10.1007/11871842_29

[28] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018). arXiv:1808.03196 http://arxiv.org/abs/1808.03196

[29] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[30] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal* 27, 5 (01 Oct 2018), 643–668. https://doi.org/10.1007/s00778-017-0480-7

[31] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *CoRR* abs/1904.03711 (2019). arXiv:1904.03711 https://arxiv.org/abs/1904.03711

[32] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM'18)*. ACM, New York, NY, USA, Article 3, 4 pages. https://doi.org/10.1145/3211954.3211957

[33] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust Query Processing Through Progressive Optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. ACM, New York, NY, USA, 659–670. https://doi.org/10.1145/1007568.1007642

[34] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st ed.). John Wiley & Sons, Inc., New York, NY, USA.

[35] Karthik Ramachandra and Kwanghyun Park. 2019. BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1810–1813. https://doi.org/10.14778/3352063.3352072

[36] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 432–444. https://doi.org/10.1145/3186728.3164140

[37] Astrid Rheinländer, Ulf Leser, and Goetz Graefe. 2017. Optimization of Complex Dataflows with User-Defined Functions. *ACM Comput. Surv.* 50, 3, Article 38 (May 2017), 39 pages. https://doi.org/10.1145/3078752

[38] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[39] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement learning - an introduction.* MIT Press. http://www.worldcat.org/oclc/37293240

[40] Michel Tokic. 2010. Adaptive epsilon-Greedy Exploration in Reinforcement Learning Based on Value Difference. In *KI 2010: Advances in Artificial Intelligence, 33rd Annual German Conference on AI, Karlsruhe, Germany, September 21-24, 2010. Proceedings.* 203–210. https://doi.org/10.1007/978-3-642-16111-7_23

[41] Immanuel Trummer, Samuel Moseley, Deepak Maram, Saehan Jo, and Joseph Antonakakis. 2018. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 2074–2077. https://doi.org/10.14778/3229863.3236263

[42] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. *CoRR* abs/1901.05152 (2019). arXiv:1901.05152 http://arxiv.org/abs/1901.05152

[43] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.

[44] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. 1990. A Linear-time Probabilistic Counting Algorithm for Database Applications. *ACM Trans. Database Syst.* 15, 2 (June 1990), 208–229. https://doi.org/10.1145/78922.78925

[45] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. 2016. Sampling-Based Query Re-Optimization. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1721–1736. https://doi.org/10.1145/2882903.2882914

[46] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[47] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: Parallel Databases Meet MapReduce. *The VLDB Journal* 21, 5 (Oct. 2012), 611–636. https://doi.org/10.1007/s00778-012-0280-z

[48] Jia Zou, R. Matthew Barnett, Tania Lorido-Botran, Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian, Binhang Yuan, and Chris Jermaine. 2018. PlinyCompute: A Platform for High-Performance, Distributed, Data-Intensive Tool Development. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 1189–1204. https://doi.org/10.1145/3183713.3196933