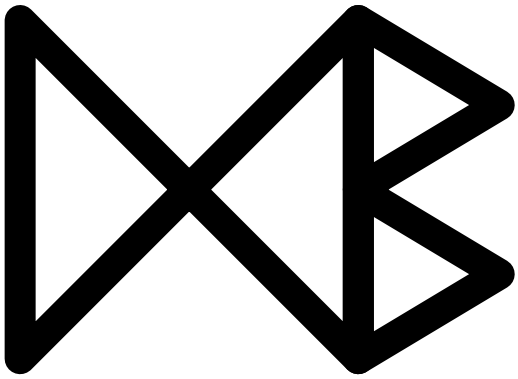


A Reinforcement Learning Approach for Adaptive Query Processing

Kostas Tzoumas, Timos Sellis and Christian S. Jensen

June 27, 2008

TR-22

A DB Technical Report

Title	A Reinforcement Learning Approach for Adaptive Query Processing
	Copyright © 2008 Kostas Tzoumas, Timos Sellis and Christian S. Jensen . All rights reserved.
Author(s)	Kostas Tzoumas, Timos Sellis and Christian S. Jensen
Publication History	June 2008. A DB Technical Report

For additional information, see the DB TECH REPORTS homepage: ⟨dbtr.cs.aau.dk⟩.

Any software made available via DB TECH REPORTS *is provided "as is" and without any express or implied warranties, including, without limitation, the implied warranty of merchantability and fitness for a particular purpose.*

The DB TECH REPORTS icon is made from two letters in an early version of the Rune alphabet, which was used by the Vikings, among others. Runes have angular shapes and lack horizontal lines because the primary storage medium was wood, although they may also be found on jewelry, tools, and weapons. Runes were perceived as having magic, hidden powers. The first letter in the logo is "Dagaz," the rune for day or daylight and the phonetic equivalent of "d." Its meanings include happiness, activity, and satisfaction. The second letter is "Berkano," which is associated with the birch tree. Its divinatory meanings include health, new beginnings, growth, plenty, and clearance. It is associated with Idun, goddess of Spring, and with fertility. It is the phonetic equivalent of "b."

Abstract

In adaptive query processing, query plans are improved at runtime by means of feedback. In the very flexible approach based on so-called eddies, query execution is treated as a process of routing tuples to the query operators that combine to compute a query. This makes it possible to alter query plans at the granularity of tuples. Further, the complex task of searching the query plan space for a suitable plan now resides in the routing policies used. These policies must adapt to the changing execution environment and must converge at a near-optimal plan when the environment stabilizes.

This paper advances adaptive query processing in two respects. First, it proposes a general framework for the routing problem that may serve the same role for adaptive query processing as does the framework of search in query plan space for conventional query processing. It thus offers an improved foundation for research in adaptive query processing. The framework leverages reinforcement learning theory and formalizes a tuple routing policy as a mapping from a state space to an action space, capturing query semantics as well as routing constraints. In effect, the framework transforms query optimization from a search problem in query plan space to an unsupervised learning problem with quantitative rewards that is tightly coupled with the query execution. The framework covers selection queries as well as joins that use all proposed join execution mechanisms (SHJs, SteMs, STAIRs). Second, in addition to showing how existing routing policies can fit into the framework, the paper demonstrates new routing policies that build on advances in reinforcement learning. By means of empirical studies, it is shown that the proposed policies embody the desired adaptivity and convergence characteristics, and that they are capable of clearly outperforming existing policies.

1 Introduction

In conventional database management systems, a query optimizer generates appropriate query execution plans during a separate query optimization phase; having been generated, a plan is considered static. Query optimizers rely on cost models and statistical information for their functioning. This arrangement falls short in two respects.

First, the available statistics are often unreliable and present only a coarse approximation of the underlying database. This applies especially to data integration queries [27]. In some applications, it is effectively impossible to gather appropriate statistics, e.g., in data collection from autonomous data streams [5]. Further, traditional uniformity assumptions may be inappropriate even in standard applications faced with highly correlated data [29]. Statistical model errors propagate exponentially while estimating the cost of a join plan [25, 26]. Finally, although statistics relating to alphanumeric data are well understood, data of complex types raise the complexity of the problem [30].

Second, existing, simple cost models are unsuitable for highly dynamic environments, such as federated databases and settings in which data travels through a network with unknown topology. Such environments often exhibit unpredictable and bursty behavior. More complex cost models can also deteriorate when used in highly dynamic environments [5].

In adaptive query processing (AQP) [9, 12], the query processor receives run-time feedback that enables query plans to be changed while the query is being executed. In perhaps the most flexible approach, the so-called eddy operator acts as a tuple router, intercepting all incoming and outgoing tuples among the operators in the dataflow and making routing decisions about these on the fly [1]. To achieve per-tuple adaptivity, operators must be fully or near-fully pipelined. Two basic approaches exist for join processing that essentially reflect the execution time versus storage space tradeoff.

Using a flavor of the Symmetric Hash Join (SHJ) algorithm, the joins can be fully pipelined with the drawback of increased memory consumption, as every intermediate result has to be stored (history-dependent execution). Using an n-way SHJ, all intermediate results are recomputed (history-independent execution). The former join algorithms also have their unary-operator counterparts. The SteM [18] module allows an eddy to execute fully pipelined, history-independent join plans and control not only the query

plan, but also the join algorithms themselves. A different approach is adopted in the STAIRs model [8], where a history-dependent join execution is controlled by a migration operation that eliminates the impact of history on future routing decisions.

The most critical component in the eddy-based approach is the routing policy. With little or no knowledge about the environment, a policy should be able to adapt to changes in the environment and also converge to a near-optimal solution if the environment stabilizes. This paper addresses the routing problem by modeling query execution with eddies as a reinforcement learning (RL) process [22] and then by leveraging several techniques from this field for proposing new routing policies.

The paper's contributions are threefold. First, it offers a mathematical framework for query processing with eddies based on reinforcement learning, that is capable of capturing the complexity of selection and join processing. Conjunctive selection queries (the selection ordering problem) and natural join queries (the join order problem) are addressed. For the latter, we enable the framework to accommodate the different join operators as well as routing constraints that have been proposed [1, 7, 8, 18]. We also discuss a solution to the "burden of routing history" problem [8].

Second, we break down the execution into two orthogonal phases, namely the update phase that expresses the kind of information about the environment being stored, and the improvement phase that constitutes an approach to the exploration-exploitation tradeoff. We introduce the so-called Q values as the only meta-data to be stored, and we show that these can capture the history-dependent execution of joins using various update policies. We also show how existing routing policies can be incorporated into the framework as special cases of improvement policies.

Third, we propose a number of routing policies that leverage well-known RL algorithms and that span a wide spectrum of approaches in the exploration-exploitation tradeoff space. All these algorithms have asymptotic convergence proofs if the problem environment stabilizes and adapt well to environmental fluctuations. The overall goal is to make it possible to move towards globally optimal or near-optimal query plans with only local decisions, so that the per-tuple adaptivity is not sacrificed. We experimentally evaluate our techniques in terms of learning speed, adaptivity, and performance, and compare them to previous solutions. The results show that our algorithms can clearly outperform proposed solutions in a variety of settings.

The rest of the paper is organized as follows. Section 2 reviews eddy-based query processing and formalizes the semantics of operators and various sets of routing constraints. Section 3 covers background material from the reinforcement learning theory. Section 4 presents our query execution framework based on reinforcement learning in different settings and provides pseudocode that shows how learning and execution can be interleaved. Then, Section 5 presents our implementation and experimental results. Section 6 covers related work, and we conclude and present directions for future research in Section 7.

2 Query Execution with Eddies

We briefly describe the operators that take part in the execution of selection and natural join queries, and we describe the different sets of routing constraints that can be applied to join processing and reflect a spectrum of join plan spaces.

2.1 The Eddy Operator

The eddy [1] takes an arbitary number of inputs, connected to the data sources associated with a query to be executed. Its single output returns data tuples. It is also connected with the operators required to perform the query. The eddy maintains an internal tuple buffer with the tuples already seen, but not yet sent to the output or discarded. At each time step, it examines a tuple in the buffer and chooses an operator to route it

to, or, if appropriate, sends the tuple to the output. If the tuple buffer is empty, the eddy chooses a source to pull tuples from.

To route tuples so that the query is computed correctly (without introducing duplicates), the eddy must enforce routing constraints and must maintain additional meta-data about the tuples, the tuple descriptor.

2.2 Operators Involved in Query Processing

In our query processing setting, we focus on conjuctive selections queries over one relation, and natural join queries over many relations. The operators that take part in query processing and their semantics are summarized in Table 1. Let $r \in R$ denote that tuple r belongs to the relation R, and let $r \in G \supset R$ denote that tuple r contains at least the attributes of relation R. We also let $t \in STAIR(R.a)$ denote that tuple t is stored in $STAIR(R.a)$.

The scan operator fetches new tuples from a data source. The index operator returns all tuples from a data source that match its input tuple in terms of the index predicate.

The unary selection operator returns its input tuple if the associated predicate evaluates to true and otherwise discards the tuple. The eddy must route tuples to the various selection operators so that no tuple is routed to the same operator more than once. Furthermore, no tuple can be routed to a selection operator with a predicate over a relation other than the tuple's relation. To achieve that, the eddy has to associate a tuple descriptor with each tuple that consists of two bit arrays: the ready bits and the done bits. They are of size equal to the number of operators involved in the query. A ready bit is set if the tuple is eligible to be routed to the corresponding operator and a done bit is set if a tuple has already been passed from the corresponding operator. The eddy must keep track of its last action and set the ready and done bits accordingly.

Contrary to selections, a binary join is a stateful operator that can block the execution for a long period of time before returning the control to the eddy. In order for the eddy to be able to adapt more effectively, fully pipelined join algorithms like the Symmetric Hash Join (SHJ) are preferable. An SHJ maintains a hash table on each of its inputs. When it obtains a new tuple, it inserts it into the corresponding hash table, probes it into the other hash table, and returns the matches. For a join query executed using SHJs, the eddy has to keep track of the schema of the tuple, which serves as the tuple descriptor, in order to ensure semantically correct routing.

The STAIR [8] operator addresses the problem of the "burden of history," which occurs because past routing decisions result in intermediate results stored inside the binary SHJ operators, affecting the future ability of the eddy to adapt. It does so by splitting an SHJ operator into its two hash tables. A STAIR operator on relation R is a dictionary of tuples containing at least the attributes of R with a probe and an insertion operation. Each STAIR has its dual STAIR, together with which it forms a join. In the dual routing policy, every tuple, before being probed to a STAIR, is inserted into its dual, resulting to an execution identical to that of using binary SHJ operators [8]. However, STAIRs provide state migration primitives that move already stored tuples from one join to another, altering the accumulated state [8].

The SteM [18] operator similarly splits a join into its underlying physical operators, but differs from the STAIR in two ways. It adopts a history-independent join processing scheme by recomputing all the intermediate results, similarly to an n-way SHJ operator. Moreover, a more general set of routing constraints than the dual routing policy have been developed [18] that decouple the probes and the insertions, thus allowing the eddy to change not only the join plan, but also the join algorithms as well as the spanning tree on the fly.

Table 1: Query Execution Operators

Operator	Action	Input	Output	Description
$Scan(R)$	$get(R)$	nothing	null or $\{r_1, ..., r_n \mid n \geq 1\}$	Returns null if all the relation's tuples are over, otherwise the relation's tuples
$Index(R.a)$	$probe(r, Index(R.a))$	$r \in T, R.a \in T$	null or $\{r_1, ..., r_n \mid n \geq 1\}$	Returns null if no matches are found, otherwise the matches found
$\sigma_{p(R.a)}(R)$	$\sigma_{p(R.a)}(r)$	$r \in G \supset R$	null or r	If r passes predicate $p(R.a)$ outputs r, else outputs a special "null" tuple.
$R \bowtie S$	$route(r, R \bowtie S)$	$r \in G \supset R$	null or $\{rs_1, ..., rs_n \mid n \geq 1\}$	Inserts r into the appropriate hash table and probes it to the other. Outputs the matches if any or a "null" tuple.
	$route(s, R \bowtie S)$	$s \in G \supset S$	null or $\{r_1 s, ..., r_n s \mid n \geq 1\}$	Inserts s into the appropriate hash table and probes it to the other. Outputs the matches if any or a "null" tuple.
$STAIR(R.a)$	$insert(r, R.a)$	$r \in G \supset R$	r	Inserts r into the hash table and returns it to the caller.
	$probe(s, R.a)$	$s \in G \supset S$	null or $\{r_1 s, ..., r_n s \mid n \geq 1\}$	Finds matches for s in the STAIR and outputs concatenated results if any, or a "null" tuple.
	$demotion(R.a, t, t')$	$t \in G \supset R$, $t \in STAIR(R.a)$, $t' \in G' \subset G$		Replaces t by t'.
	$promotion(R.a, t, S.b)$	$t \in G \supset S, T$,		Removes t from $STAIR(R.a)$. Inserts t into $STAIR(S.b)$.
		$t \in STAIR(R.a)$		Probes $STAIR(T.b)$ using t. Inserts matches into $STAIR(R.a)$.
$SteM(R.a)$	$insert(r, R.a)$	$r \in R$	null or r	Inserts r into the hash table and returns it to the caller, if that is demanded by the routing constaints.
	$probe(s, R.a)$	$s \in G \supset S$	null or $\{r_1 s, ..., r_n s \mid n \geq 1\}$	Finds matches for s in the STAIR and outputs concatenated results if any, or a "null" tuple. Returns also s if demanded by the routing constraints.

Table 2: Routing Constraint Sets

Set name	Constraint	Description
DRC	BuildFirst	Every singleton tuple must be first be inserted into the SteM/STAIR of its relation. In the STAIRs case, every intermediate tuple must be first inserted into the appropriate STAIR.
	BounceBack	Only insertion tuples are returned to the eddy.
	Atomicity	Insertions and probes are atomically coupled.
	BoundedRepetition	No tuple can be routed to the same SteM/STAIR more than once.
DeRC	BuildFirst	Same as above.
	BounceBack	Same as above.
	TimeStamp	After r has probed a SteM and found a match s, the result $\langle r, s\rangle$ is returned to the eddy iff $TS(r) > TS(s) > LastMatchTS(s)$.
	BoundedRepetition	Same as above.
GeRC	BuildFirst	A singleton tuple must first be built into its SteM iff its relation has multiple access methods or it has an index access method.
	BounceBack	A SteM returns an insertion tuple, unless a duplicate is already stored. A SteM returns a probe tuple, unless it already contains all matches for it, or the SteM's relation has a scan access method, and all base-tuple components of the probe tuple are already stored in other SteMs.
	TimeStamp	Same as above.
	BoundedRepetition	No tuple can be routed to a SteM more than a finite amount of times.
	ProbeCompletion	A prior prober must not be routed to any SteM other than its completion table. A prior prober can be removed only after it has probed one of its probe completion access methods.

2.3 Routing Constraints

In order to achieve correct query results without duplicates, the routing possibilities of the eddy are subject to certain constraints. In selection-only queries, the proper management of the ready and done bitmaps for each tuple is the only necessary constraint, which results in the full space of possible selection orderings. In the case of join processing, several different sets of constraints can be enforced. We focus on three alternatives; a more complete coverage is found elsewhere [18].

In the simplest set of routing constaints, the Dual Routing Constraints (DRC), inserions and probes are atomically coupled. In the SteM's case, every new tuple from a base relation should be first inserted in the SteM of its relation, and then immediately probe another SteM, without forming a Cartesian product. In the STAIR's case, every new tuple (base or intermediate) should be inserted first in the appropriate STAIR and then probed to its dual. Using the DRC constraints, the plan space does not contain Cartesian products, there is a fixed spanning tree, and the only join algorithm is the SHJ. We also do not allow index structures in this case.

The Decoupled Routing Constraints (DeRC) break the atomicity of insertions and probes, and allow indexes and multiple access methods (which are modeled as scan operators in our formulation) that compete with each other. We discuss them only for the SteM's case. First, the eddy should insert a tuple into a SteM only once in its lifetime. Additional constraints are enforced in the SteMs implementation as described in [18], but we are interested only in the constraints for the routing policy. We now allow the eddy to explicitly fetch new tuples from scans even if the tuple buffer is not empty, thus changing the current tuple. The plan space expands and also includes several pipelined and non-pipelined join algorithms.

The General Routing Constraints (GeRC) are the least restrictive ones proposed. They expand the plan space by allowing cyclic queries without an a priori fixed spanning tree, and allowing the eddy to not build a SteM for a relation. Several constraints have to be enforced on the SteM and routing policy implementations [18]. We are interested only in the routing policy constraints, as we treat operators as black boxes. In order to avoid duplicates, the eddy must now insert a new tuple in the appropriate SteM

only if there are several access methods (AMs) or an index over the tuple's relation. In order for cyclic queries to be allowed, the eddy has to keep track whether a tuple (called a prior prober) has in the past probed into a SteM (called its probe completion table's SteM). A prior prober cannot in the future probe into any other SteM other than than its probe completion table's. Moreover, it should not be sent to output until it has probed into an access method over the relation of its probe completion table (one of the so-called probe completion AMs) [18].

3 Reinforcement Learning

We base our proposal in Section 4 for a formal framework for adaptive query processing on reinforcement learning (RL). Here, we cover briefly key RL concepts and algorithms. A complete survey is available in the literature [22].

3.1 The Reinforcement Learning Concepts

The setting of reinforcement learning is that of an autonomous agent that communicates with its environment in a well-defined manner. For each state s in a *state space* S, we define the eligibles actions $A(s)$. The *action space* $A = \bigcup_{s \in S} A(s)$ is the union of these actions. At each time step t, the agent receives a *state* signal $s_t \in S$. Based on the environment's state, the agent selects and executes an *action* $a_t \in A(s_t)$, based on some *policy* $\pi : S \times A \rightarrow [0, 1]$. The agent's actions generally change the environment, so at time step $t + 1$, the agent receives the next environment's state s_{t+1} and a numerical *reward* signal $r_{t+1} \in \Re$.

The learning process is essentially a way to continuously change the policy in order to maximize a cummulative metric of the reward signals, called the expected return R_t. The simplest metrics that can be used are the finite-horizon undiscounted return $R_t = \sum_{i=0}^{T} r_{t+i+1}$, that includes the next T future rewards, and the infinite-horizon discounted return $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} | \gamma \in [0, 1)$, that includes all the future rewards, discounting a reward seen after k time steps with a weight γ^k.

The expected return is of theoretical value only as it involves future rewards. Yet, it highlights the fact that the agent tries to maximize a delayed reward rather than the immediate one, so it is possible that the consequences of an action can be seen only far into the future. It is the objective of a learning algorithm to approximate this process. This is achieved by two fundamental techniques: *sampling* and *bootstrapping*. Sampling uses a statistical mean to approximate the expected return, whereas bootstrapping uses the next-state information to find out more about the current state's expected return.

Generally, the *Markov property* is assumed to hold. This means that the state signal carries all the information needed by the agent to be able to make a decision. Formally, the probability of a particular next state and reward can be predicted using only the current state and action:

$$\Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, \ldots, s_0, a_0\} = \Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\}$$

Given the Markov property, a so-called *Markov model* that consists of the transition probabilities and the expected values of the returns can defined as follows:

$$\begin{aligned} P^a_{ss'} &= \Pr\{s_{t+1} = s' | s_t = s, a_t = a\}, \\ R^a_{ss'} &= E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\} \end{aligned}$$

RL algorithms try to learn an optimal policy by learning an optimal *value function* $Q : S \times A \rightarrow \Re$. The value of a state-action pair, $Q^\pi(s, a)$, estimates how good it is to make a certain action when the agent observes a certain state, while following a policy π:

$$Q^\pi(s, a) \stackrel{\text{def}}{=} E_\pi\{R_t | s_t = s, a_t = a\}$$

The optimal policy π^* is defined as the one with the maximum value function. For the optimal value function, $Q^*(s,a)$, the Bellman optimality equations hold:

$$Q^*(s,a) = \sum_{s' \in S} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a' \in A(s')} Q^*(s',a') \right] \tag{3.1}$$

Solving 3.1 for all $(s,a) \in S \times A$ is equivalent to finding the solution of the RL problem. This is of minor practical importance since the Markov model is not known, and the learning is not interleaved with the actual acting, which is a prerequisite for adaptation. In Section 3.2, we present a collection of algorithms that require no knowledge of the Markov model and update the value functions after each action taken by the agent.

3.2 Learning Algorithms

Most RL algorithms satisfy the generic pattern of Generalized Policy Iteration (GPI). The learning process thus encompasses two phases.

In the *update* phase (also known as the evaluation phase), the agent tries to bring its stored value function closer to the policy actually followed.

In the *improvement* phase, the agent tries to tune the policy it follows according to its best knowledge about the environment, which is expressed by the stored value functions. The method used in this phase reflects the so-called "exploration-exploitation tradeoff."

The GPI scheme is presented in pseudocode fashion in Algorithm 1. We assume that the algorithm uses the Q values and that the learning process repeats forever. The update phase is captured in the $chooseAction$ function, whereas the improvement phase is captured in the $update$ function. The $takeAction$ function is responsible for the execution.

Algorithm 1 Generalized Policy Iteration

1: Initialize s
2: **loop**
3: $a = chooseAction(Q, A(s))$
4: $\langle s', r \rangle = takeAction(a)$
5: $Q(s,a) = update(Q, s, a, r, s')$
6: $s \leftarrow s'$
7: **end loop**

While numerous possibilities for the improvement phase exist, we consider the following solutions for the $chooseAction$ function. However, our framework follows the most general GPI scheme, so most RL algorithms can be applied in our setting:

greedy The greedy policy chooses the action with the largest Q value. It only *exploits* the current knowledge without exploring the state-action space. The $takeAction$ function in this case is:

$$a = greedy(Q, A(s)) = \underset{b \in A(s)}{argmax}\ Q(s,b)$$

random The random policy *explores* the solution space without taking into account the accumulated knowledge. It simply picks an action at random:

$$a = random(Q, A(s)) \Leftrightarrow Pr\{a = b\} = \frac{1}{|A(s)|}$$

uniform The uniform policy select an action with a probability propotional to its Q value:

$$a = \mathit{uniform}(Q, A(s)) \Leftrightarrow \Pr\{a = b\} = \frac{Q(s,b)}{\sum_{a \in A(s)} Q(s,a)}$$

ε-greedy The ε-greedy approach behaves as the greedy one with probability $1 - \varepsilon$ and as the random one with probability ε:

$$a = \varepsilon - \mathit{greedy}(Q, A(s)) \Leftrightarrow$$
$$\Pr\{a = b\} = \begin{cases} 1 - \varepsilon & \text{if } b = \mathit{greedy}(Q, A(s)), \\ \varepsilon & \text{otherwise} \end{cases}$$

simulated annealing This policy picks an action with a probability following the Boltzmann/Gibbs distribution. Furthermore, it utilizes a *temperature* variable that is gradually reduced as learning advances. The idea is to favor exploration in the beginning of the process and gradually move towards exploiting the gained knowledge:

$$a = \mathit{annealing}(Q, A(s)) \Leftrightarrow$$
$$\Pr\{a = b\} = \frac{\exp \frac{Q(s,b)}{T}}{\sum_{a \in A(s)} \exp \frac{Q(s,a)}{T}}$$

In the update phase, the agent can utilize its experience, i.e., a statistical mean of rewards based on the past, and it can also bootstrap, using known values of other states in order to update the current state. We consider the following kinds of updates.

Monte Carlo update The Monte Carlo policy updates $Q(s,a)$ in order to estimate the statistical mean of the rewards seen so far by the agent:

$$\mathit{MCupdate}(Q, s, a, r) = Q(s,a) + \frac{1}{n(s,a) + 1}\big[r - Q(s,a)\big]$$

Here, $n(s,a)$ denotes the occurences of the (s,a) pairs throughout the learning process.

constant-α Monte Carlo update The constant-α Monte Carlo policy uses a constant update parameter α, favoring recent updates over past ones:

$$\mathit{MCupdate}(Q, s, a, r) = Q(s,a) + \alpha\big[r - Q(s,a)\big]$$

Q learning update The Q learning policy is the most popular and well-studied in the RL literature. It approximates the optimal Q values directly by bootstrapping using the optimal value of the next state:

$$\mathit{Qupdate}(Q, s, a, r, s') =$$
$$Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s', a') - Q(s,a)\big]$$

4 Query Execution as a Learning Problem

In this section, we model query execution with eddies as a reinforcement learning problem (RLP). The eddy is the agent of the setting, the state is a transformation of the tuple descriptor of the current tuple, and the actions are the operators involved in the query.

Consider the example in Figure 1. An eddy is initiated to execute the query $\sigma_1(\sigma_2(R)) \bowtie S \bowtie T$. At each discrete time step t, the eddy sees the current tuple and determines a state signal from it. The current

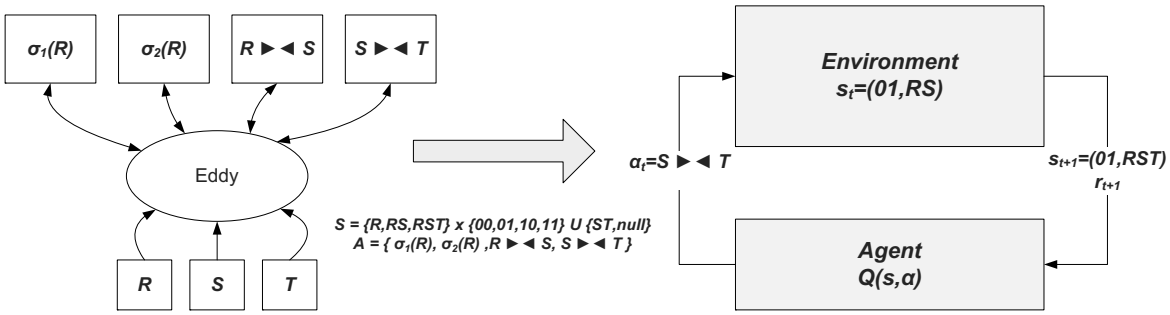


Figure 1: Query execution as a reinforcement learning problem.

state signal $s_t = (01, RS)$ means that the current tuple belongs to the relations R and S (it is a match previously returned by $R \bowtie S$), and has passed the second selection, but not the first. The eddy makes a decision about an operator to route that tuple to (an action in RL terms) based on the improvement policy followed, and the stored $Q(s, a)$ values. The eligible operators are $A(s_t) = \{\sigma_1, S \bowtie T\}$. In our example, the eddy chooses to route the tuple to the second join, $a_t = S \bowtie T$. The eddy then executes the selected action and waits for the operator to respond. The tuple returned by the operator is used to determine the next state. In this case, the join returned a match, so the next state is $s_{t+1} = (01, RST)$. The eddy also calculates the reward of the action, monitoring the actual time elapsed for the operator to respond back and the number of matches it produced, and updates the stored $Q(s, a)$ values based on the update policy used. Note that no selectivity or cost estimation is performed. Rather, the eddy just treats operators as black boxes, measuring the time elapsed for them to respond. This simple model assumes the existence of a special "null" tuple that is returned by the operators when there are no matches, and an iterator interface between the eddy and the operators. If the latter is not the case, a similar setting can be achieved as described in Section 4.5.

Following the norm in query optimization literature, we focus on two types of queries: conjuctive selection queries over one relation and natural join queries over many relations. We describe the first case in Section 4.1 and the second case in Section 4.2. We discuss the "burden of routing history" problem [8] in Section 4.3 and an extension to more general routing constraints over SteMs in Sections 4.4 and 4.5. In each section, we define the state and action spaces, we define the reward function, and incorporate the semantic and routing constraints into the model using the eligible actions sets $A(s)$. We present pseudocode that shows how the actual execution and learning are interleaved, following the GPI scheme of Algorithm 1. All the update and improvement policies presented in Section 3 can be used in all the settings presented in this section. The state and action spaces presented for the deconstructed problems can be straightforwardly combined to model more complex queries.

4.1 Selection Ordering

The selection ordering problem regards conjuctive selections queries of the form

$$\sigma_{p_1 \wedge p2 \wedge \ldots \wedge p_n}(R).$$

The objective is to find a permutation $[i_1, i_2, \ldots, i_n]$ for which the plan $\sigma_{p_{i_1}}(\sigma_{p_{i_2}}(\ldots \sigma_{p_{i_n}}(R)\ldots))$ is the one with the minimum cost. Although selections are generally cheap operators and the overhead of learning an optimal policy could be high compared to the cost of the actual execution, many classes of join queries can be reduced to the selection ordering problem [2]. In order to execute a selection query, an eddy is connected with one access method on relation R and n selection operators.

The state space is formed by a combination of the ready and done bits of the current tuple. For selections over one relation one of the two suffices, so we use the done bits in our formulation. The extension to

selections over many relations is straightforward. The action space consists of the selections involved in the query as well as the $get(R)$ action that fetches a new tuple from the relation and the $output$ action that returns a result tuple:

$$S = \{s_i | \langle donebits \rangle_2 = i\} \cup \{null\},$$
$$A = \{\sigma_i | i = 1, \ldots, n\} \cup \{get(R)\} \cup \{output\}.$$

The semantic constraints of the query can be easily captured using the sets $A(s)$. The eddy should fetch a new tuple if the current tuple is null, output the tuple if it has passed all relations, or choose an eligible selection otherwise:

$$A(s_{2^n-1}) = \{output\},$$
$$A(null) = \{get(R)\},$$
$$A(s_i) = \{\sigma_j | \textit{j-th done bit of } s_i = 0\}, s_i \neq null, s_{2^n-1}.$$

The reward function takes into consideration both the selectivity and the processing time of operators. Throughout this paper, we denote by $cost$ the time elapsed until the operator reports back to the eddy. We use negative rewards, as the notion of penalty is more straightforward than the one of the reward in this context. A selection gets a low reward, propotional to the time elapsed for the processing of the tuple, if the tuple passed the predicate, and a large reward otherwise. We choose not to reward a selection with a zero when it does not return a match. Instead, we use a reward proportional to the cost but smaller by a factor ε than the reward in the case of a returned match. In this way, operators with equal selectivities but different processing times can be more easily compared:

$$r(s_t, \sigma_i) = \begin{cases} -\varepsilon \cdot cost & \text{if } s_{t+1} = null, \\ -cost & \text{otherwise.} \end{cases}, \varepsilon \ll 1 \tag{4.1}$$

In the case of a multi-threaded execution environment in which an operator can receive a tuple only if it is not busy, the back-pressure phenomenon [1] keeps track of the operator costs so it is sufficient for the reward function to monitor only the operator selectivity:

$$r(s_t, \sigma_i) = \begin{cases} 0 & \text{if } s_{t+1} = null, \\ -1 & \text{otherwise.} \end{cases} \tag{4.2}$$

Rewards for the $output$ and $get(R)$ actions are 0. Reward function 4.2 approaches the negative of the operator's selectivity when averaged, whereas 4.1 approaches the negative product of the operator's selectivity and cost. The size of the state space is $|S| = 2^n + 1$, whereas the size of the action space is $|A| = n + 2$. The state space grows exponentially with the number of operators, which may seem as a problem if an algorithm needs to store the $Q(s, a)$ function in a tabular form. However, selections are inherently stateless operators, so only one value per action $Q(a)$ should be stored. The notion of the state is used only to capture the semantic constrains of the query in the $A(s)$ sets and to determine the reward by keeping track of the next state (state aggregation [21]). In this sense, selection ordering is an equivalent problem to the well studied k-armed bandit problem [3]. Algorithm 2 shows how the learning process can be interleaved with the execution of a selection query using an iterator interface. The $getState$ function returns the state of a tuple, and the $calcReward$ function calculates the reward. The $chooseAction$ and $update$ functions can be any of those mentioned in section 3.2. If rewards 4.2 are used, the Q values approximate the negative of the "tickets" in the lottery scheduling algorithm. Therefore, a combination of the uniform improvement policy and Monte Carlo updates forms the lottery scheduling algorithm. A combination of the random improvement policy and Monte Carlo updates is equivalent to the naive routing policy [1].

Algorithm 2 $EddySelections.getNext()$

1: $s \leftarrow null$
2: $a \leftarrow chooseAction(Q, A(s))$
3: $t \leftarrow a.getNext()$
4: $r \leftarrow calcReward()$
5: $Q(a) \leftarrow update(Q(a), r)$
6: **if** $t = null$ **then**
7: **return** $null$
8: **end if**
9: $s \leftarrow t.getState()$
10: **repeat**
11: $a \leftarrow chooseAction(Q, A(s))$
12: $t \leftarrow a.getNext()$
13: $r \leftarrow calcReward()$
14: **if** $t = null$ **then**
15: **return** $null$
16: **end if**
17: $s' \leftarrow t.getState()$
18: $Q(a) \leftarrow update(Q(a), r)$
19: $s \leftarrow s'$
20: **until** $s = n - 1$
21: **return** t

4.2 Join Order

Finding a good order for the join operators of a query involving many relations is the most essential and difficult part of the query optimization process. We assume an acyclic query of natural joins over n relations that can be executed without Cartesian products:

$$R_1 \underset{a_1}{\bowtie} R_2 \underset{a_2}{\bowtie} \cdots \underset{a_{n-2}}{\bowtie} R_{n-1} \underset{a_{n-1}}{\bowtie} R_n.$$

Other join types such as a θ join can be modeled as a selection over a natural join. This section assumes that the DRC routing constraints hold. In order for a join query to be executed, an eddy is connected with n source modules on the n relations, participating to the query, and the modules used to execute the joins. We introduce the notion of the join action, which, depending on the operators used (SHJs, STAIRs or SteMs) may mean any of the following, assuming that t is the current tuple seen by the eddy.

- In the binary SHJ operators case, a join action on a tuple is simply routing this tuple to the appropriate SHJ operator:
$$\bowtie_i = route(t, R_i \underset{a_i}{\bowtie} R_{i+1}), t \in R_i \vee t \in R_{i+1}.$$
- In the STAIRs case, a join action is a combination of inserting a tuple to a STAIR and then probing its dual:
$$\bowtie_i = probe(insert(t, R_i.a_i), R_{i+1}.a_i), t \in G \supset R_i,$$
$$\bowtie_i = probe(insert(t, R_{i+1}.a_i), R_i.a_i), t \in G \supset R_{i+1}.$$

- The SteMs case is the same as the STAIRs, with the difference being that non-singleton tuples do not get inserted into the SteMs:

$$\bowtie_i = probe(insert(t, R_i.a_i), R_{i+1}.a_i), t \in R_i,$$

$$\bowtie_i = probe(insert(t, R_{i+1}.a_i), R_i.a_i), t \in R_{i+1}.$$

With the above interpretation in mind, the join order problem in the eddy context becomes a problem of learning the optimal policy in the following RL problem formulation. The schema of the tuple can uniquely determine its execution history, so it can serve as the state variable together with the special null tuple. The action space is formed by the join actions, and the special $get(R_i)$ and $output$ actions as they were introduced in Section 4.1. The semantic constraints can be expressed by the sets $A(s)$ as follows:

$$\begin{aligned}
S &= \{tuple\ schema\} \cup \{null\}, \\
A &= \{\bowtie_i | i = 1, \ldots, n-1\} \cup \\
&\quad \{get(R_i) | i = 1, \ldots, n\} \cup \\
&\quad \{output\}, \\
A(null) &= \{get(R_i) | i = 1, \ldots, n\}, \\
A(R_1 R_2 \cdots R_n) &= \{output\}, \\
A(s) &= \{\bowtie_i | s \in G \subseteq R_i \vee s \in G \subseteq R_{i+1}\}.
\end{aligned}$$

Since we do not allow Cartesian products and use a fixed spanning tree, the size of the state space does not grow exponentially on the number of relations involved in the query but rather quadratically, whereas the size of the action space is linear on the number of relations.

The reward function is defined in a similar way to the selection order problem, with the difference being that now a join can return several (m) matches. In order to keep track of the selectivity as well as the cost of the operator, the reward function 4.3 should be used, whereas if only selectivities have to be monitored, definition 4.4 is suitable. Rewards for the $output$ and $get(R)$ actions are 0.

$$r(s_t, \bowtie_i) = \begin{cases} -\varepsilon \cdot cost & \text{if } s_{t+1} = null, \\ -m \cdot cost & \text{otherwise.} \end{cases} \tag{4.3}$$

$$r(s_t, \bowtie_i) = \begin{cases} 0 & \text{if } s_{t+1} = null, \\ -m & \text{otherwise.} \end{cases} \tag{4.4}$$

Algorithm 3 combines query execution and policy learning using an iterator interface and a stack [7]. [1]

Unlike selection ordering, in the join order problem the state variable plays a vital role in the learning process. The $Q(s, a)$ value does not take into account only the cost and selectivity of join a. If the Q update policy is used, it approximates the cost of the full join plan until a tuple is ready for output, since the update takes into account the best next state-action pair. Put differently, the join order problem is more difficult than a bandit problem, because local decisions affect greatly the future. The RL model with Q updates is a model strong enough to capture this complexity. The Q learning algorithm has asymptotic convergence guarantees if the problem characterisits stabilize. Therefore, through local rewards and decisions, an optimal or near-optimal join plan can be learned.

[1] In this implementation style, the rewards do not take into account the number of matches m, as one reward is calculated for each match in line 5 of Algorithm 3. They are $-cost$ or $-\varepsilon \cdot cost$, depending on whether the join returned a match or not.

Algorithm 3 $EddyJoins.getNext()$

```
1: s ← null
2: if stack ≠ Empty then
3:    a ← top(stack)
4:    t ← a.getNext()
5:    r ← calcReward()
6:    s' ← t.getState()
7:    Q(a) ← update(Q, A(s), s', r)
8:    if t = null then
9:       pop(top(stack))
10:      goto 2
11:   end if
12: else
13:   s ← s'
14:   a ← chooseAction(Q, A(s))
15:   t ← a.getNext()
16:   r ← calcReward()
17:   if t = null then
18:      return null
19:   end if
20:   s' ← t.getState()
21:   Q(a) ← update(Q, A(s), s', r)
22: end if
23: s ← s'
24: a ← chooseAction(Q, A(s))
25: if a = output then
26:   return t
27: else
28:   push(a)
29:   goto 2
30: end if
```

4.3 The Migration Problem

The STAIR operator provides the $demotion(R.a, t, t')$ and $promotion(R.a, t, S.b)$ primitives to handle the inner state accumulated in its hash tables during query execution. The state can be changed if it does not anymore agree with the best plan to be followed. These primitives combined result in a complete state migration from one join to another, through the $Migrate$ action.

$Migrate(\bowtie_i \longmapsto \bowtie_{i+1}) =$
for all $t \in R_i R_{i+1}$ stored in $R_{i+1}.a_i$ **do**
  $demotion(R_{i+1}.a_{i+1}, t, t')$
  $promotion(R_{i+1}.a_{i+1}, t', R_{i+1}.a_i)$
end for

$Migrate(\bowtie_{i+1} \longmapsto \bowtie_i) =$
for all $t \in R_i R_{i+1}$ stored in $R_{i+1}.a_{i+1}$ **do**
  $demotion(R_{i+1}.a_i, t, t')$

$promotion(R_{i+1}.a_i, t', R_{i+1}.a_{i+1})$

end for

State migration can be a very costly operator. When it is worthwhile to migrate is a difficult problem, and only a greedy solution has been proposed up to now [8]. We show that state migration can be treated as a stateless RLP, orthogonal to the one of join order. Let the pair $(R_i, \bowtie_j)$ denote that up to now the eddy knows that relation R_i is stored in operator $\bowtie_j$. We define the state of the problem as the list of current knowledge about tuple storage:

$$S = \Big\{ \big[(R_1, \bowtie_{i_1}), (R_2, \bowtie_{i_2}), \ldots, (R_n, \bowtie_{i_n})\big] | i_k \in \{k-1, k\}, \\ k \in \{1, 2, , \ldots, n-1\}, i_1 = 1, i_n = n-1 \Big\}.$$

The state has to be initiated according to the first tuple insertions during query execution. The migration action $Migrate(\bowtie_i \longmapsto \bowtie_j)$ alters the appropriate pair in the state variable. The action space is thus defined as:

$$A(s) = \big\{ Migrate(\bowtie_{i+1} \longmapsto \bowtie_i) \vee Migrate(\bowtie_i \longmapsto \bowtie_{i+1}) | \\ i = 1, \ldots, n-2 \big\} \cup \big\{ nothing \big\}.$$

The "nothing" action prevents any state migration when its cost is unjustifiably high. Our definition of the reward function augments the latter action with a fairly large reward. The reward of a migration from a join to another is proportional to the migration cost, and the difference of the Q values of the two joins:

$$\begin{aligned} r([\ldots, (R_{i+1}, \bowtie_i), \ldots], Migrate(\bowtie_i \longmapsto \bowtie_{i+1})) &= \\ -cost \cdot \big[Q(R_{i+1}, \bowtie_{i+1}) - Q(R_{i+1}, \bowtie_i)\big], & \\ r([\ldots, (R_{i+1}, \bowtie_{i+1}), \ldots], Migrate(\bowtie_{i+1} \longmapsto \bowtie_i)) &= \\ -cost \cdot \big[Q(R_{i+1}, \bowtie_i) - Q(R_{i+1}, \bowtie_{i+1})\big], & \\ r([\ldots, (R_{i+1}, \bowtie_i), \ldots], nothing) &= 0, \\ r([\ldots, (R_{i+1}, \bowtie_{i+1}), \ldots], nothing) &= 0. \end{aligned}$$

The latter rewards definition has the interesting property that an $\varepsilon - greedy$ evaluation policy will perform a state migration with probability ε. The join order and state migration problems are orthogonal in the sense that they can be solved in different timescales (e.g. someone could seek for a possible migration after seeing 1000 tuples and a join order every tuple), but tightly connected since the migration reward is a function of the join actions' Q values.

4.4 Join Algorithm Learning

In this section, we investigate join processing using SteMs and the DeRC routing constraints. The eddy should now keep track of whether a base tuple has already been inserted into the appropriate SteM and not insert a tuple twice. A new bit of information, named $isNew$ has to be introduced in the state. This should be true until the tuple gets inserted into the appropriate SteM, when it is set to false. For non-singleton tuples, it should always be false. The new state space is

$$S = \big(\{tuple\ schema\} \times \{true, false\}\big) \cup \{null\}.$$

The action space consists of the SteM probe and insertion operations, that are now distinct actions, and the scan and index probe operations as follows, assuming n relations, k scans, and m indexes:

$$\begin{aligned} A = \quad & \{insert(SteM_i)|i = 1, \ldots, n\} \cup \\ & \{probe(SteM_i)|i = 1, \ldots, n\} \cup \\ & \{probe(Index_i)|i = 1, \ldots, m\} \cup \\ & \{get(R_i)|i = 1, \ldots, k\} \end{aligned}$$

For a given state, the competition of the possible SteM probes corresponds to the join order problem. The competition between SteM and index probes corresponds to the competition between SHJs and index joins. Further, in order for non-pipelined join algorithms, such as a Hybrid Hash Join, to be permitted the eddy should be able to explicitly request a new tuple from a scan instead of routing the current tuple to an operator. Therefore, the competition of SteM probes, index probes and scans results to the learning process of the best join algorithms. The following sets $A(s)$ allow such a competition:

$$\begin{aligned} A(null) &= \{get(R_i)|i = 1, \ldots, n \wedge R_i \; not \; finished\}, \\ A((R_i, true)) &= \{insert(SteM_i)\}, \\ A((R_i, false)) &= \{probe(SteM_j)|join \; eligible\} \cup \\ & \{probe(Index_j)|index \; eligible\} \cup \\ & \{get(R_j)|j = 1, \ldots, n \wedge R_j \; not \; finished\}, \\ A((R_{i_1} R_{i_2} \cdots R_{i_k}, false)) &= \{probe(SteM_j)|join \; eligible\} \cup \\ & \{probe(Index_j)|index \; eligible\} \cup \\ & \{get(R_j)|j = 1, \ldots, n \wedge R_j \; not \; finished\}, \\ A((R_1 R_2 \cdots R_n, false)) &= \{output\} \cup \\ & \{get(R_j)|j = 1, \ldots, n \wedge R_j \; not \; finished\}. \end{aligned}$$

A join is eligible if it is not a cartesian product, and an index is eligible if the probe tuple contains the index attribute.

As non semantically related actions (e.g. probes and scans) are competing with each other, the reward function takes into account the cost of each operator as follows:

$$\begin{aligned} r(s_t, insert(SteM)) &= -\varepsilon \cdot cost, \\ r(s_t, probe(SteM)) &= \begin{cases} -\varepsilon \cdot cost & \text{if } s_{t+1} = null, \\ -m \cdot cost & \text{otherwise,} \end{cases} \\ r(s_t, probe(Index)) &= \begin{cases} -\varepsilon \cdot cost & \text{if } s_{t+1} = null, \\ -m \cdot cost & \text{otherwise,} \end{cases} \\ r(s_t, get(R)) &= -\varepsilon \cdot cost, \\ r(R_1 R_2 \cdots R_n, output) &= 0. \end{aligned}$$

4.5 The SteMs General Case

We now move to the GeRC set of constraints. The eddy has to keep track whether a tuple is a prior probe. If it is, it cannot be probed into any SteM other than its probe completion table's, and even if it is an output tuple it should remain in the dataflow until it has probed one of its probe completion AMs. Moreover, the

eddy has to keep track of whether a tuple belongs to a relation that has an index of multiple access methods, so it should be first be inserted into the appropriate SteM. This adds some extra variables to the state space:

$$\begin{aligned} S = \big\{ &(tuple\ schema, isNew, hasMultipleAMs, \\ &hasIndexAM, isPriorProber, probeCompletionTable, \\ &probeCompletionAMs) | \\ &isNew \Rightarrow \neg isPriorProber \wedge \\ &\neg isPriorProber \Rightarrow (probeCompletionTable = null) \wedge \\ &\neg isPriorProber \Rightarrow (probeCompletionAMs = null) \big\}. \end{aligned}$$

The action space and the reward function are identical with the ones in Section 4.4. The only difference is that the eddy has to enforce the GeRC constraints. This is achived via the $A(s)$ sets as follows:

$$\begin{aligned} A(null) &= \{get(R_i) | i = 1, \ldots, n \wedge R_i\ not\ finished\}, \\ A((R_i, true, true, \_, false, null, null)) &= \{insert(SteM_i)\}, \\ A((R_i, true, \_, true, false, null, null)) &= \{insert(SteM_i)\}, \\ A((R_i, true, false, false, false, null, null)) &= \\ \{insert(SteM_i)\} \cup \{probe(SteM_j) | join\ eligible\} &\cup \\ \{probe(Index_j) | index\ eligible\}, &\cup \\ \{get(R_j) | j = 1, \ldots, n \wedge R_j\ not\ finished\}, & \\ A((R_i, false, \_, \_, false, \_, \_)) &= \\ A((R_{i_1} R_{i_2} \cdots R_{i_k}, false, \_, \_, false, \_, \_)) &= \\ \{probe(SteM_j) | join\ eligible\} &\cup \\ \{probe(Index_j) | index\ eligible\} &\cup \\ \{get(R_j) | j = 1, \ldots, n \wedge R_j\ not\ finished\}, & \\ A((R_1 R_2 \cdots R_n, false, \_, \_, false, \_, \_)) = \{output\} &\cup \\ \{get(R_j) | j = 1, \ldots, n \wedge R_j\ not\ finished\}, & \\ A((R_i, false, \_, \_, true, PCT, \_)) &= \\ A((R_{i_1} R_{i_2} \cdots R_{i_k}, false, \_, \_, true, PCT, \_)) &= \\ \{probe(SteM_{PCT})\} &\cup \\ \{probe(Index_j) | index\ eligible\} &\cup \\ \{get(R_j) | j = 1, \ldots, n \wedge R_j\ not\ finished\}, & \\ A((R_1 R_2 \cdots R_n, false, \_, \_, true, \_, CAMs)) &= \\ \{probe(Index_j) | Index_j \in CAMs\} &\cup \\ \{get(R_j) | j = 1, \ldots, n \wedge R_j\ not\ finished\}, & \\ A((R_1 R_2 \cdots R_n, false, \_, \_, true, \_, null)) &= \\ \{output\} &\cup \\ \{get(R_j) | j = 1, \ldots, n \wedge R_j\ not\ finished\}. & \end{aligned}$$

We denote by "_" any possible value and we assume that as soon as a tuple has probed one of its probe completion AMs, the corresponding field in its state is set to null. An asynchronous execution environment [18] makes the integration of execution and learning somewhat more difficult. The correctness of the result is guaranteed by the routing constraints formalized above, as well as special End-Of-Transmission

(EOT) tuples that inform the eddy that a certain SteM has all the matches of a previous input. These special tuples have to be incorporated into the reinforcement learning model as a seperate state, $s = EOT$, with the following eligible actions:

$$A(EOT) = \{insert(SteM_i) | SteM \ eligible\}.$$

Moreover, since the eddy does not wait for the result of an operator to respond, the reward and the next state of an action might be known only long after an action. A solution to the latter is to pass around in the dataflow not only the data tuples, but signals in the form of $(t, s_{prev}, a_{prev}, r)$ carrying the tuple with its descriptor as well as the state, action and reward that produced this tuple as a result. A simple algorithm that shows the execution and learning coupled in this fashion is shown in Algorithm 4.

Algorithm 4 $EddySteMs$ asynchronous execution

1: $s = initState()$
2: **repeat**
3: receive current $(t, s_{prev}, a_{prev}, r)$
4: $s \leftarrow t.getState()$
5: $Q(s_{prev}, a_{prev}) \leftarrow update(Q, A(s), r, s)$
6: $a \leftarrow chooseAction(Q, A(s))$
7: route t to operator a
8: **until** end of execution

5 Implementation Details and Experimental Results

5.1 Implementation Details

The eddy has been implemented in an iterator-based environment in the context of PostgreSQL [7, 8] as well as in the context of TelegraphCQ that uses the fjord interface [16] and an asynchronous, multi-threaded execution environment. In this paper, we have generally assumed a single-threaded iterator-based execution environment that makes analysis easier, and is also the base of our implementation with the exception of section 4.5 that gives implementation guidelines for the asynchronous case [18].

We have implemented our algorithms both in a simulation fashion and in the context of the publicly available TelegraphCQ system. For the latter, we used the so-called "single-query" eddy execution mode, which is an iterator-based implementation of eddies, SteMs and STAIRs that uses a stack to execute join queries [7]. There have been two approaches in literature for implementing the tuple state, namely the ready and done bit arrays [1], and a routing policy data structure [7]. We chose to leverage the latter, although an implementation of the state variable using bit arrays is straightforward. We implemented all the improvement and update policies mentioned in Section 3 for the selection order and join order problems as they are described in Sections 4.1 and 4.2. For the first implementation, selections are modeled as artificial cost units with a randomized decision according to a user-defined selectivity. Joins are modeled as a selectivity-cost pair but they actually store the tuples following the SHJ algorithm in order to maintain their state. We use a stack to process join queries as described in Algorithm 3.

5.2 Adapting to Selection Characterisitics

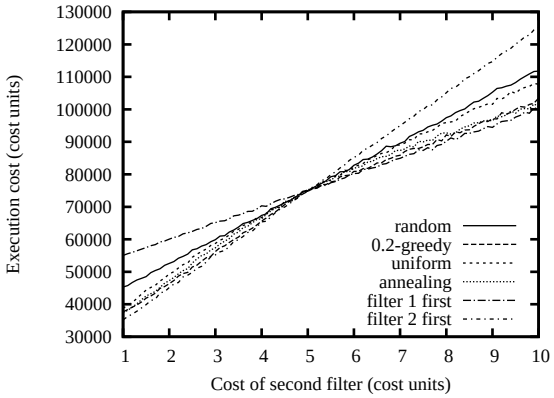
This section investigates how well various routing policies can adapt to changes of the cost and selectivity of selections. First of all, we use only simple MC updates for the selection ordering problem. They perform better than the constant-α MC updates as they favor all the past rewards equally. This is natural in the

selection ordering problem, as there is no notion of history. Our simulation also confirms this. We experiment with four of the improvement policies mentioned in Section 3.2, namely the random, a 0.2-greedy, the uniform and the annealing policy. Since we use simple MC updates, the lottery scheduling algorithm is fairly represented by the uniform improvement policy. We use two selection operators on a relation of 10000 tuples for our experiment. We reduce the temperature for the annealing policy by a factor of 0.94 at each tuple seen.

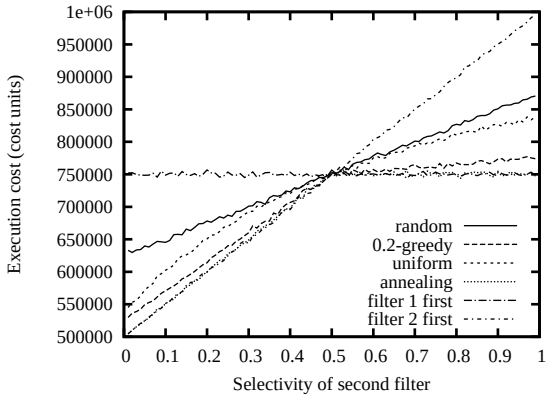
For our first experiment, we fix both selectivities at 0.5, fix the cost of the first operator at 5 units and allow the cost of the second vary from 1 to 10. Figure 2(a) shows the execution cost for the four improvement policies. The 0.2-greedy, the uniform and the annealing policies perform near-optimally with simulated annealing performing slightly better than the others. They all switch their routing preference as soon as the first selection becomes a better choice.

For our second experiment, we fix the cost of the operators and the selectivity of the first selection, and allow the selectivity of the second selection to vary from 0 to 1. Selectivities prove harder to be monitored than costs. Figure 2(b) shows that the 0.2-greedy and the annealing policies outperform the uniform policy. Figure 2(c) shows the cummulative ratio of tuples sent to the second selection first, and sheds some more light in the execution of the query. The annealing policy has the steepest transition curve as the first selection becomes a better choice for routing. The 0.2-greedy policy also switches its preference fast, but always leaves a 0.2 probability for exploration, so it cannot route more than 80% of the tuples to the best operator. The uniform policy learns selectivities much slower, having a smoother transition curve.

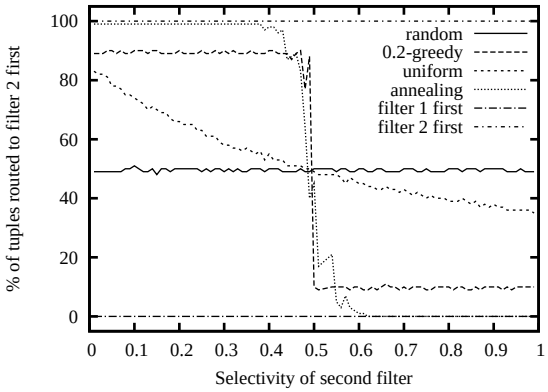
Testing the effect of dimensionality on the various improvement policies, we generated selections with random selectivities and costs. We varied the number of selections in the query from 2 to 10000 and measured the execution cost of the proposed improvement policies. The number of tuples of each relation was fixed at 10000. As the number of operators becomes higher, the learning capability deteriorates, as there is not enough data to learn from. This represents the curse of dimensionality in our setting. Figures 2(d) and 2(e) show the normalized execution cost, which is the execution cost divided by the execution cost of the optimal plan, in terms of the number of selections. The uniform policy performs much worse in settings with a small number of operators, raising to at least twice the cost of the optimal plan. A 0.2-greedy and the simulated annealing policy perform much better, in both a setting with a low number of operators and in one with a high number of operators. All policies end up with worse performance than a random policy when the number of selections in the query rises above 4000, showing the effect of dimensionality on the learning capabilities.



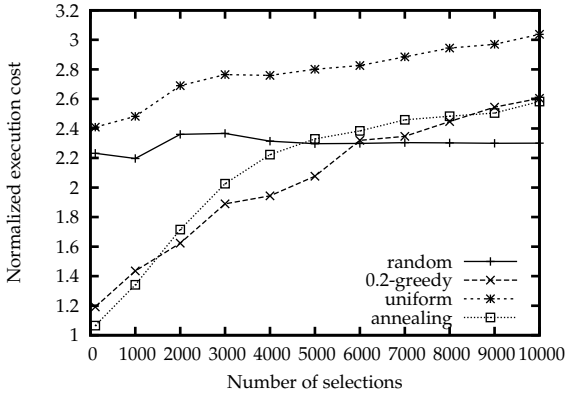
(a) Performance of two 50% selections. First selection has cost 5, cost of second selection varies across runs.



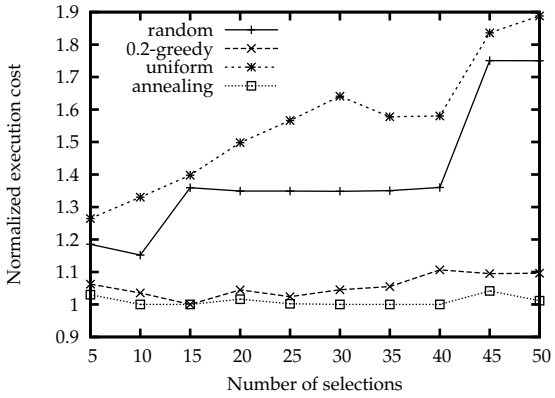
(b) Performance of two selections of cost 5. First selection has selectivity 0.5, selectivity of second selection varies across runs.



(c) Tuple flow of two selections of cost 5. First selection has selectivity 0.5, selectivity of second selection varies across runs.



(d) The effect of dimensionality: Normalized cost versus number of selections in the query.



(e) The effect of dimensionality: Normalized cost versus number of selections in the query.

Figure 2: Selection order experiments

5.3 Joins

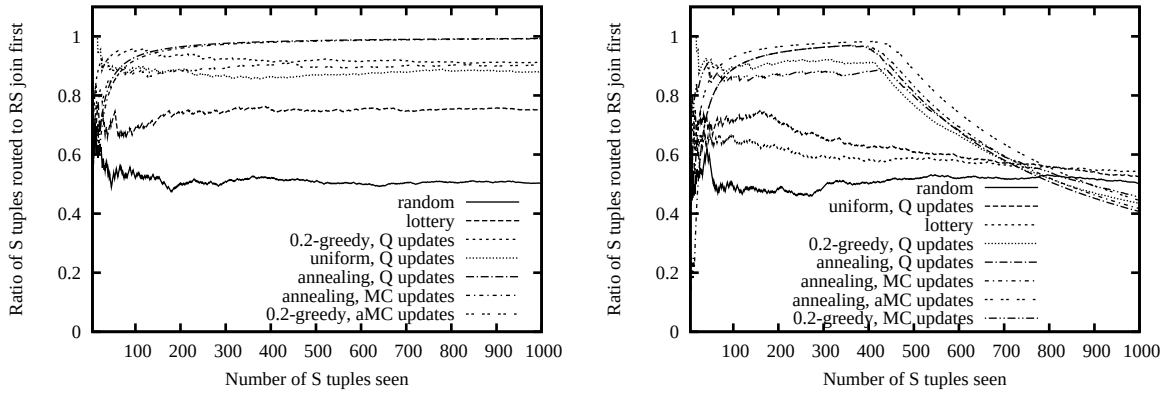
In this section, we investigate how the update and improvement policies behave in join processing in the spirit of Section 4.2. We have implemented the simple MC, constant-α MC and Q update policies, and the ε-greedy, random, uniform and annealing improvement policies. We have also implemented a lottery scheduling algorithm that uses the "tickets" data structure but is not multi-threaded, so it cannot monitor costs. We use it for comparison only for selectivity monitoring but again, the uniform policy with simple MC updates is a good representative of lottery scheduling in a single-threaded setting.

We issue a 2-join query $R \bowtie S \bowtie T$ over three relations R, S, T with 500, 1000, and 500 tuples respectively with controlled costs and selectivities. For our first experiment, we set the selectivities of the joins to 0.3 and 0.6 and their costs to 5, trying to investigate the transient response of our policies. We reduce the temperature for the annealing policy by a factor of 0.9 at each tuple. We use $\alpha = \gamma = 0.7$. These are the parameter settings we use throughout this section, unless otherwise stated. The tuple flow is depicted in Figure 3(a). We can see that the annealing, the uniform and a 0.2-greedy policy learn very fast the best join, with the annealing policy being the best. The lottery scheduling algorithm stabilized somewhere around 70%. For brevity, we only show some of the improvement-update policy combinations. The results for the rest are similar. As we discuss later, the update policy makes a difference mostly for monitoring selectivities.

In our second experiment, we investigate the adaptivity of our policies to environmental fluctuations. We issue the same query, fixing the cost and selectivity of $S \bowtie T$ to 5 and 0.3 respectively, and gradually increasing the cost and selectivity of $R \bowtie S$ from 1 to 10 and from 0.1 to 1.0 respectively while the query is being executed. We use the same parameter settings for the routing policies. Figure 3(b) shows that the annealing and 0.2-greedy perform well, gradually switching their preference shortly after $S \bowtie T$ becomes a better choice. The lottery scheduling and the uniform policy adapt slower. Again, the effect of the update policy is not so prevalent, so we omit some combinations of improvement and update policies that produced similar results.

We also conducted some more controlled experiments changing the cost and the selectivity of a join across runs. We issue the same query, fixing the selectivities of the joins at 0.5 and the cost of $R \bowtie S$ at 5.0, while changing the cost of $S \bowtie T$ across runs. Figure 4(a) shows that the annealing and the 0.2-greedy policy clearly outperform the uniform policy and monitor costs in a near-optimal way. We do not show the lottery scheduling algorithm, as in our implementation it does not monitor costs, but we expect it to perform similar to the uniform policy with Monte Carlo updates. The performance using other kinds of updates is similar and omitted from the graph. All the update policies are able to monitor the operators' costs.

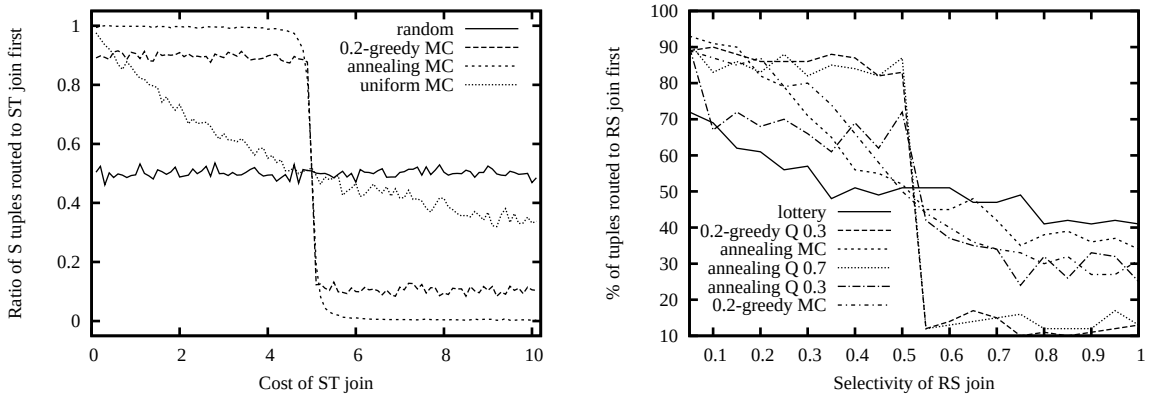
For our next experiment, we vary the selectivity of $R \bowtie S$ across runs. The results are shown in Figure 4(b). Selectivities prove much harder to learn than costs. The simple MC and the constant-α MC update policies yield poor adaptivity results, regardless of the improvement policy. Figure 4(b) shows the simple MC case for the annealing and a 0.2-greedy policy, that perform similarly to the lottery scheduling scheme. On the other hand, the Q update policy acts as a "forgetting scheme", and coupled with the annealing or the ε-greedy improvement policy manages to perform near-optimally. This result confirms our observation of Section 4.2 that the state variable and Q updates can efficiently capture the complexity of the join order problem, unlike simple MC updates.



(a) Tuple flow of a two join query. $R \bowtie S$ has selectivity 0.3 and cost 5, $S \bowtie T$ has selectivity 0.6 and cost 5.

(b) Tuple flow of a two join query. $S \bowtie T$ has selectivity 0.3 and cost 5, $R \bowtie S$ has selectivity and cost increasing gradually over time from 0.1 to 1.0 and 1.0 to 10.0 respectively.

Figure 3: Transient response of join queries



(a) Tuple flow of a two join query. Cost of $S \bowtie T$ varies across runs, cost of $R \bowtie S$ is fixed at 5.0.

(b) Tuple flow of a two join query. Selectivity of $R \bowtie S$ varies across runs, selectivity of $S \bowtie T$ is fixed at 0.5.

Figure 4: Join order experiments

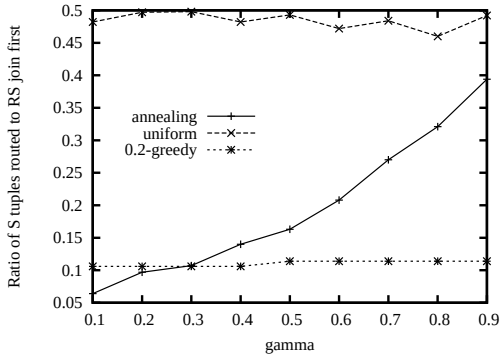
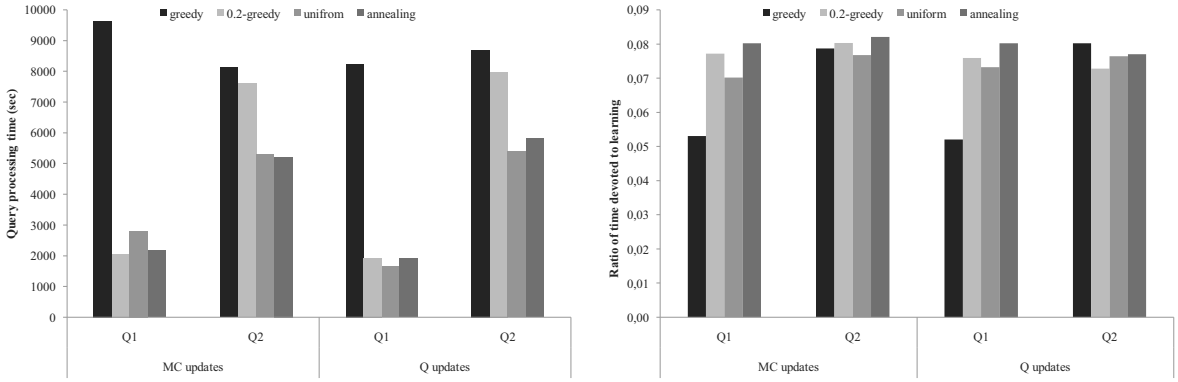


Figure 5: The effect of γ to the annealing policy.

Another observation is that the speed of convergence can be controlled via the γ parameter. We show in Figure 4(b) the performence of the annealing policy with Q updates for $\gamma = 0.3$ and $\gamma = 0.7$. Increasing the γ parameter slows down the learning process. This may be desirable for highly dynamic settings, in which the annealing policy can "overlearn" the environmental characteristics. To investigate this effect further, we issue a query $R \bowtie S \bowtie T$ that is "difficult" to learn. The costs of the joins are equal to 5, $R \bowtie S$ has selectivity 0.55 and $S \bowtie T$ has selectivity 0.45. We use Q updates and the 0.2-greedy, annealing and uniform policies. Figure 5, shows that the uniform policy routes to the joins with almost equal probability and the 0.2-greedy policy learns that the join $S \bowtie T$ is the best and routes 90% of the tuples to it. Using the annealing policy on the other hand, we can control the aggressiveness of learning using the γ parameter. The ratio of S tuples routed to $S \bowtie T$ first can vary from 95% to 60%.

To understand the overheads associated with the proposed policies, we conducted experiments with a data set modeled after the TPC-C benchmark. The learning overhead that is reported consists of the time needed to maintain and update the Q values and the time to select an action for each tuple seen. We ran two 5-table joins, Q_1 and Q_2. We present the results for the greedy, 0.2-greedy, uniform, and annealing policies with constant-α MC and Q updates. As shown in Figure 6(b), the learning overheads amount to 5% to 10% of the query execution time. The greedy policy naturally has the smallest overhead, but performs much worse in terms of running time, as shown in Figure 6(a). The annealing and uniform policies exhibit the highest overheads, but yield the best performance in all cases. The Q updates do not introduce substantially more overhead, as the search for the best Q value at each update yields a better query plan. Large intermediate join results caused by low-overhead, but poor routing decisions incur high costs, as the tuples of such results need to be processed further. For example, in Q_1 with MC updates, the greedy policy's overhead is nearly half that of the annealing policy, but it performs 5 times worse. These experiments thus suggest that it is worthwhile to expend resources on making good routing decisions, rather than simply using a low-overhead policy.



(a) 4 join queries – performance

(b) 4 join queries – learning overhead

Figure 6: 5 table joins on TPC-C data

As a conclusion, the annealing and the ε-greedy improvement policies can learn the environmental characteristics faster than the uniform policy or the lottery scheduling algorithm. They use a less fair competition scheme, and favor the best operator with a higher probability. In the selection order problem, simple MC updates behave well since the Q values estimate the product of the cost and the selectivity of the operators. On the other hand, in the join order problem, simple MC and constant-α MC updates are able to monitor the operators' costs but fail to monitor selectivities. The Q update policy succeeds to the latter, effectively capturing the complexity of the join plan space.

6 Related Work

This paper builds on several works in adaptive query processing. The eddy was proposed by Avnur and Hellerstein [1] as a means of achieving a highly adaptive approach to query processing. The SteM [18] and STAIR [8] operators expose the internals of a join state to the eddy and represent history-independent and history-dependent approaches to join processing. The eddy has been implemented as a traditional DBMS operator [7] and has also been extended to handle continuous queries over streaming data [5, 6, 17]. This paper's contributions are orthogonal to these and can be applied in all the proposed settings. A comprehensive survey of adaptive query processing is available in the literature [9].

Reinforcement learning is a brand of unsupervised learning in which the learner tries to map situations to actions in order to maximize a numerical reward signal using a trial-and-error search. As a research field, it derives from early research in artificial intelligence and optimal control and has attracted substantial interest in recent years, resulting in relative maturity [13, 22]. We are not aware of other works that apply reinforcement learning to adaptive query processing.

A handful of proposals exist for eddy routing policies. The naive eddy policy [1] randomly routes tuples to operators. When used in a multi-threaded environment, this policy is, however, capable of monitoring the costs of the operators. The lottery scheduling policy [1, 24] awards tickets to operators that do not return matches to keep track of their selectivities, and it selects an operator with a probability propotional to its tickets. Several policies exist that emulate static plans and then pause execution periodically for a new plan search using more up-to-date statistics. This category includes the A-Greedy policy [2], a policy that maximizes partial results [19] and the rank ordering policy [15]. We depart from previous work by (1) introducing a framework that is capable of accommodating a broad spectrum of existing routing policies and (2) introducing novel routing policies, all of which can adapt in a per-tuple fashion or in a batch-processing mode. Finally, a number of routing policies have been examined in a distributed context, in which the eddy does not exert central control; instead, every operator chooses an operator to send its output to [23].

7 Conclusions and Future Work

The paper proposes a formal framework for adaptive query processing that builds on concepts from the reinforcement learning theory. Query optimization in the context of eddies is framed as an unsupervised learning problem with quantitative rewards. The framework encompasses conjunctive selection queries as well as join queries using all of the approaches proposed in literature (binary SHJs, STAIRs, SteMs); and it covers a variety of routing constraint sets that correspond to a spectrum of join plan spaces.

Using the Generalized Policy Iteration scheme, query execution is broken down into an update and an improvement phase. We show that the framework is capable of expressing existing routing policies as special cases of improvement and update policies. The framework introduces so-called Q values as the only metadata stored about the operators. Experimental results are reported that show that these are able to capture the complexity of the join order problem using the Q learning algorithm, a task at which simpler alternatives fail.

Novel eddy routing policies can be derived naturally within the framework, by leveraging well-known reinforcement learning algorithms. Our empirical studies show that the ε-greedy and the simulated annealing improvement policies outperform more uniform approaches (e.g., the lottery scheduling algorithm) in a variety of settings. These policies are capable of learning the best query plan faster and adapting with a steeper transition curve to environmental changes. Simulated annealing has previously been found to be good for static query optimization [10, 11]; this paper's results suggest that simulated annealing may serve the same role for adaptive query processing.

We can identify several promising directions for future research. Incorporation of more advanced tech-

niques from the reinforcement learning field [14] can result in more powerful routing policies. Study of these techniques may also shed light on the underlying theoretical properties of adaptive query processing. Next, it would be of interest to expand the state space in order to include correlated selectivities or content-based routing techniques [4].

Finally, we believe that the eddy-based approach to adaptive query processing, if backed by routing policies with good convergence and adaptation characteristics, may perhaps be able to serve as a general query optimizer and processor, thus eliminating much of the complexity involved in today's query optimizers. Additional studies aimed at demonstrating this are thus in order.

Acknowledgments

The authors thank Amol Deshpande, Joe Hellerstein, and Sailesh Krishnamurthy for their valuable help with the TelegraphCQ system. Also, thanks to Carmen Ruiz, Dalia Tiešytė, Laurynas Biveinis and Dimitris Soulios for useful feedback on previous drafts of this paper.

References

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

[2] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, pp. 407–418, 2004.

[3] D. A. Berry and B. Fristedt. *Bandit Problems: Sequential Allocation of Experiments.* Routledge, 1986.

[4] P. Bizarro, S. Babu, D. J. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In *VLDB*, pp. 757–768, 2005.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[6] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB*, pp. 203–214, 2002.

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

[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. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[10] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD Conference*, pp. 312–321, 1990.

[11] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *SIGMOD Conference*, pp. 9–22, 1987.

[12] Z. G. Ives, A. Deshpande, and V. Raman. Adaptive query processing: Why, how, when, and what next? In *VLDB*, pp. 1426–1427, 2007.

[13] L. P. Kaelbling, M. L. Littman, and A. P. Moore. Reinforcement learning: A survey. *J. Artif. Intell. Res.*, 4:237–285, 1996.

[14] M. Kearns and S. Singh. Near-optimal reinforcement learning in polynomial time. In *Proc. 15th International Conf. on Machine Learning*, pp. 260–268, 1998.

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

[16] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pp. 555–566, 2002.

[17] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pp. 49–60, 2002.

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

[19] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *SIGMOD Conference*, pp. 275–286, 2002.

[20] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD Conference*, pp. 23–34, 1979.

[21] S. P. Singh, T. Jaakkola, and M. I. Jordan. Reinforcement learning with soft state aggregation. In *NIPS*, pp. 361–368, 1994.

[22] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[23] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pp. 333–344, 2003.

[24] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI*, pp. 1–11, 1994.

[25] Y. E. Ioannidis and S. Christodoulakis On the propagation of errors in the size of join results. In *SIGMOD Conference*, pp. 268– 277, 1991.

[26] S. Christodoulakis Implications of certain assumptions in database performance evauation. ACM TODS, 9(2):163–186, 1984

[27] Z. G. Ives, A. Y. Halevy and D. S. Weld. Adapting to source properties in processing data integration queries. In *SIGMOD Conference*, pp. 385–406, 2004.

[28] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pp. 238–249, 2005.

[29] M. Stillger, G. M. Lohman, V. Markl and M. Kandil. LEO–DB2's LEarning Optimizer. In *VLDB*, pp. 19–28, 2001.

[30] P. M. Aoki. How to avoid building datablades that know the value of everything and the cost of nothing. In *SSDBM*, pp. 122–133, 1999.