Lifting the Burden of History from Adaptive Query Processing*

Amol Deshpande[†] and Joseph M. Hellerstein[† ‡]

[†] University of California, Berkeley and [‡] Intel Research, Berkeley
{amol, jmh}@cs.berkeley.edu

Abstract

Adaptive query processing schemes attempt to reoptimize query plans during the course of query execution. A variety of techniques for adaptive query processing have been proposed, varying in the granularity at which they can make decisions [8]. The eddy [1] is the most aggressive of these techniques, with the flexibility to choose tuple-by-tuple how to order the application of operators. In this paper we identify and address a fundamental limitation of the original eddies proposal: the *burden of history* in routing. We observe that routing decisions have long-term effects on the state of operators in the query, and can severely constrain the ability of the eddy to adapt over time. We then propose a mechanism we call STAIRs that allows the query engine to manipulate the state stored inside the operators and undo the effects of past routing decisions. We demonstrate that eddies with STAIRs achieve both high adaptivity and good performance in the face of uncertainty, outperforming prior eddy proposals by orders of magnitude.

1 Introduction

"Stair above stair the eddying waters rose
circling immeasurably fast..."
– Percy Bysshe Shelley

In many scenarios, it is difficult or impossible at compile time to translate a declarative query into an efficient static execution plan. This problem has been highlighted in querying remote data sources [11, 1], in querying data streams [14, 3, 15], and even in traditional centralized databases [9, 4, 12]. To address this problem, a variety of adaptive query processing techniques have been proposed [8]. Among these, the eddy [1] is the most flexible and aggressive mechanism. An eddy is a tuple router that is placed at the center of a dataflow, intercepting all incoming and outgoing tuples between operators in the flow. By sitting at the center of the flow, the eddy can both observe the rates of all the operators, and make decisions about the order in which tuples will visit the operators. Eddies are intended to merge the statistics-collection and operator ordering facilities of a query optimizer into a query engine's runtime system.

In principle, eddies are able to choose different operator orderings for each tuple during query processing, subject only to the constraints of whether an operator is able to process a tuple (for example, a selection operator on table S can only process tuples from that table). But, as we observe in this paper, the *query execution plans* that eddies can effect for multi-join queries are limited not only by the semantic properties of the operators, but also by the *burden of routing history*: routing decisions made early in query's execution limit the eddy's routing options later on. These limitations can remove much of the adaptive power of eddies. The crux of the problem is the state accumulated by joins: once routed, a tuple that resides in the state of a join can effectively determine the order of execution for subsequently arriving tuples from other tables. To illustrate, we review an example from the original eddies paper [1]:

Example: Consider the query $R \bowtie_a S \bowtie_b T$, using two pipelining hash join operators (Figure 1). At the beginning of query processing, the data source for R is stalled, and no R tuples arrive. Hence the $R \bowtie_a S$ operator never produces a match, which makes it an attractive destination for routing S tuples: it efficiently removes work from the query engine. The result is that the eddy emulates a static query plan of the form $(R \bowtie S) \bowtie T$. Some time later, R tuples arrive in great quantity and it becomes apparent that the best plan would have been $(S \bowtie T) \bowtie R$. The eddy can switch the routing policy so that subsequent S tuples are routed to $S \bowtie_b T$ first. Unfortunately, this change is "too little too late": all the previously-seen S tuples are still stored in the internal state of the $R \bowtie_a S$ operator. As R tuples arrive, they *must* join with these S tuples before the S tuples are joined with T tuples. As a result, the eddy effectively continues to emulate the suboptimal plan $(R \bowtie S) \bowtie T$, even after its routing decision for S has

*This work was supported by NSF under grants 0208588 and 0205647, and by an IBM Fellowship.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

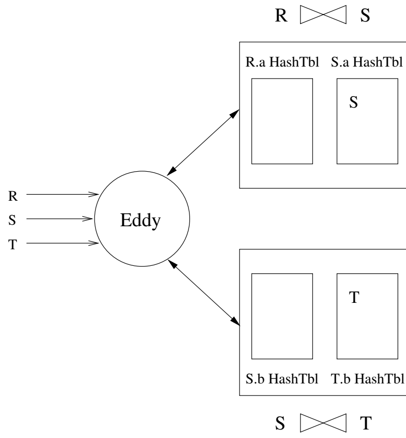

Figure 1: If the data from relation R is delayed, the eddy may route all of S to $R \bowtie_a S$, resulting in accumulation of state as shown.

changed.

This example illustrates both how easy it is to make incorrect routing decisions, and how these decisions can have permanent effects on the query plan achieved by an eddy – even when routing policies are changed. As we discuss in Section 2.3, the state accumulation inside the operators also makes it impossible for the eddy to perform aggressive adaptations such as changing the join spanning tree used for cyclic queries, or to perform *query scrambling* [18] in presence of delayed data sources.

In this paper we introduce a modified eddy architecture that makes use of a query operator called a STAIR (Storage, Transformation and Access for Intermediate Results), which holds the state traditionally encapsulated in joins. STAIRs address the problems illustrated above by allowing query state to be modified and migrated across STAIRs during query execution. The ability to modify and migrate state lifts the burden of history described above, enabling this architecture to undo the effects of prior routing decisions. In essence, STAIRs complete the picture begun with eddies: they allow a query executor to achieve the full effect of adapting the operator ordering at any point during runtime. As we discuss in Section 3.3, the flexibility in state management provides a number of additional novel optimization opportunities as well for trading off computation and storage during query execution.

We have implemented our architecture in the PostgreSQL 7.3 database management system in the context of the Telegraph project [2], and we present experimental results showing that eddies with STAIRs achieve both high adaptivity and good performance in the face of uncertainty, outperforming prior eddy proposals by orders of magnitude.

2 Eddies: Review and Pitfalls

We begin with a description of the eddy mechanism, and illustrate how state accumulation through bad routing choices can restrict eddy's ability to adapt. We then briefly review the SteMs architecture [16] that also provides a solution to this problem, and discuss why that solution is unsatisfactory. To make this discussion concrete, we focus on a common usage scenario for adaptive query processing [1, 10, 16, 18, 11], where the query processor is asked to evaluate a declarative select-project-join query over a set of finite relations that are being *streamed into* the query processor (from disk or from network). We ignore selection predicates, and restrict the choice of join operators to pipelining symmetric hash join operators. The discussion in this section, however, can be easily extended to other kinds of join operators as well (*e.g.*, index join operators, traditional non-pipelining join operators *etc.*).

2.1 Eddies

Traditional database systems execute queries by choosing a query plan a priori, and adhering to it throughout the query execution. The basic idea behind eddies is to treat query execution as a process of *routing* tuples through operators, and to allow changing the order in which tuples are routed on a per-tuple basis. A special *eddy* operator is used to route tuples between the query operators, and follows a simple procedure:

- Choose a tuple to process next; this could either be a new tuple from a base relation, or it could be the result of processing an earlier tuple.
- Among the operators that are *valid* routing destinations for this tuple, choose one, route the tuple to it, and store the resulting tuples in the eddy's internal buffer. Valid routing destinations for a tuple are determined by the semantic properties of the operators. For example, a tuple can be routed to a *join operator* only if the tuple contains a component from exactly one of the relations in the join.

Notation

Before going on, we define the notation we use in this paper. We use capitalized italics to denote base relations (*e.g.*, R, S), and small italics to denote base tuples belonging to those relations (*e.g.*, r, s). We use boldface letters to denote sets of relations (*e.g.*, $\mathbf{Q} = \{R, S\}$). Given a set of relations $\mathbf{Q}$ such that the relations in the set can be joined without use of cartesian products, we denote the resulting join relation by $\bowtie \mathbf{Q}$, or simply as the concatenation of the relations in $\mathbf{Q}$ (*e.g.* RS denotes the result relation obtained by joining R and S). We use small boldface letters to denote *intermediate* tuples belonging to such joined relations (e.g., $\mathbf{q}$ is a tuple in $\mathbf{Q}$.) Finally, we denote by R_τ all the tuples of R that have been processed by the eddy at time τ.

Definition 2.1 *[16] Consider a tuple* $\mathbf{q}$ *that belongs to the join of* k *base-tables* $T_1, \ldots, T_k$*. The tuples of these base-tables that participate in the generation of this tuple* $\mathbf{q}$ *are called* base-table components *of* $\mathbf{q}$*, and are denoted by* $q_{T_1}, \ldots, q_{T_k}$ *respectively. We denote* $\mathbf{q}$ *by* $q_{T_1} q_{T_2} \ldots q_{T_k}$*.*

Definition 2.2 *A tuple* $\mathbf{q}' \in \mathbf{Q}'$ *is called a* sub-tuple *of a tuple* $\mathbf{q} \in \mathbf{Q}$*, if* $\mathbf{Q}' \subseteq \mathbf{Q}$ *and all base-table components of* $\mathbf{q}'$ *are also present in* $\mathbf{q}$*. We call* $\mathbf{q}$ *a* super-tuple *of* $\mathbf{q}'$*.*

Given this background, Figure 2 shows the eddy instantiated for a three-relation join query $R \bowtie S \bowtie T$ (we will use

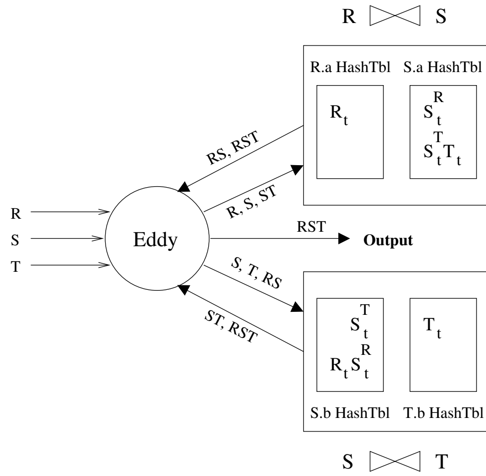

Figure 2: An eddy instantiated for $R \bowtie S \bowtie T$. Valid options for routing are labeled on the edges. The state materialized inside the join operators at time τ is shown, where S_τ^R and S_τ^T denote the parts of S_τ that were routed to $R \bowtie S$ and $S \bowtie T$ respectively.

this query as the running example throughout this paper), and the valid routing choices for various types of tuples. For example, the only valid routing choice for R tuples is the $R \bowtie S$ operator, whereas S tuples can be routed to either of the two join operators.

2.2 Uncertainties in Routing

The key to achieving good performance with eddies is making intelligent routing decisions. Unfortunately, the eddy can never have complete information about the data, or about the environment, and has to make decisions in presence of incomplete information. Continuing with our example query, let us say that the eddy was able to compute the selectivities of the two joins $R \bowtie S$ and $S \bowtie T$ accurately over the data it has seen. The eddy now could use these selectivities to make routing choices, under the assumption that these selectivities will remain constant in the future. But, there are many reasons why such extrapolation may turn out to be wrong.

- **Delays in arrival of data:** If there are delays in arrival of data from some relations, the eddy may have a very small sample of the data, leading to incorrect estimation of selectivities, and possibly of the sizes of the relations. To be able to estimate join selectivities accurately, both these factors need to be correctly estimated.
- **Order-dependent selectivities:** Selectivities are often correlated with the order of arrival of data. For example, if the data is ordered by *age*, then the selectivity of a predicate on *salary* changes as the tuples stream in.
- **Unpredictable data rates:** Unpredictable data rates can result in incorrect estimation of table sizes, and consequently of selectivities of joins involving those relations.

2.3 Burden of Routing History

Given the unpredictable nature of the data, there is not much the eddy can do to avoid making routing mistakes, and the penalties for such wrong decisions must be incurred to some degree during execution. Unfortunately, because of the state that gets materialized in the operators, especially the join operators, the effects of such mistakes can be long-lived. Figure 2 shows the state materialized in the joins at time τ. As we can see, the routing choices made for S determine what gets stored in the joins (S_τ^R was routed toward $R \bowtie S$, and S_τ^T toward $S \bowtie T$ resulting in those two parts of S getting stored in those join operators respectively). This results in several problems in query execution later. We discuss them in turn.

Constraints on Future Adaptation
The join state can significantly constrain the adaptation opportunities that the eddy has in future. Though the eddy can choose how to route a tuple when it first processes it, the choice made at that time constrains subsequent operator orderings for future results that involve this tuple. Coming back to our example, when a tuple $s \in S_\tau^R$ arrived at the eddy, the eddy could have chosen to route it to either of the joins. But once this choice has been made (in this case, $R \bowtie S$ operator), the query plan used for generating *any* result tuple that contains s (even one that may be generated in future), is constrained to be $(R \bowtie S) \bowtie T$. This is because any R tuples that come in later will be routed to $R \bowtie S$ and thus will join with s. As such, even if the eddy, in future, deduces that joining s with R is sub-optimal, it is prevented from doing any adaptation. This can result in significantly sub-optimal performance in many cases, as we will see in Section 5.

Cyclic Queries
A join query can be represented as a query graph, with vertices denoting the relations and edges denoting join predicates between relations. Cyclic queries are those in which the query graph has cycles. For example, the query $\sigma_{R.a=S.a \wedge S.b=T.b \wedge R.c=T.c}(R \times S \times T)$, is a cyclic query. As in traditional database systems, eddies choose a spanning tree of the query graph a priori, and are not able to change the spanning tree mid-execution. This is because the tuples that get routed to a join operator are "lost" if a spanning tree without that join operator is used later on. As an example, if we routed according to the spanning tree containing $R \bowtie S$ and $S \bowtie T$ in the example above, resulting in state as shown in Figure 2 at time τ, switching to the spanning tree consisting of $R \bowtie T$ and $S \bowtie T$ is not possible, because R_τ is stored inside $R \bowtie S$ and will not be available to the new spanning tree.

Pre-computation in presence of delays
If data from remote data sources is delayed, it might be attractive to perform other useful work while waiting for that data to arrive. Such pre-computation can be useful to produce partial results [17] that may contain missing attribute values, or to join the data that has already arrived as much as possible [18]. The eddy is prevented from performing such adaptations because of the accumulation of state inside the operators, and the eddy's inability to change rout-

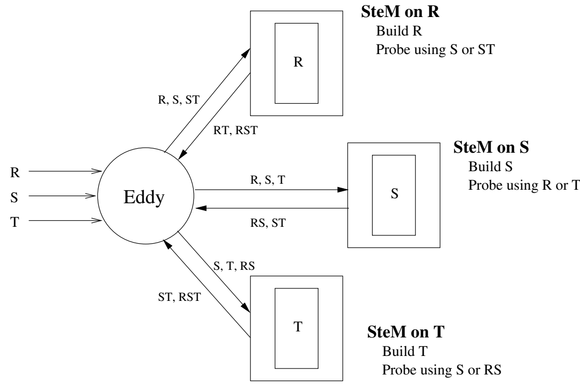

Figure 3: SteMs architecture for the query $R \bowtie S \bowtie T$; annotations on the edges show the types of tuples that are exchanged.

ing decisions once they are made. In fact, the inability to produce partial results aggressively was one of the main reasons the SteMs architecture that we describe below was developed.

2.4 SteMs: A Preliminary Solution

The State Modules (SteMs) architecture [16] is an extension of the eddy architecture, and inoculates eddies from these problems by ensuring that the state stored in the operators is entirely independent of the routing history. The main operator in this architecture is a *SteM*, which is instantiated for each relation in the query as shown in Figure 3. This operator stores all the tuples from that relation, and also handles all the probes involving that relation. The query is once again executed by routing tuples through these operators. As an example, when a new R tuple comes into the system, it is (1) inserted into the R SteM, (2) probed into the S SteM to find matching S tuples corresponding to the join $R \bowtie S$, and (3) the resulting RS tuples are probed into the T SteM to find matching T tuples in order to generate RST results. The intermediate RS tuples are not stored anywhere, and are thrown away as soon as results are produced.

As a result of not storing any intermediate tuples, the state accumulated inside the SteMs is independent of the routing history. This addressed some of the challenges mentioned above, but it is also the cause of two significant drawbacks:

- **Re-computation of intermediate tuples:** Since intermediate tuples generated during the execution are not stored for future use, they have to be recomputed each time they are needed.
- **Constrained plan choices:** More importantly, the query plans that can be executed for any new tuple are significantly constrained. For example, any new R tuple, r, that comes in at time τ must join with S_τ (the tuples for S that have already arrived) first, and then with T_τ. This effectively restricts the query plans that the eddy can use for this tuple to be $(r \bowtie S_\tau) \bowtie T_\tau$, even if the eddy *knows* that that plan is sub-optimal.

As we will see in Section 5, these inherent flaws with the basic design of the mechanism result in significantly worse performance than the original eddy architecture in most cases.

3 STAIRs

As we argued in the preceding section, and as we will further demonstrate in Section 5, routing mistakes are quite common in adaptive query processing, and the resulting burden of history can have long-lasting effects on the query execution. These effects can be attributed to the state that gets stored in the operators during execution. This observation naturally leads us to the basic idea behind our proposed modifications to the eddy architecture: *expose the state stored in the operators to the eddy, and allow the eddy to manipulate this state.* We do this by introducing an operator we call a *STAIR*, which holds the state traditionally encapsulated in joins, and provides the eddy with primitives to manipulate this state.

3.1 STAIR Operator

A STAIR operator encapsulates the state typically stored inside the join operators. More formally, a STAIR on relation R and an attribute a, denoted by $\mathcal{R}.a$, holds (possibly intermediate) tuples that contain a base-table component from relation R, and supports the following two basic operations[1]:

insert($\mathcal{R}.a$, t)
Given a tuple $\mathbf{t}$ that contains a base-table component from relation R, store the tuple inside the STAIR.

probe($\mathcal{R}.a$, *val*)
Given a value ***val*** from the domain of the attribute $R.a$, return all tuples r stored inside $\mathcal{R}.a$ such that $r.a$ = ***val***.

Figure 4 shows the STAIRs instantiated for our example 3-relation query. In essence, for each join operator that would have been instantiated in the original *EddyJoins*[2] approach of [1], we instead use two STAIRs that interact with the eddy directly. We will call the pair of STAIRs corresponding to a single join *duals* of each other. Note that even if both the joins were on the same attribute, we would have two STAIRs on relation S and attribute $a (= b)$. These two STAIRs are treated as separate operators, as they participate in different joins.

The query execution using STAIRs is similar to query execution using join operators. Instead of routing a tuple to a join operator as in EddyJoins, the eddy itself performs an *insert* on one STAIR, and a *probe* into its dual. In fact, in this paper, we will assume that the following property is always obeyed during query execution:

[1] To capture join predicates such as $R.a = S.a$ AND $R.b = S.b$, a STAIR can be more generally defined on a *list* of attributes from the schema of R. For ease of exposition we assume a single join attribute in our discussion; the extension to multi-attribute predicates is straightforward.

[2] We will refer to the original eddy architecture that uses explicit join operators by EddyJoins, to distinguish it from the other eddy-based architectures that we discuss in this paper.

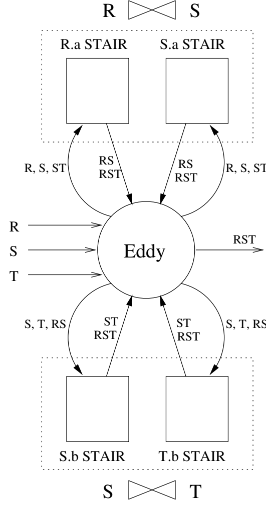

Figure 4: Executing $R \bowtie S \bowtie T$ using an eddy and STAIRs.

Dual Routing Property: *Whenever a tuple is routed to a STAIR for probing into it, it must be simultaneously inserted into the dual STAIR.*

This property is analogous to the BuildFirst constraint described in [16], and is also obeyed by the symmetric join operator. The Dual Routing Property can be relaxed by using timestamps in a similar fashion to the one described in [16], so that the probe must follow the insert in time but need not be atomically combined with the insert.

For brevity, we will use the phrase *routed to* $R \bowtie_a S$ *operator* to mean performing the two operations outlined above on the two STAIRs, $\mathcal{R}.a$ and $\mathcal{S}.a$, corresponding to the join.

3.1.1 State Management Primitives

Other than the two basic operations described above, STAIRs also support two state management primitives:

Demotion($\mathcal{R}.a$, t, t')
Intuitively, the *demotion* operation involves reducing an intermediate tuple stored in a STAIR to a sub-tuple of that tuple, by removing some of the base tuple components that it contains.

The following pre-conditions must be satisfied by the arguments to this function:

- $\mathbf{t}$ must be a tuple stored in $\mathcal{R}.a$,
- $\mathbf{t}'$ must be a sub-tuple of $\mathbf{t}$,
- $\mathbf{t}'$ must contain the base-table component of $\mathbf{t}$ corresponding to the relation R.

Given that these pre-conditions are satisfied, the demotion operation simply replaces $\mathbf{t}$ by $\mathbf{t}'$ in $\mathcal{R}.a$.

This operation can be thought of as *undoing* some work that was done earlier during execution, and this work may have to be redone if the tuple is required again in future. Figure 5 shows the result of applying $demotion(\mathcal{S}.b, r_1s_1, s_1)$ to an example initial state. As we can see, after applying this operation, the tuple r_1s_1 in $\mathcal{S}.b$ gets replaced by the tuple s_1.

Promotion($\mathcal{R}.a$, t, $\mathcal{S}.b$)
The *promotion* operation replaces a tuple in a STAIR with *super*-tuples of that tuple that are generated using another join in the query.

The following conditions must be satisfied by the input to this operation:

- $\mathbf{t}$ must be stored in $\mathcal{R}.a$.
- $\mathbf{t}$ must contain the base-table component corresponding to relation S (note that S and R may be identical).
- Let the dual STAIR of $\mathcal{S}.b$ be $\mathcal{T}.b$. Then, $\mathbf{t}$ must not contain the base-table component corresponding to relation T.

Intuitively, the point of promotion is to use the join $S \bowtie T$ to generate super-tuples of $\mathbf{t}$; the last two conditions simply make sure that this is a valid operation.

Given this, the promotion operation performs the following steps:

1. Remove $\mathbf{t}$ from $\mathcal{R}.a$,
2. Insert $\mathbf{t}$ into $\mathcal{S}.b$,
3. Probe $\mathcal{T}.b$ using the tuple $\mathbf{t}$, and
4. Insert the resulting matches (that are super-tuples of $\mathbf{t}$), if any, back into $\mathcal{R}.a$.

Figure 5 shows the result of applying $promotion(\mathcal{S}.b, s_3, \mathcal{S}.a)$, to an example initial state. As a result of this, the s_3 tuple in $\mathcal{S}.b$ gets replaced by r_1s_3 and r_2s_3, whereas s_3 itself gets stored in $\mathcal{S}.a$.

3.1.2 Duplicates

Both these state management operations, as described above, can result in a state configuration that allows spurious duplicate results to be generated in future. Such duplicates may be acceptable in some scenarios, but can also optionally be removed by maintaining the following local invariant on the STAIRs.

Invariant: *A STAIR never contains two tuples* $\mathbf{t_1} \in \mathbf{T_1}$ *and* $\mathbf{t_2} \in \mathbf{T_2}$*, such that* $\mathbf{t_1}$ *and* $\mathbf{t_2}$ *match on all base-table components corresponding to the relations in* $\mathbf{T_1} \cap \mathbf{T_2}$.

In Appendix A we discuss techniques to maintain this invariant during query execution, and also how it guarantees duplicate-free execution.

3.2 Lifting the Burden of History using STAIRs

As we discussed in Section 2.2, storing intermediate result tuples generated during query execution can have a significant impact on future query processing. In this section we show an example of how STAIRs can be used to manipulate the state stored within the join operators so that such prior decisions can, in effect, be reversed.

Figure 6 shows the state maintained inside the join operators at time τ for our example query. Let us say that, at this time, we have better knowledge of the future and we know that routing S^R_τ toward $R \bowtie S$ was a mistake, and will lead to sub-optimal query execution in future (say

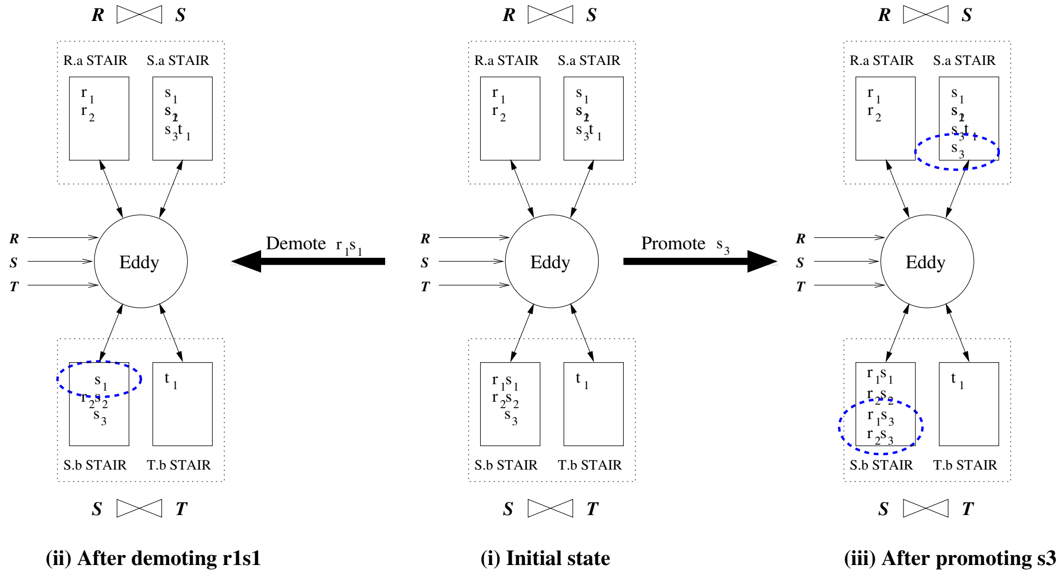

Figure 5: **Examples of promotion and demotion.** The middle plan (i) shows the state of the system after the eddy has received six tuples, $r_1, r_2, s_1, s_2, s_3, t_1$, and has chosen to route s_1 and s_2 to $R \bowtie S$, and s_3 to $S \bowtie T$. The left-hand plan (ii) shows the state after $r_1 s_1$ has been demoted to s_1 in $\mathcal{S}.b$. The right-hand plan (iii) shows the effect, starting from the initial state, of promoting s_3 using $S.a$.

because $R \bowtie S_\tau^R$ has high selectivity). This prior routing decision can be reversed as follows (Figure 6):

- Demote the $S_\tau^R \bowtie R_\tau$ tuples in $\mathcal{S}.b$ to S_τ^R.
- Promote the S_τ^R tuples from $\mathcal{S}.a$ to $S_\tau^R \bowtie T_\tau$.

Figure 6 shows the result of these operations. As we can see, the state now reflects what it would have been if S_τ^R had previously been routed to the $\mathcal{S}.b$ and $\mathcal{T}.b$ STAIRs, instead of $\mathcal{R}.a$ and $\mathcal{S}.a$ STAIRs. As a result, future R tuples will not be forced to join with S_τ^R.

We will refer to this process of moving state from one join operator to another as *state migration*.

3.3 Further motivating adaptive state management

Executing queries using STAIRs allows the eddy to adapt for reasons other than removing the burden of history. In this section, we will briefly discuss how this can be done.

Cyclic queries

Like the original eddy architecture, the base STAIRs architecture only works naturally with acyclic queries; as such, a spanning tree of cyclic query graphs must be chosen at the query initialization. However, unlike the EddyJoins architecture, the state management features provided by the STAIRs can be used to switch the spanning tree used for execution mid-way through query processing. Briefly, this is done by manipulating the state inside the operators to reflect query execution using the new spanning tree. In the interest of brevity we omit the details in this paper. We do note here that this process can involve fair amount of state movement, and if we expect to change the spanning tree used frequently, the SteMs architecture is an attractive alternative since it can change the spanning trees more easily.

Partial Results, Query Scrambling

When a data source in the query is relatively slow or even stalled, it is often desirable to ameliorate the delay by either producing partial results, or by aggressively joining together previously-received tuples (query scrambling). Both of these ideas involve a form of pre-computation of results. The *promotion* primitive provided by STAIRs can be used to perform such pre-computation as required. As an example, if the data from a relation R is delayed and the query engine wants to join data from relation S that is waiting for data from R (and hence, was routed to $R \bowtie S$ operator) with data from relation T, the eddy can use the promotion operation to move T tuples to $S \bowtie T$ operator, and thus perform useful work while waiting for data from R. The original query scrambling proposal [18] only addressed *initial* delays; the ability of STAIRs to allow precomputation even after some data from R generalizes that approach.

Flexible Storage and Reuse of Intermediate Results

The SteMs architecture demonstrated that it is possible to do query processing without storing any intermediate results at all. The state management primitives provided STAIRs enable the eddy to take this idea even further by providing flexibility in choosing the intermediate result tuples to be stored for further reuse. As an example, the eddy could choose to store only base-table tuples in the STAIRs by instantly demoting inserted tuples. This can be used to reduce the memory footprint of the query processor down to the memory footprint of the SteMs architecture (though the actual query execution will not be identical to using SteMs).

Another instance where such flexibility could be useful is when we are executing sliding window queries over streaming data [14, 3, 2, 7, 19]. Storing and reusing intermediate results is problematic in such a setting, because when a window on a base relation slides, some base-table

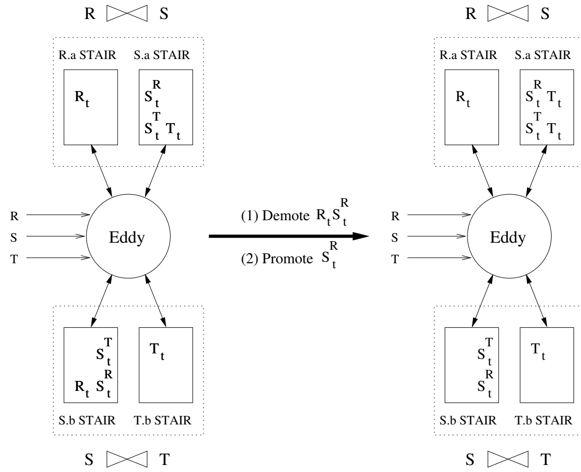

Figure 6: State Migration: Reversing prior decisions using STAIRs

tuples are dropped from the window, and all intermediate tuples that contain those base-table tuples must be removed from the engine. The cost of deletion can be quite significant, and many of these architectures choose to discard intermediate result tuples (e.g. CACQ [14]). STAIRs can be used to store only long-living intermediate tuples, thus enabling selective caching of intermediate tuples.

3.4 Correctness of Operation

Given the generality of the operations that can be performed on STAIRs, it is not clear that the eddy will always produce correct query result when using STAIRs. In Appendix A we prove the following:

Theorem 3.1 *An eddy with STAIRs always produces the correct query result in spite of arbitrary applications of the promotion and demotion operations.*

4 Implementation Details

We have implemented this architecture in the PostgreSQL 7.3 database management system in the context of the Telegraph project [2]. In this section, we will briefly discuss our implementation (please see [5] for the full details of the implementation). We have added two main new operators to the PostgreSQL code base:

Eddy: An eddy operator handles a single select-project-join query block that may contain arbitrary selections, joins and projections, but no aggregates, groupbys or subqueries. The latter constructs are handled by existing PostgreSQL operators that interact with the eddy operator using the traditional *iterator* interface. To be able to support such an interface, the eddy operator is implemented as a finite state machine that internally keeps track of its computational state. The eddy instantiates a set of operators (STAIRs, joins, SteMs *etc.*) as required, and executes the query by fetching tuples from the source modules (with which it interacts using the iterator interface as well) and by routing the tuples among the operators. The eddy also maintains statistics about the data, and uses them to make routing choices, and state migration decisions.

We developed the routing mechanism carefully to minimize the overheads of routing. The main idea behind these optimizations is to perform mini-batch routing instead of per-tuple routing. As we show in [5], the cost of making routing decisions can be effectively amortized over very few tuples, resulting in very low routing mechanism overheads with significant flexibility.

STAIR: We currently implement a STAIR as an in-memory hash table built on the tuples that have been inserted into the STAIR. It supports the four basic operations described above (*insert, probe, demote*, and *promote*), and also supports the iterator interface allowing it to be used in traditional query plans without eddies.

The STAIR operator is also used to implement two other operators:

- **Symmetric Hash Join:** As discussed previously, a symmetric hash join operator is simply two STAIRs used together.
- **SteM:** There are two main differences between SteMs and STAIRs: (1) SteMs only store base tuples, (2) a single SteM manages all inserts and probes on the base tuples corresponding to a single relation. We emulate this architecture by (1) inserting each base tuple into all the STAIRs on the corresponding relation, (2) never inserting an intermediate tuple into any of the STAIRs, and (3) routing a probe on a relation to the appropriate STAIR on that relation. It might seem that we pay a penalty relative to SteMs for maintaining multiple STAIRs on the same relation. However, as described in [16], a SteM actually maintains multiple internal hash-indexes, one corresponding to each probing attribute, and as such, the overheads of the two architectures are identical.

4.1 Routing Policy

The routing policy we use for the experiments in this paper is based on the rank ordering technique [13] for ordering selections. The eddy maintains two sets of statistics on the data: (1) the selectivities of the predicates on the base relations, and (2) the domain sizes of the join attributes corresponding to the joins, which are used to compute the join selectivities periodically. For SteMs, a greedy routing policy of routing a tuple such that it joins with the relation with the lowest join selectivity is used; this policy produces the minimal number of intermediate tuples in response to a new base tuple, and hence is optimal under that metric for SteMs (assuming that the selectivity estimates are accurate).

The same policy is however not optimal for EddyJoins or STAIR; the state accumulation through making locally optimal routing choices can result in sub-optimal overall performance. For these two techniques, we use the following routing policy to route a given tuple:

1. For each join (or corresponding STAIRs) that the tuple could be routed to, compute *cumulative* selectivity of joining it with the tuples from all relations that the

tuple is connected to through this join (e.g., while considering a R tuple and the $R \bowtie S$ join in our example query, cumulative selectivity is the result of joining this R tuple with $S_\tau \bowtie T_\tau$).

2. Route the tuple to the join with the lowest cumulative selectivity.

Note that, the computation of these selectivities is amortized using the mini-batch routing optimizations as we describe in detail in [5], and is not done per-tuple.

4.2 State Migration Policy

We currently use a greedy state migration policy that eagerly migrates state when routing decisions are changed. This approach ignores the cost of state migration, and could potentially result in sub-optimal performance for two reasons: the eddy might thrash by performing too many migrations in response to fluctuating selectivities, and the eddy might perform a large state migration at the end of query execution, with no subsequent payoff. As our experimental results demonstrate, these scenarios may be infrequent in practice. We are planning to address these shortcomings of the state migration policy by instituting a back-off mechanism to avoid thrashing, and by migrating state in stages to avoid large migrations at the end of a query.

5 Experimental Study

In this section, we present an experimental study that validates our approach. We begin by studying in detail an illustrative query on the TPC-H benchmark schema, to analyze the effect of state on the eddy architecture. We demonstrate the advantages of our architecture for both a completion-time metric, and an online delay metric that looks at the output times of tuples. We then present a more extensive set of experiments on an synthetic database modeled after the Wisconsin benchmark [6]. We use a set of randomly generated queries on this benchmark, and focus on the completion time metric to demonstrate the advantages of such adaptive query processing.

5.1 Setup

We ran the experiments on two machines, a 2GHz Pentium IV machine and a 1.4 GHz Pentium III machine with 512MB of memory each, running RedHat Linux. We present experiments comparing four query processing techniques: *Symmetric Hash Joins (SHJ)*, which uses traditional static query plans with symmetric hash join operators, and our three adaptive schemes: *EddyJoins*, *SteMs*, and *STAIRs*. Each experiment compared all four schemes on the same machine. The data was read off of local disks, and for the adaptive query processing techniques, the data was presented to the eddy through a module that controlled the input rates, and introduced delays as required by the setup. The “mini-batch” size for the eddy was set so that it could change routing decisions every 1000 tuples.

5.2 An Illustrative TPC-H Query

We begin with an illustrative query to validate the need for state management in adaptive query processing, and to understand the trade-offs involved in the state migration process. We use the following query on the TPC-H Benchmark schema that asks for *lineitem's* in a specified period and corresponding to a specified set of customers.

```
select *
from customer c, orders o, lineitem l
where c.custkey = o.custkey and
      o.orderkey = l.orderkey and
      c.nationkey = 1 and
      c.acctbal > 9000 and
      l.shipdate > date '1996-01-01'
```

The data from relation *lineitem* arrives at the query engine sorted in ascending order by *shipdate*. Early on in the query, the selectivity of the predicate on *lineitem* appears much lower than that of *customer*, so the eddy makes routing mistakes in the beginning by routing the tuples from the *orders* to the join with relation *lineitem*. Figure 7(i) shows the execution times of the various query processing techniques for this query in the scenario, where data rates are proportional to the sizes of the relations. This, *in effect*, gives the eddy correct estimates of the sizes of the relations, in spite of which, EddyJoins performs significantly worse than STAIRs because of the initial routing mistake of routing *orders* tuples to *lineitem* $\bowtie$ *orders*. On the other hand, STAIRs is able to correct this routing mistake by migrating the *orders* tuples to the second join when it gets more information, performing almost as well as having routed correctly from the beginning. This *switch* happens when the eddy has seen about 55% of the tuples from *orders*, and the corresponding state migration operation migrates all those tuples to the second join at that time.

Even though reuse of intermediate tuples is quite minimal in this query, SteMs perform poorly compared to the other techniques because there is no way to avoid joining the later-arriving *lineitem* tuples with *orders*; intermediate results from *customer* $\bowtie$ *orders* must be materialized to avoid this expense.

To better understand the trade-offs involved in state migration, we change the setup so that the eddy is not allowed to either change its routing decisions or perform state migrations until it has seen a certain number of *orders* tuples. Figure 7(ii) shows the results of this experiment. Since the change in the selectivities happens at about 55%, the execution times of these techniques are unaffected until that time. As we can see, the state migration cost up to this point is about 7% of the total execution cost with STAIRs. As we increase the number of *orders* tuples that need to be migrated, the cost of state migration increases, and its benefit goes down. Even then, the cost of state migration is low enough that STAIRs outperform EddyJoins, except at the very end when STAIRs migrate the entire *orders* relation without any benefit to doing so since the query execution is over.

Finally, we take a brief look at an interactive metric, namely, the rate at which output tuples are produced by these techniques. Figure 7(iii) shows the output rates for these four techniques (for SHJ, we show the output rates two plans, the best static plan and the worst static plan) for the above TPC-H query. As we can see, STAIRs pro-

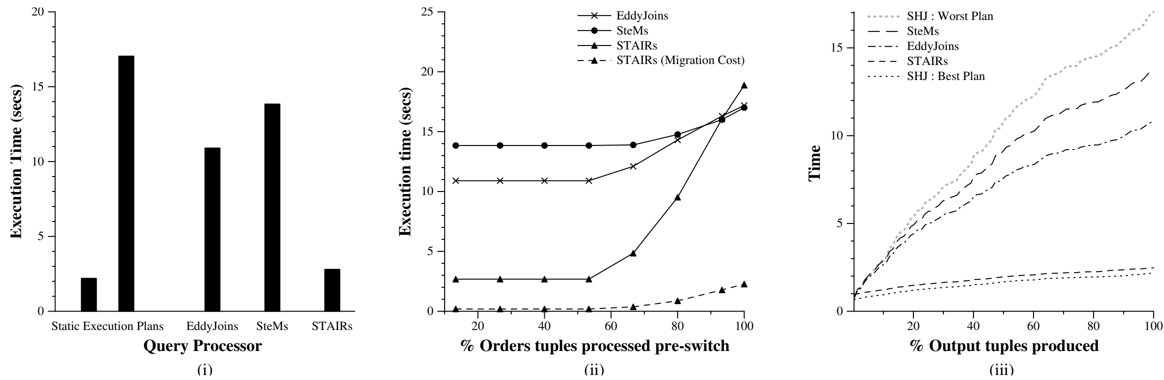

Figure 7: An illustrative TPC-H query: (i) The execution costs of the query processing techniques assuming data is streamed at rates proportional to the sizes of the relations (for SHJ, we show the costs for *two* static query execution plans, the best plan and the worst plan), (ii) Micro-benchmarking the state migration cost, (iii) Rates at which tuples are produced.

duce results at a much better rate except when the *best* static execution plan is used for SHJ, and even in that case, the performance of STAIRs is very close.

5.3 Synthetic Query Workload

To get a sense of benefits in a variety of different scenarios, and for different queries, we present results from a set of experiments over a synthetic dataset modeled after the Wisconsin Benchmark [6] using a randomly generated query workload. We populate the database with a total of 20 tables, five each of sizes 1k, 10k, 50k and 100k. Queries were generated randomly by first choosing a query graph shape from among four choices (path-shaped queries with 4 or 5 relations each, and star-shaped queries with 4 or 5 relations each), choosing the relations participating in the query randomly among the 20 tables above, choosing selection predicates with randomly chosen selectivities, and finally, choosing the selectivities of the joins randomly between 0 and 10. The initial plan for the all techniques was also chosen randomly among all the query plans for the query. Finally, we randomly varied the rates at which data from various relations was streamed, and also introduced random delays in the beginning.

We ran these experiments for 475 such setups, and measured the execution costs of the four techniques (SHJ, SteMs, EddyJoins, and STAIRs). In Figure 8 we present three graphs, comparing the relative performance of STAIRs with each of the other three techniques. In each data point, we plot the *ratio of the slower runtime to the faster*; when STAIRs is faster, the plot is *dark* gray. The x axis orders the experiments in ascending order of the benefit of STAIRs.

As we can see in Figure 8, STAIRs perform better than either SHJ or SteMs in almost all cases, with orders of magnitude difference in many cases. While comparing EddyJoins and STAIRs, we observe that, in about 66% of the cases, the execution costs of the two techniques differed by less than 10%, whereas out of the remaining 33% cases, STAIRs outperform EddyJoins in most cases, once by a factor of almost 10. In spite of using a greedy state migration policy which can result in late migrations and/or thrashing, STAIRs perform worse than EddyJoins in a very few cases, and at worst by a factor of 2.25. In future, we plan to address both these remaining problems with better migration policies as sketched in Section 4.2.

5.4 Adaptivity Benefits in a Traditional Setting

We close this section with an example over locally stored relations. The example illustrates the potential for "mixtures" of query plans to improve query performance even in traditional database scenarios, a direction we hope to pursue more deeply in future.

We use a 3-relation join query, $R \bowtie S \bowtie T$, over a synthetic dataset. The tables contains 50000 100-byte tuples each, and the selectivities of the two joins were set up such that, over the first 25000 tuples of the tables (ordered by the primary key of the table), the selectivity of $R \bowtie S$ was high (each S tuple joins with 5 R tuples), and the selectivity of $S \bowtie T$ was zero, whereas over the last 25000 tuples, the selectivities were reversed. Figure 9 shows the results of running this query for the variety of query processing architectures (including *Base*, the vanilla PostgreSQL query processor with its full range of join algorithms). We use a variation of the routing policy described in Section 4.1 for the adaptive query processing techniques, where the routing and state migration decisions are made based on the selectivities observed over a small window in the past. As we can see, both EddyJoins and STAIRs execute the query much faster than the best static plan for the query, with STAIRs performing slightly better because of the initial delay in adapting, during which the eddy makes bad routing choices.

The fundamental reason behind this is that the table S is naturally divided into two *partitions*, which exhibit very different join characteristics. Choosing the same query plan for both of them results in sub-optimal performance. Both EddyJoins and STAIRs, on the other hand, are able to identify and exploit this horizontal partitioning by routing part of the table through one query plan, and part of the table through another query plan. This example is simplified by

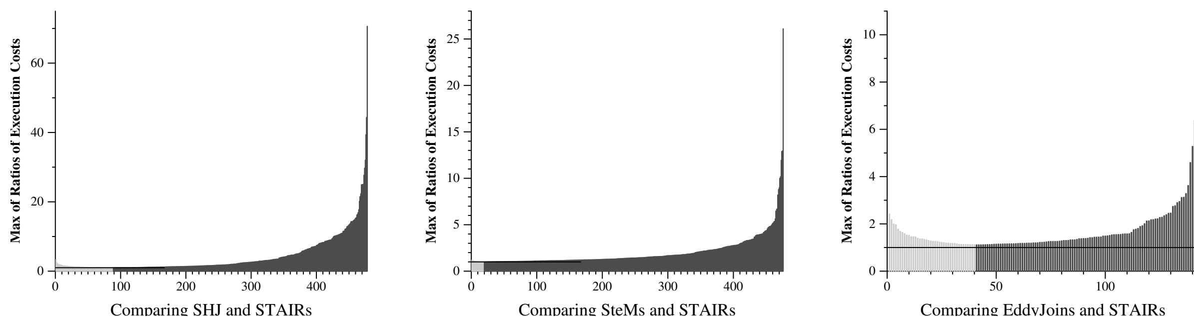

Figure 8: Comparing the execution costs of SHJ, SteMs and EddyJoins with STAIRs over 475 runs. The plots show the distribution of the ratios of execution costs, with gray bars on the left showing cases where STAIRs perform worse than the alternative technique. For the comparison of EddyJoins and STAIRs, we only plot results for the experiments where the execution costs differed by more than 10% (about 33% runs).

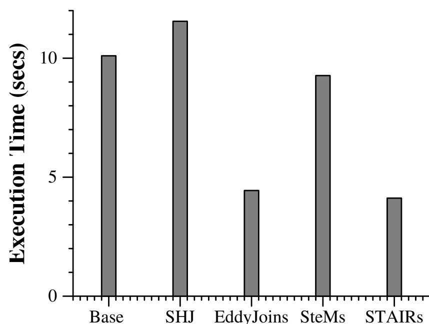

Figure 9: An experiment illustrating that the eddy architecture can perform much better than the *best* static plan when the data has natural *horizontal partitioning*.

the ordering of the relations. In future work we intend to explore techniques to learn content-sensitive routing strategies that can identify such partitions via tuple attributes rather than arrival order.

6 Related Work

There has been much work on adapting join and selection ordering during query execution. Due to lack of space, we will briefly discuss only the most relevant work here - for a detailed survey, please see [8].

Kabra and DeWitt [12] propose mid-query reoptimization, where a running query is reoptimized after every blocking point in the query plan. Tukwila [11] uses a similar technique, where in the absence of enough information about the data, only partially-complete plans are built in the beginning. Query scrambling [18] reacts to delays in the arrival of sources by rescheduling operators mid-flight, and in some cases, by reoptimizing the query to enable other operators. Most of these techniques, however, can not change the order of in-flight joins. Convergent query processing [10] proposes changing the query execution plan (and the join order) in-flight in response to changing runtime conditions. The times at which the query plans were changed divide the query execution in *phases*, and the *inter-phase* query results are generated at the end of the query execution, using the optimal plan for the query. In contrast, the eddy architecture produces all query results as soon as they are available. However, in absence of *state migration*, eddies can result in sub-optimal performance because of the query plans it is forced to use as a result of accumulated state.

Our work builds on the earlier work on eddies [1] and SteMs [16]. The eddy architecture has since been extended to execute continuous queries over streaming data [14, 3, 2]. These continuous query engines are based primarily on the SteMs architecture, and hence, do not reuse intermediate results generated during query processing. We believe that reusing intermediate results using STAIRs can result in better performance in data-stream environments as well, and we plan to work on this in future.

7 Conclusions

In this paper, we focused on the effect of the *burden on history* on the effectiveness of eddies, a highly adaptive query processing technique. Despite the ability to make mid-flight corrections to the query plan chosen to execute a query, the state that gets accumulated in the query operators can significantly constrain the ability of an eddy to adapt, resulting in sub-optimal performance in many cases. To alleviate this problem, we propose STAIRs, a modified eddy architecture that exposes the state accumulated inside the operators to the eddy, and provides state management primitives to manipulate this state. Our implementation of this architecture in a full-function database management system (PostgreSQL 7.3) demonstrates the viability of our architecture, even in traditional query processing applications.

References

[1] Ron Avnur and Joe Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

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

[3] Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

[4] Richard Cole. A decision theoretic cost model for dynamic plans. *IEEE Data Engineering Bulletin*, 2000.

[5] Amol Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, March 2004.

[6] David J. DeWitt. The Wisconsin Benchmark: Past, present, and future. In *The Benchmark Handbook Database and Transaction Systems (2nd Edition)*. 1993.

[7] Lukasz Golab and M. Tamer Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, 2003.

[8] Joe Hellerstein *et al.* Adaptive query processing: Technology in evolution. *IEEE Database Engineering Bulletin*, June 2000.

[9] Yannis Ioannidis and Younkyung Cha Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD*, 1990.

[10] Zachary Ives. *Efficient query processing for data integration*. PhD thesis, University of Washington, Seattle, 2002.

[11] Zachary G. Ives *et al.* An adaptive query execution system for data integration. In *SIGMOD*, 1999.

[12] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[13] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *VLDB*, 1986.

[14] Sam Madden, Mehul Shah, Joe Hellerstein, and Vijayshankar Raman. Continously adaptive continous queries over streams. In *SIGMOD*, 2002.

[15] Rajeev Motwani *et al.* Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[16] Vijayshankar Raman, Amol Deshpande, and Joe Hellerstein. Using state modules for adaptive query processing. In *ICDE*, 2003.

[17] Vijayshankar Raman and Joe Hellerstein. Partial results for online query processing. In *SIGMOD*, 2002.

[18] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD*, 1998.

[19] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, 2003.

A Proof of Correctness

Given the generality of the operations that can be performed on STAIRs, it is not clear that the eddy will produce the correct query result in all cases. In this section, we give a rigorous proof of correctness of this architecture. For simplicity, we will assume that when the eddy receives a new base tuple s from a relation S, it completely finishes processing s as well as all the intermediate tuples that result from joining s with existing tuples, before starting to process a new base tuple. We will also assume that the promotion and demotion operations are only applied at such times, also called *points of stasis*.

We begin with an invariant on the state maintained in the STAIRs operators at points of stases.

Let $\mathbf{C}$ be a connected subgraph of the query graph. As the query graph is acyclic, $\mathbf{C}$ induces a cut on the query graph as shown in Figure 10 (i). Let $X_i \bowtie Y_i$ denote the edge between $\mathbf{C}$ and $\mathbf{D_i}$. Furthermore, let $\mathbf{C}_\tau$ denote the result of joining the data for the relations in $\mathbf{C}$ that has arrived before time τ.

Invariant A: *At any point of stasis τ, for any connected subgraph $\mathbf{C}$ of the query graph, the state maintained in the STAIR operators satisfies the following property: for every tuple $\mathbf{c} \in \mathbf{C}_\tau$, a sub-tuple of $\mathbf{c}$, $\mathbf{c}'$, is contained in some $X_i \bowtie Y_i$ (*ie., *$\mathbf{c}$ has been built into a STAIR on some X_i).*

We say that $\mathbf{c}$ is present as $\mathbf{c}'$ in join $X_i \bowtie Y_i$.

Intuitively, if this were not true, then if there were to exist a final result tuple $\mathbf{q}$ such that (1) $\mathbf{c}$ is a sub-tuple of $\mathbf{q}$, and (2) the final base component of $\mathbf{q}$ that arrives in the system is from a relation outside $\mathbf{C}$, then this final base component will not be able to join with $\mathbf{c}$ to produce $\mathbf{q}$ at all.

The invariant is clearly true in the beginning of the query (at $\tau = 0$), when $\mathbf{C}_\tau = \mathbf{C_0}$ is empty for all $\mathbf{X}$.

Lemma A.1 *If the invariant is true before applying the demotion operation, the invariant remains true after application of the operation.*

Proof: If $\mathbf{c} \in \mathbf{C}_\tau$ was present in a join $X_i \bowtie Y_i$ as $\mathbf{c}'$, and if we replace $\mathbf{c}'$ by a sub-tuple $\mathbf{c}''$, $\mathbf{c}$ is still present in $X_i \bowtie Y_i$.

Lemma A.2 *If the invariant is true before applying the promotion operation, the invariant remains true after application of the operation.*

Proof: Let $\mathbf{c} \in \mathbf{C}_\tau$ be present in $X_1 \bowtie Y_1$ as a sub-tuple $\mathbf{c_1} \in (\mathbf{C_1})_\tau$, where $\mathbf{C_1} \subseteq \mathbf{C}$. $\mathbf{C_1}$, which itself is a connected subgraph, induces a cut on $\mathbf{C}$ as shown in Figure 10 (ii).

We will prove that promoting tuple $\mathbf{c_1}$ does not change the invariant for $\mathbf{c}$.

As $\mathbf{c_1} \in (\mathbf{C_1})_\tau$, the pair of STAIRs used for promoting $\mathbf{c_1}$ must correspond to a join that includes one relation from $\mathbf{c_1}$. There are two such sets of STAIRs.

- STAIRs corresponding to a join that includes a relation outside $\mathbf{C}$, and a relation inside $\mathbf{C_1}$. Say we use the STAIRs on the join $X_2 \bowtie Y_2$, where $X_2 \in \mathbf{C_1}$. In that case, during Step 2 of the promotion operation (cf. 3.1.1), $\mathbf{c_1}$ will be built into the STAIR on X_2 and the invariant remains true for $\mathbf{c}$, as $\mathbf{c}$ will now be present in $X_2 \bowtie Y_2$ as $\mathbf{c_1}$.
- STAIRs corresponding to a join on two relations in $\mathbf{C}$. Let us say we use the join between Z_1 and Z_2 for this purpose, where $Z_1 \in \mathbf{C_1}$ and $Z_2 \in \mathbf{C_3}$. Let $\mathbf{c_3} \in (\mathbf{C_3})_\tau$ be the projection of $\mathbf{c}$ on the relations in $\mathbf{C_3}$. Applying the invariant to $\mathbf{c_3}$, a sub-tuple of it (say $\mathbf{c_3'}$) must:
  - *Either* be built into the join between Z_1 and Z_2: in that case, when $\mathbf{c_1}$ is promoted using that join, the result $\mathbf{c_1 c_3'}$ will be built back into $X_1 \bowtie Y_1$ (Step 4, Section 3.1.1), and the invariant will remain true for $\mathbf{c}$, as $\mathbf{c_1 c_3'}$ is a sub-tuple of $\mathbf{c}$.

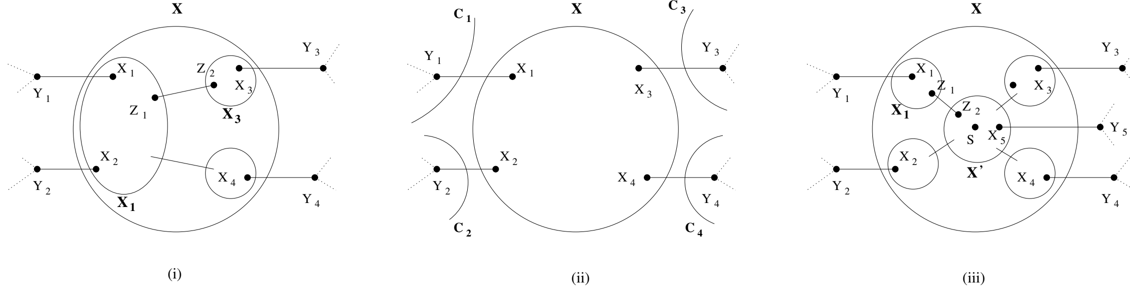

Figure 10: Proof of correctness figures

- *Or* be built into the join between some relation in $\mathbf{C_3}$ and some relation outside $\mathbf{C}$: in that case, the invariant will still be satisfied for $\mathbf{c}$, as $\mathbf{c'_3}$ is a sub-tuple of $\mathbf{c}$ built into a join that goes outside $\mathbf{C}$.

Lemma A.3 *If a new base tuple $s \in S$ arrives in the system at time τ, and if the invariant is true at time τ, then the invariant remains true after the eddy has finished processing s and all the intermediate tuples that it generates (at time τ').*

Proof: The invariant may only be affected for connected subgraphs of the query graph that contain S, and furthermore, only for those intermediate tuples that have the new tuple s as a base-table component. Let $\mathbf{C}$ be such a connected subgraph and let $\mathbf{c} \in \mathbf{C}_\tau$ be a tuple for which the invariant is not satisfied after the eddy has finished processing s. We have that s is sub-tuple of $\mathbf{c}$.

Let $\mathbf{c'}$ be the largest sub-tuple of $\mathbf{c}$ that was generated by the eddy when it processed s.

Clearly, $\mathbf{c'} \neq \mathbf{c}$. The only valid routing choices for $\mathbf{c}$, if it was generated during processing, are $X_i \bowtie Y_i$ operators, and it would have been built into one such join when the eddy routed it. Let $\mathbf{c'} \in \mathbf{C'}_\tau$, where $\mathbf{C'} \subset \mathbf{C}$. Once again, $\mathbf{C'}$ induces a cut on $\mathbf{C}$ as shown in Figure 10 (iii). Now, the valid routing choices for $\mathbf{c'}$ are:

- $X_5 \bowtie Y_5, X_5 \in \mathbf{C'}, Y_5 \notin \mathbf{C}$ (Figure 10 (iii)): If routed to this join, $\mathbf{c'}$, which is a sub-tuple of $\mathbf{c}$, would have been built into that join, satisfying the invariant for $\mathbf{c}$. As such, $\mathbf{c'}$ could not have been routed to such a join.
- $Z_1 \bowtie Z_2, Z_1 \in \mathbf{C'}, Z_2 \in \mathbf{C} - \mathbf{C'}$: Now, let $\mathbf{c_1} \in (\mathbf{C_1})_\tau$ be the intermediate tuple that contains base-table components of $\mathbf{c}$ corresponding to relations in $\mathbf{C_1}$. Note that, $\mathbf{c_1}$ does not contain s and as such, the invariant is true for this tuple. Hence, we get that, a sub-tuple $\mathbf{c_1}$, say $\mathbf{c'_1}$ (which is also a sub-tuple of $\mathbf{c}$), is:
  - *either* present in join $X_i \bowtie Y_i, X_i \in \mathbf{C_1}, Y_i \notin \mathbf{C}$: this is not possible as the invariant would have been satisfied for $\mathbf{c}$ in that case.
  - *or* present in the join $Z_1 \bowtie Z_2$: in that case, when the eddy routed $\mathbf{c'}$ to $Z_1 \bowtie Z_2$, it would have joined with $\mathbf{c'_1}$ to produce $\mathbf{c'c'_1}$, which would have been returned to the eddy. This contradicts our assumption that $\mathbf{c'}$ was the largest sub-tuple of $\mathbf{c}$ processed by the eddy, as $\mathbf{c'c'_1}$ is also a sub-tuple of $\mathbf{c}$.

Lemma A.4 *If a new base tuple $s \in S$ arrives in the system at time τ, and if the invariant is true at time τ, then the eddy produces the result of joining that tuple with all the tuples of the other relations that have already arrived,* ie.*, the eddy produces $s \bowtie (T_1)_\tau \bowtie \ldots \bowtie (T_l)_\tau$, where $T_1, \ldots, T_l$ denote the rest of the relations in the query.*

Proof: This follows from the above proof, by letting $\mathbf{C}$ be the entire query graph, and observing that, in that case, the only join $\mathbf{c'_1}$ can be present in, is $Z_1 \bowtie Z_2$, and thus contradicting the assumption that the largest sub-tuple of the result tuple $\mathbf{c}$ that was generated, was a strict sub-tuple of $\mathbf{c}$. ∎

From Lemmas A.1, A.2, A.3, and A.4, it follows that:

Theorem A.1 *The eddy always produces all the result tuples for acyclic queries inspite of arbitrary applications of the promotion and demotion operations.*

A.1 Duplicates

Though the above proof guarantees that all results for a query will be produced, it does not guarantee that every result will be produced exactly once. We avoid generating duplicate results by maintaining the following local invariant on the STAIRs at all times.

Invariant B: *A STAIR never contains two tuples $\mathbf{t_1} \in \mathbf{T_1}$ and $\mathbf{t_2} \in \mathbf{T_2}$, such that, $\mathbf{t_1}$ and $\mathbf{t_2}$ agree on all base-table components corresponding to the relations in $\mathbf{T_1} \cap \mathbf{T_2}$.*

As an example, at time τ, $\mathcal{S}.a$ is not allowed to contain both $sab, s \in S_\tau, a \in A_\tau, b \in B_\tau$ and $sac, s \in S_\tau, a \in A_\tau, E \in E_\tau$, as the two tuples agree on the common base-table components s and a.

Due to lack of space, we will state the following theorem without proof:

Theorem A.2 *If the above invariant is true when a new tuple enters the system, no duplicate results will be generated, and the invariant will remain true after processing that tuple to completion.*

As the duplicate avoidance invariant remains true after processing a new tuple, we only need to explicitly enforce it after a state management operation (which we do explicitly using a sorting-based algorithm). The following theorem completes our proof of correctness:

Theorem A.3 *If* **Invariant A** *is true and the eddy manipulates the state to enforce* **Invariant B**, **Invariant A** *will remain true after the operation.*