Using State Modules for Adaptive Query Processing[*]

Vijayshankar Raman
IBM Almaden Research Center
rshankar@almaden.ibm.com

Amol Deshpande Joseph M. Hellerstein
University of California, Berkeley
{amol,jmh}@cs.berkeley.edu

Abstract

We present a query architecture in which join operators are decomposed into their constituent data structures (State Modules, or SteMs), and dataflow among these SteMs is managed adaptively by an eddy routing operator [2]. Breaking the encapsulation of joins serves two purposes. First, it allows the eddy to observe multiple physical operations embedded in a join algorithm, allowing for better calibration and control of these operations. Second, the SteM on a relation serves as a shared materialization point, enabling multiple competing access methods to share results, which can be leveraged by multiple competing join algorithms. Our architecture extends prior work significantly, allowing continuously adaptive decisions for most major aspects of traditional query optimization: choice of access methods and join algorithms, ordering of operators, and choice of a query spanning tree.

SteMs introduce significant routing flexibility to the eddy, enabling more opportunities for adaptation, but also introducing the possibility of incorrect query results. We present constraints on eddy routing through SteMs that ensure correctness while preserving a great deal of flexibility. We also demonstrate the benefits of our architecture via experiments in the Telegraph dataflow system [26]. We show that even a simple routing policy allows significant flexibility in adaptation, including novel effects like automatic "hybridization" of multiple algorithms for a single join.

1. Introduction

It is often difficult to predict values of the parameters that govern database query execution. Cardinality estimates are highly imprecise [25, 3], and competing demands on memory, system load, and network bandwidth are typically known only at runtime [19, 32]. In federated and web database systems, data distributions and rates often cannot be known in advance [31, 27]. Other environments with volatile parameters include continuous query systems [17] and interactive systems [11].

Such uncertainties have led to a focus on adaptive execution in many recent query systems, including Tukwila, Telegraph, Aurora, and Query Scrambling [14, 26, 4, 29]. Perhaps the most adaptive of these approaches is the eddy operator [2] of Telegraph, which executes queries by routing tuples between query modules such as selections and joins, dynamically reconsidering the ordering of such modules on a per-tuple basis.

This paper presents an adaptation mechanism that substantially enhances the power of the eddy, allowing continuously adaptive decisions for most of the major aspects of traditional query optimization: not only the ordering of operators, but also the choice of access methods, join algorithms, and the selection of a spanning tree in the query graph [13, 16]. Our core idea is to refine the granularity of query modules, by breaking up join modules and elevating the data structures typically encapsulated within them into separate State Modules (SteMs).

The Join is a logical construct in the relational algebra; join algorithms typically involve multiple physical operations. The motivation behind splitting joins into SteMs is to decouple the physical operations that are typically encapsulated within join modules. This exposes these physical operations directly to the eddy, for performance calibration, fine-grain routing adaptation, and work sharing.

Informally, a SteM is a half-join. It encapsulates a dictionary data structure over tuples from a table, and handles *build* (insert) and *probe* (lookup) requests on that dictionary. We show that all select-project-join queries can be executed by routing tuples carefully between access methods on data sources, SteMs, and selections. Join algorithms are *not explicitly programmed*, but are instead captured in the routing of tuples between SteMs and access methods on the sources.

The breaking of algebraic join encapsulation has two benefits. First, the eddy can now monitor and control physical operations that are normally hidden within joins. By adapting the routing of tuples to the SteMs, the eddy adapts the order of these physical operations, and thereby the join algorithm itself. We will see an example in Section 4.2 where this allows the eddy to distinguish between cached and uncached lookups in a networked index join, resulting in a simple routing policy with better performance than the corresponding join algorithm from the literature. In fact, by appropriate routing the eddy can even simulate *hybrid* join algorithms that combine elements of different traditional algorithms. For example, we shall see an experiment in Sec-

[*]This work was supported by the NSF under Grants 0122599 and 0208588, a UC MICRO grant, a Microsoft Fellowship and gifts from Microsoft, IBM and Intel. Infrastructure for the research was provided by NSF grant EIA-9802069.

tion 4.3 where the eddy "hybridizes" index and hash join algorithms, gradually converting one into the other during query execution.

Second, SteMs provide a shared data structure for materializing and probing the data accessed from a given table, regardless of the number of access methods or join algorithms involving that table. This sharing is especially useful for access method adaptation. The choice of access methods is difficult in federated systems [26, 10, 14], because a given table may be provided by multiple data sources, and a single source may support multiple access methods corresponding to different sets of bind-fields. An eddy can run multiple access methods *concurrently*, and dynamically choose among them based on observed performance. The use of SteMs helps avoid redundant work during this competition; all access methods on a table build into the same SteM. Moreover, although an eddy routing policy can effectively try out multiple competing join algorithms, all lookups on a table probe the same SteM, taking advantage of the shared materialization.

The flexibility enabled by SteMs comes with a challenge: arbitrary routing from multiple access methods through SteMs may not correspond to a valid query execution plan. Incorrect routing can lead to duplicate results, missing results, or infinite routing loops. Therefore we develop a set of constraints on the routing that guarantee correct query execution (Section 3), while preserving opportunities for the flexible kinds of adaptation described above.

1.1. An Example

Consider a join of three tables R, S, and T, with equi-join predicates between R–S and S–T. Suppose there is a scan access method on each relation, and an index access method on T corresponding to the join attributes with S. Figure 1 shows three ways of running this query. Figure 1(a) is a traditional, statically chosen query plan involving a hash join and an index join. Figure 1(b) shows the approach of [2] where an eddy is used to dynamically adapt the join order by controlling the tuple flow between the joins. Note that both these approaches make use of only the index access method on T, and pre-chosen implementations for the RS and ST joins. Figure 1(c) shows the same query being executed with SteMs. All access methods over data sources are used simultaneously. Tuples coming into the eddy from these access methods are not routed to joins, but instead to SteMs and other access methods. This plan allows use of all the access methods, and a variety of routing decisions that correspond to different join algorithms and join orders. We develop the details of this approach in the body of the paper.

1.2. Background

The setting for this work is Telegraph, an adaptive dataflow system for querying streams of networked data [26]. An early application of Telegraph was Federated Facts and Figures (FFF), a query system to combine data from diverse and distributed data sources. These include not only relational databases but also websites providing services and data backed by databases (the so-called "Deep Web"). Among the factors that we discussed earlier, our interest in adaptive query execution is motivated by two unpredictable properties in FFF:

Volatility of distributed data sources: Since Web sources are autonomously maintained, their speeds and availability are hard to estimate at optimization time, and could vary during query execution.

Volatility of user interests during online query processing: Since users often specify queries in an iterative, exploratory fashion, FFF uses an online performance metric [11, 22] and gives out partial results during query execution. As the user sees these partial results, their interests in different parts of the result may change.

1.3. Outline of the paper

In the rest of the paper, we develop the SteM mechanism, and show how it helps in an environment like FFF (for a more elaborate presentation, please see [21]). We begin with a description of the modules in our architecture (Section 2), and then describe how arbitrary select-project-join queries can be executed *correctly* using these modules (Section 3). Next, we present an experimental study that illustrates the various kinds of adaptations allowed by SteMs, and the performance benefits we get under an online query processing metric (Section 4). We discuss related work in Section 5, and conclude with a discussion of other implications of SteMs and directions for future work (Section 6).

2. Mechanics of Query Execution with SteMs

In this section, we first describe the modules in our architecture, including the State Modules (SteMs), and discuss how they are instantiated for an arbitrary query. We then illustrate a simple but important example of these modules in use: an n-ary version of the *symmetric hash join* operator.

2.1. Eddy, State, Access, and Selection Modules

Our architecture uses four kinds of modules: (1) selection modules that correspond to query predicates, (2) access modules that correspond to access methods over data sources, (3) State Modules (SteMs) that encapsulate data structures used in traditional join algorithms, and finally (4) an eddy module that routes tuples between the other modules[1]. Each module runs asynchronously in a separate thread, though this asynchrony can also be achieved in a

[1] We assume that projections are done by each module as early as possible, and that Group By, Aggregation, and complex SELECT-list expressions are implemented above the eddy, before results are output to the user.

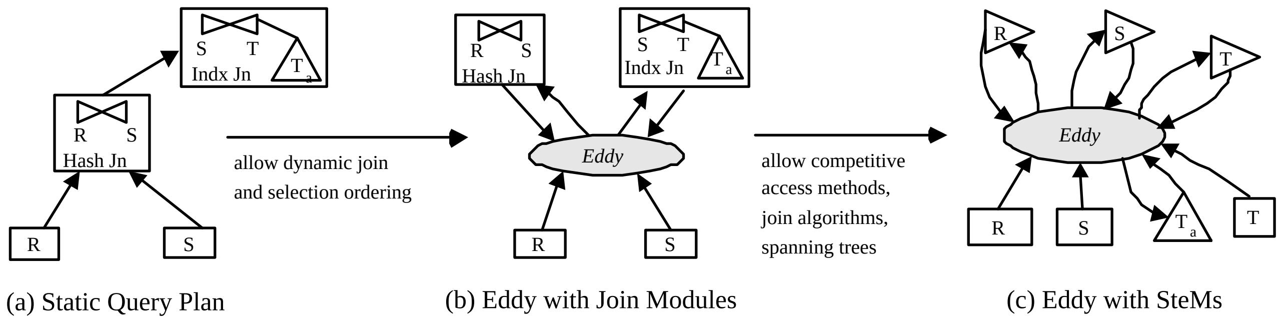

Figure 1. A three table join performed in three ways. The eddy continually routes tuples between modules, which run as concurrent threads. Indexes are represented by triangles, and shown encapsulated within index join as suggested in [2]. SteMs are shown as sideways triangles (to signify that they are half-joins).

single-threaded implementation [24]. We now describe the module functionality in detail. Simplified pseudo-code is given in Table 1. We start with some definitions.

Definition 1 (Base-table Component, Span) *Consider a tuple t belonging to the join of k base-tables* $T_1, T_2, \ldots, T_k$. *The projections of* $\{t\}$ *on the columns from each of these base-tables form relations with a single row each. Each of these rows are called the* base-table components, $t_{T_1}, t_{T_2}, \ldots, t_{T_k}$, *of t. We denote t by* $\langle t_{T_1}, t_{T_2}, \ldots, t_{T_k}\rangle$, *and say that t* spans *the tables* $T_1, T_2, \ldots, T_k$.

Definition 2 (Singleton tuple) *A singleton tuple is one that contains a single base-table component.*

2.1.1 Eddy Module

The eddy's role is to continuously route tuples among the rest of the modules, according to a *routing policy*. When a module other than the eddy processes a tuple t, it can generate other tuples and send them back to the eddy, for further routing. It can also optionally return (or *bounce back*) t to the eddy if t requires additional processing. A tuple is removed from the eddy's dataflow and sent to the output if it spans all base tables and is verified to pass all predicates. The eddy terminates the query when there are no tuples in the dataflow, and each module has finished processing all the tuples sent to it.

Each tuple also carries some state with it, called its *TupleState*, to track the work it has done in furthering query progress. The exact structure of TupleState depends on the routing policy. However, as a bare minimum, the TupleState must contain (a) the tables spanned by the tuple, and (b) the predicates that the tuple has passed (our implementation uses a bitmap, like the *done* bits of [2]). The former denotes the type of the tuple, and the latter is used by the eddy to decide when the tuple is ready for output. In fact, this state alone suffices for all but one special class of *cyclic* queries; we will discuss that exception in Section 3.4.

2.1.2 Selection Modules (SMs)

Selection modules (SMs) are simple. When a selection module M receives an input tuple t, it returns t to the eddy if t passes the selection predicate, and removes it from the dataflow otherwise. To track the progress made by t, if t passes the predicate, M marks this fact in t's TupleState.

2.1.3 Access Modules (AMs)

An Access Module (AM) encapsulates a *single* access method over a data source – it can either be a scan, or an index on some set of columns. Each access method on a given relation is encapsulated in a separate AM.

A tuple t that is routed to an AM is called a *probe tuple*, and corresponds to a request for the AM to output tuples that "match" the probe tuple – the matches from an AM on table S are all $s \in S$ such that the concatenation of t and s satisfies all query predicates that are defined over the union of the columns spanned by s and t[2]. As in traditional database access methods, the output schema of an AM is the same as that of the data source. In particular, the AM does not concatenate the probe tuple to its output tuples. Such concatenation will be performed only by SteMs.

Scans are also treated as AMs, but only accept a special empty probe tuple we call a *seed tuple*, and in return, output all tuples in their data source. These are initialized by passing seed tuples to them at query initialization time.

In addition to returning matches, AMs asynchronously bounce back each probe tuple t to the eddy. Intuitively the bounce back is required because the probe tuple is needed later for eventual concatenation with each of its matches. This is discussed in more detail in Section 3.3.

Asynchronous Indexes and EOTs: As demonstrated in [8], the throughput of accesses to Web sources can be improved significantly by sending multiple asynchronous probes; similar arguments can be made about asynchronous random disk I/Os. In this spirit, we assume that all AM

[2]Some of these predicates will be enforced by the index lookup; the AM applies the others after the lookup.

Module	Input tuple	Output tuple(s)	Action
SM	t	t or nothing	Bounce back t iff it matches predicate
AM	t	t	Asynchronously bounce back t
		matches for t	Asynchronously return all matches for t
		EOT	Return EOT after all matches have been returned.
SteM	$build_t$	—	Build $build_t$ into the SteM.
	EOT	—	Build EOT into the SteM.
		$build_t$ or nothing	Asynchronously bounce back $build_t$ if needed for correctness (Section 3).
	$probe_t$	—	Find matches for $probe_t$ among tuples in SteM.
		concatenated results	Concatenate these matches with $probe_t$ and return concatenated results.
		$probe_t$ or nothing	Asynchronously bounce back $probe_t$ if needed for correctness (Section 3).

Table 1. Functionality of the main query processing modules in our architecture.

probes and responses are asynchronous. This asynchrony complicates issues somewhat, because the system needs to track when all matches have been returned for a given probe. We use the dataflow itself to pass this information. When an AM on a table T has returned all matches to a probe, it sends an *End-Of-Transmission (EOT) tuple* encoding the probing predicate (in the case of a scan AM, the predicate is simply "true"). In the common case of index lookups using equality predicates, the EOT tuple is a regular tuple with a special *EOT* value in all the non-bound fields (*e.g.,* ⟨ *15 John EOT EOT* … ⟩ if the probe tuple binds the first two fields to 15 and John). For non-equality predicates, the EOT tuple contains pointers to the predicates, which are stored in a data structure created during query parsing. The advantage of encoding EOTs as tuples rather than as control messages is that the EOTs can be stored in SteMs itself, alongside standard tuples, as we will see below.

2.1.4 State Modules (SteMs)

A SteM essentially corresponds to half of a traditional join operator. It stores homogeneous tuples (tuples spanning the same set of tables) formed during query processing, and supports insert (build), search (probe), and optionally delete (eviction) operations. In this paper, we only consider SteMs over base tables; *i.e.,* all tuples in a SteM are singleton tuples from the same table. As such, all joins on a given base table can and do use the same SteM for builds and probes involving that base table. For this purpose, we allow a SteM to perform searches on arbitrary predicates.

Two kinds of tuples can be routed to a SteM. When a *build tuple* $t \in T$ is routed to a SteM on T (called $SteM_T$), t is added to the set of tuples in $SteM_T$, and auxiliary data structures (such as indexes), if any, are updated accordingly. An EOT tuple from an AM on T is also routed as a build tuple to $SteM_T$. When a *probe tuple* p is routed to $SteM_T$, $SteM_T$ returns *concatenated matches* for it to the eddy. These concatenated matches are all tuples in $\{p\} \bowtie SteM_T$ that satisfy all query predicates that can be evaluated on the columns in p and T.

Note that since the SteM is continually being built, it may not have all the tuples in $\pi_T(\{p\} \bowtie T)$. This is tracked by the presence of EOT tuples. If an EOT tuple in $SteM_T$ matches a probe p, then $SteM_T$ knows that it definitely contains all matches for a probe p. If not, the SteM might have to bounce back p so that it can be routed to other modules (to find the missing matches).[3] The logic for when such bounce backs are needed is determined by the routing constraints, and will be developed in Section 3.

In our present implementation, we speed up join predicate lookups through indexes. A SteM on a table T has one main-memory index (hash table or binary tree) on each column of T that is involved in a join predicate. These are all secondary indexes having pointers to the same tuples in memory. We do not focus on disk-resident indexes in this paper because the datasets we have encountered in Web sources are typically small enough to fit in main memory. We defer discussion of multi-table SteMs and disk data management within SteMs to Section 6.

2.2. Query Planning

The use of an eddy and SteMs obviates the need for query optimization because there are no *a priori* decisions to be made. Unlike in [2], there is no need even for a "pre-optimizer" that chooses the join implementations, access methods, and query spanning tree. A query is instantiated as follows :

1. Check that the query is valid, *i.e.,* it can be executed given the bind-field constraints on the data sources (we use the algorithm from Nail [18]).
2. Create an AM on each access method that can possibly be used in the query.
3. Create a SM on each predicate in the query.
4. Create a SteM on each base table in the query.
5. Create seed tuples as needed for scans (Section 2.1.3).

As described in the earlier section, only one SteM is created per data source. This SteM is shared not only among the join predicates involving that data source, but also among multiple instances of the source in the FROM clause, if any exist (*e.g.,* a self-join).

[3]This assumes that tuples from an AM arrive at the SteM in order; in the full paper [21], we describe techniques to handle out-of-order arrivals.

Though this paper focuses on execution of a single query, a SteM can also be used to share work and storage across concurrent queries. Related work in Telegraph on continuous query processing uses SteMs in this way [17, 5].

2.3. Example: An N-way Symmetric Hash Join

We now give an example of how these modules can be used to implement an n-way version of the symmetric hash join (SHJ) [23, 30]. The traditional, binary SHJ is a pipelining join that works by simultaneously building hash tables on both its inputs. Each input tuple is first built into a hash table on that input, and then immediately probed into a hash table on the other input. Due to its pipelining nature this operator is well-suited for interactive processing. Though originally designed as a memory-resident algorithm, it has subsequently been extended by [14] and [27] to spill to disk in memory-constrained environments.

There are two ways to extend the SHJ to multi-table queries. Consider an equi-join $R \bowtie_a S \bowtie_b T$.

Pipelining Binary Joins: Figure 2(i) shows how multiple binary SHJs can be pipelined to perform an n-way SHJ. To the best of our knowledge, this is the approach used in all current literature (*e.g.,* [28]).

n**-ary SHJ Operator:** Figure 2 (ii) shows how all the SHJs can be unified into a single operator that uses four hash indexes: one on R, one on T, and one on each join column of S (one of these is a secondary index). When a new R (T) tuple comes in, it is first built into the corresponding hash index H_{R_a} (H_{T_b}), and then probed into H_{S_a} (H_{S_b}). The resulting matches, if any, are then used to probe into H_{T_b} (H_{R_a}) and the result is output. When a new S tuple comes in, it is similarly built into H_{S_a} and H_{S_b}. At this point, we have a choice, corresponding to different join orders. We can either probe the S tuple into H_{R_a} and probe H_{T_b} with the resulting matches, or we can probe into H_{T_b} and then into H_{R_a}.

The initial eddy paper [2] was based on the first approach – by connecting a set of pipelining binary join modules to an external eddy module, the ordering of the join modules can be decided dynamically. In contrast, the SteMs mechanism is based on the second approach – it essentially places an eddy *within* the n-ary SHJ operator, so that the ordering of the hashtable lookups can be decided dynamically. This is the core effect of SteMs – to give the eddy access to the data structures typically stored inside join algorithms. However, the SteM approach is not implemented as part of the SHJ, and therefore becomes more generally applicable.

Figure 2 (iii) illustrates the translation from the unified n-ary SHJ operator to a routing through SteMs. We use a SteM on each source to encapsulate the hash indexes on that source, and an eddy to route tuples between the SteMs. Each tuple is first built into a SteM on its source, and then immediately routed to the other SteMs. The eddy can dynamically adapt the join ordering by changing the way it routes S tuples after it is built into $SteM_S$.

In addition to different routing opportunities, the n-ary hash join materializes different state than the traditional binary-SHJ scheme. Note that the n-way SHJ description above stores only singleton tuples in hash tables, whereas the traditional pipeline of binary SHJs materializes intermediate result tuples from joins below the root (*e.g.,* tuples in $R \bowtie_a S$). SteMs can in principle support either scheme, or both, via a SteM to materialize each base-table or intermediate relation desired. This represents a tradeoff of performance for memory space – less memory is likely to be used if intermediate result tuples are not stored, but more probes may need to be made since the same intermediate results may need to be recomputed multiple times. In this paper (as in [17, 5]), we choose not to store intermediate tuples in SteMs. In addition to the space/time tradeoffs, a secondary advantage of not materializing intermediate results is that tuple eviction is simplified. Each base-table component is stored in a single SteM, and so it can be easily evicted by the SteM if needed. Although not the focus of this paper, sliding-window queries and queries over unbounded data streams require tuple eviction, and [17, 5] both use SteMs with eviction. We are currently investigating a hybrid approach that partially materializes intermediate results to the extent of available memory (Section 6).

The n-ary SHJ can be used for any select-project-join query where all sources have scan access methods. In the next section, we generalize this simple operator to use other join algorithms as well as index access methods, and show how the eddy can dynamically adapt the join algorithms, access method choices, and spanning tree choices.

3. Executing Arbitrary Select-Project-Join Queries with SteMs

Superficially, query execution with SteMs is simple. We only need to instantiate the AMs, SteMs, and SMs, and let the eddy route tuples through these operators. Unfortunately, arbitrary routing policies may lead to incorrect results and non-terminating queries. Since we want the eddy to adapt the routing dynamically, we now develop *constraints* on the routing policy that will ensure correctness.

The n-ary SHJ operator corresponds to one correct routing policy. We start by identifying the routing constraints that are implicit in this operator, and gradually generalize these constraints to a larger space of execution strategies. Our presentation is intended to be intuitive and informal; more rigorous treatment of these issues, as well as proofs of correctness can be found in the full paper [21].

3.1. Acyclic SPJ queries with a Single Scan AM on each Table

The n-ary SJH is captured by two rules. The first is that the SteMs be implemented with hash indexes. The second is

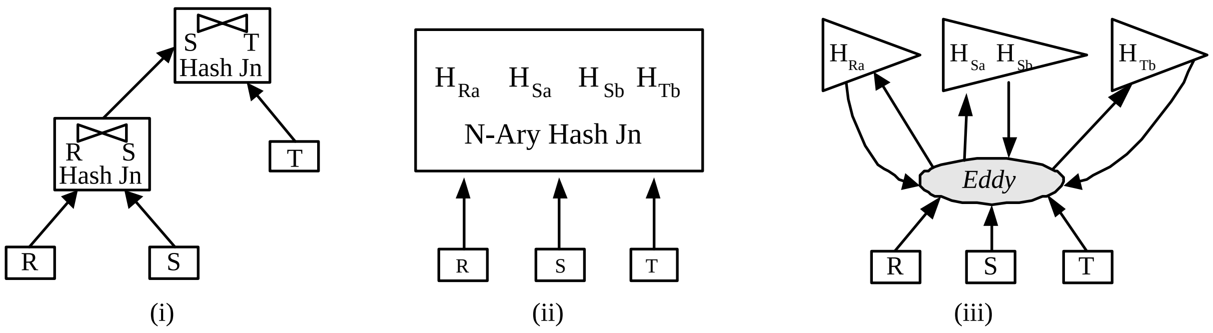

Figure 2. Three ways of doing a 3-table symmetric hash join (SHJ): (i) with pipelined binary SHJs, (ii) with a 3-ary SHJ operator, and (iii) with an eddy and SteMs

that the eddy must obey the following routing constraints:

BuildFirst: A singleton tuple from a table T must first be routed to build into $SteM_T$.

SteM BounceBack: A SteM must always bounce back build tuples (so that they can probe the other SteMs for matches), and never bounce back probe tuples.

Atomicity: The building of a singleton tuple into a SteM must be atomically coupled to the probing of that tuple into the other SteMs.

BoundedRepetition: No tuple must be routed to the same module more than once.

The first three constraints capture the essence of the n-ary SHJ, and BoundedRepetition ensures query termination. Two relaxations of these constraints allow the eddy to adapt over a much wider space of join algorithms.

Constraint Relaxation to allow other Join Algorithms

Our first relaxation removes the constraint that the SteMs must be implemented with hash indexes. For example, the SteM may use a linked list when it holds a small number of tuples, and switch to a hash-based implementation when the list size increases. This switch can be made independent of other modules.

Our next relaxation is to remove the Atomicity constraint, and *decouple* the build and probe operations of each tuple. This allows the eddy to interleave probes and builds of tuples in arbitrary ways, and thereby change join algorithms (in Section 3.5, we will relax this further by allowing the build to be completely avoided). Unfortunately, this build-probe decoupling can cause duplicate query results. For example, Figure 3 shows four steps in a SHJ. If $\langle r_1, s_1 \rangle$ satisfies the join predicate, $\langle r_1, s_1 \rangle$ is output at both steps 3 and 4 because the builds and probes of r_1 and s_1 tuples are interleaved. To avoid such duplicates, we add a TimeStamp constraint [20], to form the following set of constraints:

BuildFirst, SteM BounceBack, BoundedRepetition: These constraints remain unchanged from above.

TimeStamp:

- Each singleton tuple t is assigned a global, monotonically increasing *Timestamp* $TS(t)$ when it builds into a SteM. Before building, $TS(t)$ is defined to be ∞. For other tuples $TS(\langle t_1, \ldots, t_n \rangle)$ is defined to be $\max(TS(t_1), \ldots, TS(t_n))$ *i.e.,* the timestamp of its last arriving base-table component.
- When a tuple r probes into a SteM and finds a match s, the result $\langle r, s \rangle$ is returned to the eddy iff $TS(r) > TS(s)$.

The TimeStamp constraint ensures that only the last arriving base-table component of a result tuple will generate that tuple, by probing into other SteMs to join with previously-arrived components.

Simulating and Hybridizing Non-Pipelined Join Algorithms

These relaxed constraints allow the eddy to simulate several join algorithms besides the SHJ. Consider a two table join of R and S. The following sequence of steps can simulate many non-pipelining join algorithms:

1. Route all R tuples to build into $SteM_R$
2. Route all S tuples to build into $SteM_S$
3. Route all S tuples to probe into $SteM_R$
4. Route all R tuples to probe into $SteM_S$

The SteM implementation decides exactly which join algorithm will be simulated. *E.g.,* the following "asynchronous" hash index implementation simulates a *Grace Hash Join* [7]. While build tuples are routed to $SteM_R$ and $SteM_S$, the SteMs create hash partitions on disk. But instead of bouncing back these build tuples immediately, they do so asynchronously, *clustered by the hash partition*. Therefore in Step 3, when the bounced-back S tuples probe $SteM_R$, $SteM_R$ gets very good I/O locality. Because of the TimeStamp constraint, Step 4 does not produce any results. It can be completely avoided by maintaining in each SteM the minimum timestamp of all tuples the SteM contains – the eddy need only route to a SteM the probe tuples with timestamp greater than this minimum timestamp.

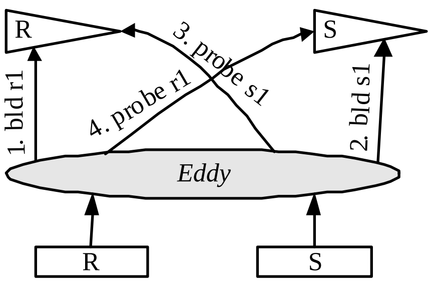

Figure 3. Duplicates arise because of decoupling build and probe of r_1

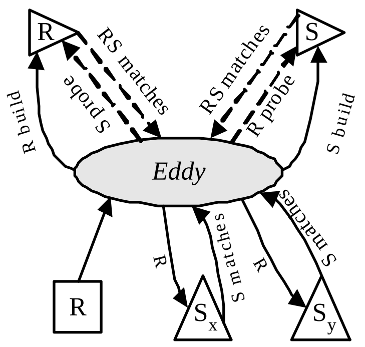

Figure 4. $R \bowtie S$ query with index AMs on S

It is unusual to describe Grace Hash Join in terms of a routing policy. But the advantage is that the eddy can now dynamically *hybridize between SHJ and Grace Hash Join*, by changing its routing as follows. Rather than do all of Steps 1 and 2 before Steps 3 and 4, the eddy can dynamically decide to interleave them. Specifically, when a tuple r is bounced back after building into $SteM_R$, the eddy may choose to immediately probe r into $SteM_S$. This choice is based on the level of interactivity desired. For instance, the eddy can start with frequent probes to give interactive responses early on, and later degenerate to occasional probes in order to reduce completion time (with infrequent probes, more probes for the same partition are clustered together, so I/O cost is lesser). The frequent probe phase simulates an SHJ, and the occasional probe phase is similar to Grace.

An exactly analogous implementation of SteMs with tournament trees that spill sorted runs to disk will simulate a Sort-Merge join. The Hybrid-Hash Join [6] is simulated if the SteMs maintain a full in-memory hash table on some of the partitions and bounce back build tuples for these partitions ahead of others. The eddy can then route S tuples from these in-memory partitions to probe into $SteM_R$ even before all S tuples have been built.

Note that one part of the join logic – choosing whether the indexes are hash indexes or tournament trees, and the spilling policies – is captured in the SteMs implementation. It is up to the SteM implementation to internally adapt this if needed. But the remaining part, i.e., the interleaving of builds and probes, is captured in the routing policy, and can be dynamically adapted by the eddy.

3.2. Competitive AMs

We now expand our class of queries to include those over tables with more than one AM. Such alternate AMs are very common for Web sources in Telegraph FFF. Different websites often provide the same data, and a single website may support multiple AMs corresponding to different sets of fields that can be chosen as the lookup key. We address tables with multiple scan AMs in this section, and discuss index AMs in the next section.

Traditional database systems typically pick one AM per data source at optimization time. We want to be able to run multiple AMs on a single source in competition with one another, and let the eddy dynamically choose one AM, or switch between AMs. For example, if a particular AM stalls because the underlying source is delayed, the eddy should be able to use the alternate AMs. In our architecture, this is quite straightforward to do since all the access methods are exposed to the eddy. The main problem turns out to be duplicates; the same tuple can be generated by different AMs. However because of the BuildFirst constraint, such duplicates can be easily removed when they build into the SteM on the source itself. We only need a simple enhancement to the SteM BounceBack constraint:

SteM BounceBack: A $SteM_S$ must bounce back a build tuple s unless it is a duplicate of another s' that is already in $SteM_S$.

Identifying and handling duplicates is not easy, especially with AMs over different, possibly inconsistent, Web sources. We currently adopt a *set* semantics, where a SteM removes any build tuple that is identical to another tuple already present in the SteM.

3.3. Index AMs

When a data source has an index AM, we encounter another problem. Figure 4 shows the execution of a simple two-table join query in this class. Recall that our indexes are allowed to return matches asynchronously. A tuple r from R is first built in $SteM_R$, and then probed into $SteM_S$ to see if matches for r have been already cached there. But unless *all* matches are already cached, *r must be bounced-back by $SteM_S$*, so that it can probe into one of the AMs on S. The difference from the previous section is that there is no scan AM on S, so r must probe into an index AM to seed the generation of its matches.

Subsequently, the index AMs on S will return matches for r, say s_1 and s_2. These matches will be first built into $SteM_S$ and then probed into $SteM_R$. It is only during this probe that s_1 and s_2 will join with r (and possibly with other

R tuples as well). Thus $SteM_R$'s role is as a *rendezvous buffer* [8] to hold pending probe tuples until matches arrive.

Since s_1 and s_2 are built into $SteM_S$, subsequent R tuples with the same bind column values as r will find index matches in $SteM_S$ itself. So $SteM_S$ will not bounce back these R tuples ($SteM_S$ verifies that it has *all* relevant matches by checking its EOT tuples). Thus $SteM_S$'s role is that of a cache on index lookups into S. In fact, when there are multiple AMs on a source, they all cooperate in building the same cache, and the work of probing alternate AMs is not wasted. This has the effect of reducing the overall cost of competition.

When a data source has both scan and index AMs, the tuple routing determines whether an index join is performed or a hash join is performed. We will see an experiment in Section 4.3 where the eddy dynamically adapts its routing to switch between the two during query execution.

To summarize, the enhanced SteM BounceBack constraint is as follows:

SteM BounceBack:

- A $SteM_S$ must bounce back a build tuple s unless it is a duplicate of another s' that was previously in $SteM_S$.
- A $SteM_S$ must bounce back a probe tuple r unless S has a scan AM or $SteM_S$ already has all matches for r.

3.4. Cyclic Queries

Cyclicity in the query join graph complicates matters still further. Traditionally, the plan chosen by the query optimizer contains join modules only over a spanning tree of the query join graph. This spanning tree is determined before query execution, even for prior adaptive query processing schemes like the initial Eddy paper [2]. Static spanning tree choices hurt in two ways:

- The spanning tree choice is typically made based on selectivities that are hard to estimate for Web sources.This can lead to arbitrarily sub-optimal execution strategies.
- A static spanning tree choice can also constrain the generation of partial query results. Consider a three way join of R, S, T where there are join predicates between each pair of tables. If we choose $R \bowtie S \bowtie T$ as the spanning tree and source S stalls during query execution, the entire query blocks. If the spanning tree could be changed dynamically, RT tuples could be generated. These partial results with missing values for S columns could be very valuable in interactive querying environments [22].

The problem with not fixing a spanning tree a priori is that duplicates can arise even after timestamping. Consider the following sequence of events in the above 3-way join query: (1) a tuple t probes into $SteM_S$ to find a match $\langle s,t\rangle$, (2) $\langle s,t\rangle$ probes into $SteM_R$ to find a match $\langle r,s,t\rangle$, (3) $SteM_S$ bounces back t as per the SteM BounceBack constraint, (4) t probes into $SteM_R$ to produce $\langle r,t\rangle$ which probes into $SteM_S$ to produce $\langle r,s,t\rangle$ again.[4]

To avoid such duplicates, we must ensure that previously bounced-back tuples (like t) cannot probe other SteMs.

ProbeCompletion Constraint: A tuple t that has been bounced back after probing into a $SteM_S$ must not probe into any other SteM afterwards. The routing policy must however maintain t in the dataflow, routing it to other modules, until it has been probed into an AM on S.

Definition 3 (Prior Probers, Probe Completion Tables) *Tuples like t that have been bounced back after probing into SteMs are called* prior probers*. The corresponding table S is called the* probe completion table *of t, and the AMs on S are called the* probe completion AMs *of t. The identity of the probe completion table is marked in the TupleState of t.*

3.5. Relaxing the BuildFirst Constraint

The constraints developed so far guarantee that all select-project-join queries will be executed correctly. But one of these constraints, the BuildFirst constraint, is particularly restrictive and could result in highly inefficient execution in situations where one of the input tables is much larger than the others. Suppose that the R table was much larger than both S and T tables in the example of Figure 2(iii). In that case, it might be better to build SteMs on the S and T tuples and probe the R tuples directly into these two SteMs, without building into $SteM_R$. This is equivalent to building a temporary index on only one side of the join.

We can enable such optimizations by allowing the eddy to *not build* a $SteM$ on a table R as long as there is only one access method on R and that access method is *scan*. If there multiple access methods on R or if there is an index AM on R, the $SteM$ is required to avoid duplicate results.

Now if an R tuple is bounced back from a $SteM_S$, it means that all S matches for this R tuple could not be found at that time. So this R tuple needs to routed back to $SteM_S$ to find the remaining matches. So we relax the BoundedRepetition constraint to allow the eddy to route a given tuple repeatedly to the same module. To ensure that these repeated probes do not produce duplicates, we assign every R tuple a *LastMatchTimeStamp*. This is initially set to 0. Every time the R tuple is routed to $SteM_S$, the $LastMatchTimeStamp$ is updated to the maximum of the timestamps of all tuples in $SteM_S$.

The constraints we have developed so far are summarized in Table 2. Notice that the SteM BounceBack and Timestamp rules are implemented internally to the AMs and SteMs, and the routing policy implementor need not be aware of them at all.

Theorem 1 (Duplicate Avoidance) *If the eddy follows a routing policy that satisfies the constraints of Table 2, dupli-*

[4]Note that this only happens if there is no scan AM on S, because otherwise $SteM_S$ does not bounce back the t tuples sent to it (Section 3.3).

Constraints to be enforced by Routing Policy Implementor	
BoundedRepetition	- No tuple can be routed to a given module more than a finite number of times.
BuildFirst	- A singleton tuple from a table T must first be built into $SteM_T$ *iff* T has multiple AMs *or*, T has an index AM
ProbeCompletion	- A prior prober t must not be routed to any SteM other than that on its probe completion table. - The eddy can remove a prior prober t from the dataflow only after t has been probed into one of t's probe completion AMs.

Constraints enforced within SteM and AM implementation	
SteM BounceBack	- A $SteM_S$ must bounce back a build tuple s unless it is a duplicate of another tuple s' that is already in $SteM_S$. - A $SteM_S$ must bounce back a probe tuple r unless $SteM_S$ already contains all matches for r, or S has a scan AM, and all base-tuple components of r have been cached in other SteMs
TimeStamp	- When a tuple r probes into a SteM and finds a match s, the result $\langle r, s\rangle$ is returned to the eddy iff $TS(r) > TS(s) > LastMatchTS(r)$.

Table 2. Routing constraints that ensure correct query execution

cate versions of a tuple will not arise in the dataflow, except for singleton tuples that have not yet been built into SteMs.

Theorem 2 (Correctness) *If the eddy follows a routing policy that satisfies the constraints of Table 2, it will not output any tuple that is not in the query result, and will output all query result tuples in a finite number of routing steps.*

Proofs of these correctness theorems can be found in the full paper [21].

4. Experimental Results

We now illustrate the kinds of adaptation that SteMs enable, through an experimental study. Our focus is on the online metric of maximizing the rate at which result tuples are generated, though some of the experiments also demonstrate the effectiveness of our system for the traditional metric of completion time. All our experiments are based on an implementation of SteMs in Telegraph [26], and were run on a lightly loaded machine with dual 666MHz Pentium-III processors and 768MB RAM, running Redhat Linux 6.0. The salient points of our experimental study are:

1. Even a simple join algorithm like the index join encapsulates multiple physical operations, and this causes a *head-of-line blocking* problem. This problem can be avoided by breaking the join module into SteMs.
2. SteMs allow the eddy to efficiently learn between competitive access methods, while doing almost no redundant work.
3. SteMs allow the eddy to dynamically choose the join spanning tree for cyclic queries.
4. SteMs allow the eddy to dynamically switch between an index join algorithm and a symmetric hash join algorithm during query execution.
5. With SteMs, the eddy can adaptively choose the way it reorders tuples in interactive environments.

We use synthetic data sources for our experiments so that the source properties can be easily controlled. The data sources that we use are as shown in Table 3.

Due to space constraints, we only report two experiments here (demonstrating points 1 and 4 above). Please see the full paper [21] for the complete set of experiments.

4.1. Eddy routing policy

Our implementation uses a routing policy designed to maximize the value of the partial results output to the user [22]. The details of this policy are not needed to understand the advantages of SteMs in our experiments. We briefly summarize it here for completeness.

When a tuple t with a TupleState T is routed to a module M, the benefit $B(t, M)$ is the value of the partial result that will be output by M. This benefit depends on the expected number of matches that M will return and the user's preferences for the matches[5]. M also takes an expected time $C(t, M)$ to process t. To maximize the value to the user over time, the eddy continually routes so as to maximize $B(t, M)/C(t, M)$. Clearly it is not feasible to do this optimization across all tuples. As discussed in [22], though, this ratio depends largely on M and the tuplestate T of t. So we only optimize at this granularity. To this policy we add the constraints of Section 3, specialized as follows:

BuildFirst: Singleton tuples are always first built into their corresponding SteMs, regardless of whether they come from sources with multiple AMs. This simplifies our implementation, and is inexpensive because Web sources typically have data sizes much smaller than memory sizes.

SteM BounceBack: In addition to the bounce back circumstances of Table 2, we set SteMs on tables with index AMs to also bounce back any probe tuple that satisfies a predicate prioritized by the user. Notice that in the case where a $SteM_S$ has both an index AM and a scan AM, this bounce back is redundant. But, if the pri-

[5]Even a tuple that does not contain the key columns of the result, and as such can not be output to the user, is still given a value because it can subsequently generate partial results by joining with other tuples.

Source	Schema	Description
R	{key: integer, a: integer}	R is a table with 1000 tuples, and has a scan access method. *key* is its primary key, and *a* is a field with 250 distinct values, randomly assigned.
S	{x: integer, y:integer}	S has two keys, *x* and *y*, and has asynchronous index access methods on both of them. All S tuples have identical values of *x* and *y*.
W	{key:integer}	W has an asynchronous index access method on its primary key *key*, and a scan access method.

Table 3. Data sources used in our experiments. Index lookups are implemented as sleeps of identical duration.

oritized probes are bounced back, they can subsequently probe into AM_S. This speeds up the entry of matches for these tuples into the dataflow and thereby the output of prioritized results to the user.

4.2. Index join improvement through SteMs

We start with an experiment that shows the effect of decoupling physical operations within a join. We use the index join algorithm to show this.
Consider the following query that joins tables R and S.

Q1 : SELECT * FROM R, S WHERE R.a = S.x

The join is an equi-join between S.x, a key column of S, and R.a. Table R has a total of 1000 tuples, with 250 distinct values of R.a. In a traditional query processor, this query will be executed using an index join module as shown in the Figure 5. In contrast, our system will use a SteM on R and S, a scan AM on R, and an index AM on S (Figure 6). $SteM_R$ holds the pending R probe tuples while AM_S processes the probes, and $SteM_S$ caches probe results.

Figure 7(i) plots the number of RS results output over time in the two schemes. The curve for the plan using the index join is parabolic, as expected. The cost of probing into the index join decreases continually over time as the cache size, and hence the probability of cache hits, increases. In contrast, the plan using the SteMs takes about the same time overall, but is almost linear in shape. It rises comparatively faster in the initial stages of the processing and as such, does better on our online processing metric.

To understand this behavior, we plot the number of probes into the remote source S, for the two approaches (Figure 7(ii)). Notice that these two curves are almost identical. Thus the lookup caches on S build up at the same rate in both cases. The difference is that with SteMs, the probes into the caches happen much more quickly.

In the first approach (without SteMs), every tuple coming out of the scan on R *does not immediately probe into the index join on* S. Since all queues between the eddy and the modules are finite in size, these probes can only happen at the speed of the index join, which in turn is bottlenecked by the speed at which the S index can handle R probes. This is unfortunate, because many of the R tuples may not need to probe into the S index at all – they may find matches in the S cache itself. With SteMs, this *"head-of-line blocking"* does not happen, because probes into the cache and the index have separate queues.

This experiment illustrates our point that even simple join operators encapsulate multiple physical operations. In this example, the index join comprises two operations, cache lookup and index lookup, that have different performance characteristics. These performance characteristics could also vary with time; *e.g.*, cache lookups may become expensive if the cache runs out of memory and starts paging to disk. Therefore it is important to avoid encapsulating such operations within the join modules.

4.3. Index/Hash join hybridization based on costs

Our next experiment studies the ability of our system to choose and hybridize among alternative join algorithms based on their costs. The query that we use for this experiment joins R with the table W that has both an index and scan access method.

Q4: SELECT * FROM R, W WHERE R.key = W.key

To ensure that our results are not affected by cache effects, we use an equijoin between the key columns of R and W. This means that there are two natural ways of joining R and W: symmetric hash join using scans on R and W, and index join utilizing the index on W. A third way is for the eddy to use both access methods on W, with SteMs on R and W, and choose a hybrid join algorithm.

Figure 8 (i) plots the number of result tuples generated over time in all these three approaches, during the first few seconds of the query execution. The index join initially outperforms the hash join because the W index outputs the exact matching W tuple for each R probe tuple, whereas the W scan outputs all W tuples in an arbitrary order – only some of the R probes find matches in the tuples scanned from W. The symmetric hash join however catches up with the index join quickly, as the R and W hash tables are filled. Figure 8 (ii) plots the same graph over entire query execution period. Overall, the hash join beats the index join handily because the scan on W is a faster access method than the index on W[6].

As we can see, the approach using SteMs tracks the best of these two approaches. In early stages, it performs much like the index join by routing most of the R tuples to the W index, whereas in the later stages, it routes most of the R tuples to the $SteM_W$. The overall completion time of the hybrid approach is slightly more than that of the hash join,

[6]Both the hash join curve and the hybrid curve are quadratic until about 59 sec because the R and W tuples are both being scanned in. At this point the scan from R stops, so the curves becomes linear, with a reduced slope.

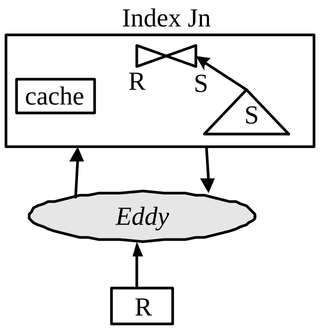

Figure 5. Executing query Q1 with a join

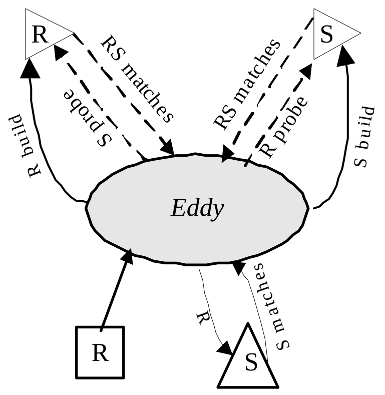

Figure 6. Executing query Q1 with SteMs

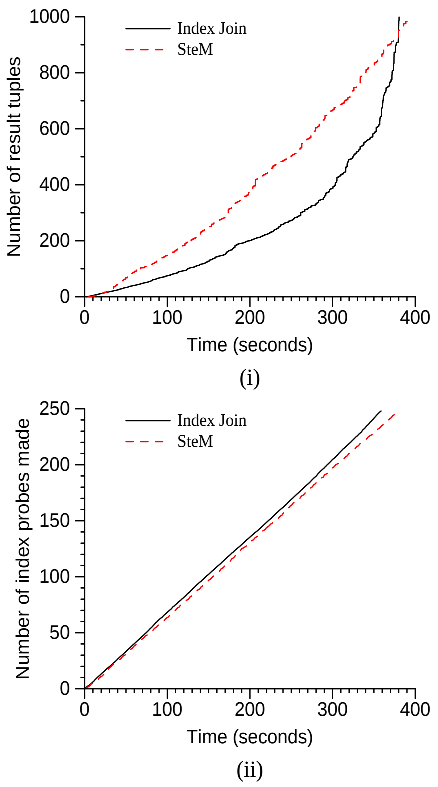

Figure 7. Number of (i) tuples output over time, (ii) probes into the S index, by the SteMs and Index Join approaches for query Q1

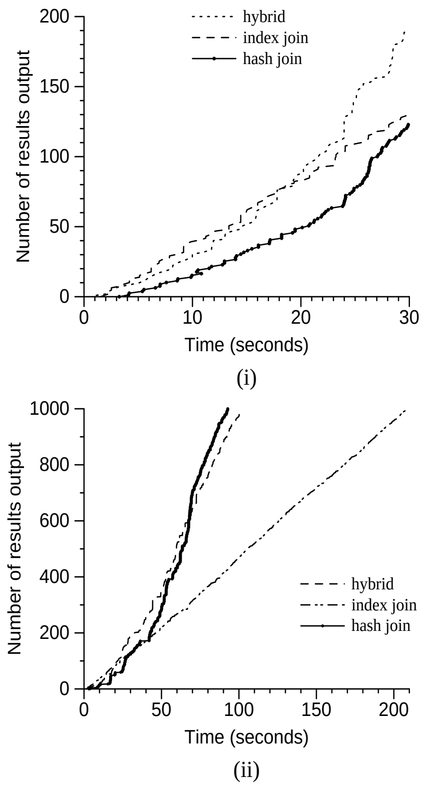

Figure 8. Number of tuples output for Q4 using index join, hash join, and the hybrid approach during: (i) first 30s, and (ii) first 200s

because the eddy keeps sending a small fraction of the R tuples to probe into the W index throughout the processing to explore alternative approaches for executing the query.

5 Related Work

SteMs were developed as a part of the Telegraph project [26], and build on the eddy tuple routing operator of [2]. Two recent sub-projects CACQ [17] and PSOUP [5] use SteMs to share state across queries for continuous query processing. The work presented here largely pre-dates CACQ and PSOUP, and is based on one of the authors' dissertation [20]. While [17] and [5] describe mainly the state-sharing aspect of SteMs, this paper explains a number of fundamental issues missing from those papers, including flexible routing constraints, work sharing, adaptive access method and join algorithm selection, adaptive spanning tree selection, handling of asynchronous data sources and duplicate elimination.

There has long been interest in adapting query optimization decisions on the fly. Due to space constraints, we only discuss the most relevant work here – for detailed surveys, please see [12] or the full paper [21]. Early work on parametric query optimization allows query plans to be chosen at run time [9]. Recently, [15] and [14] reoptimize queries at every block in the query plan, and Query Scrambling [29] reoptimizes when a source is delayed. DEC RDB ran alternative access methods simultaneously for a while before picking one [1]; aside from not being able to change the decision once it is made, this also suffers from cache fragmentation problems as we describe in [21].

At a per-operator level, the XJoin [27] dynamically changes its execution strategy to work with previously scanned tuples during source delays. There has also been some work on making hash join and sort operators adaptive to memory fluctuations [19, 32].

We depart from this prior work in two important aspects. First, we adapt execution at a fine per-tuple granularity. Second, while prior work focuses primarily on adapting join orders, our architecture allows much greater flexibility in adaptation, including choice of access paths to data sources, join algorithms, join spanning trees, and join orders.

6 Conclusions and Future Work

Join operators are an important part of traditional query processors. They typically encapsulate complex algorithms that maintain much state about the tables involved in the join. The routing of a tuple to a join often results in a chain of physical operations within the join operator.

This paper shows a way of executing queries by routing tuples not through join operators but instead through State Modules that encapsulate data structures for holding intermediate query processing state. With this mechanism, most of the decisions involved in query optimization, including the ordering of joins and selections, the choice of access methods on the tables, the choice of join algorithms, and the choice of join spanning tree are determined by the routing of tuples, and are thus made dynamically by the eddy. We have designed a set of restrictions on the eddy's routing policy that ensure correct query execution. Our experiments demonstrate that the SteMs mechanism allows powerful adaptation by the eddy in various situations.

We plan to extend this work in several directions. An important restriction of this paper is that it does not consider SteMs that span multiple tables. Though this reduces memory overheads, it can be inefficient in more traditional query execution scenarios as it leads to repeated probes that can be avoided by storing intermediate results. We are currently investigating extensions of our architecture that allow storing intermediate results, while retaining the adaptivity that SteMs provide.

Since SteMs encapsulate the data structures, and communicate directly with the eddy, they enable the eddy to observe and control memory resource utilization across *all* modules in the query. The eddy can make memory allocation decisions in a globally optimal manner, possibly based on overall memory availability as well as relative frequency of probes into each SteM. This can be extended to let the eddy control spilling of tuples to the disk as well. It will be interesting to see if such adaptive control of spilling can help the eddy simulate join algorithms such as the XJoin [27] algorithm that dynamically adapt disk spilling. In presence of multi-table SteMs, this opens up a new set of optimization opportunities, where the eddy can dynamically decide whether to materialize intermediate results or not based on memory availability.

Another important research direction is to formally study the space of join processing strategies opened up by the decoupling of state management and routing logic. We believe this will lead to better adaptive routing policies for learning many kinds of hybrid join strategies, which may be appropriate in particular circumstances but are not common enough to justify programming new join operators.

References

[1] G. Antoshnekov and M. Ziauddin. Query processing and optimization in Oracle Rdb. *VLDB Journal*, 5(4), 1996.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[3] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD*, 2002.

[4] D. Carney et al. Monitoring streams: A new class of data management applications. In *VLDB*, 2002.

[5] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

[6] D. J. DeWitt et al. Implementation techniques for main memory database systems. In *SIGMOD*, 1984.

[7] S. Fushimi et al. An overview of system software of a parallel relational database machine. In *VLDB*, 1986.

[8] R. Goldman et al. WSQ/DSQ: a practical approach for combined querying of databases and the web. In *SIGMOD*, 2000.

[9] G. Graefe and R. Cole. Optimization of dynamic query evaluation plans. In *SIGMOD*, 1994.

[10] L. M. Haas et al. Optimizing queries across diverse data sources. In *VLDB*, 1997.

[11] J. M. Hellerstein et al. Interactive data analysis: The Control project. *IEEE Computer*, 32(8), 1999.

[12] J. M. Hellerstein et al. Adaptive query processing: technology in evolution. *IEEE Data Engg. Bull.*, 23(2), 2000.

[13] T. Ibaraki and T. Kameda. Optimal nesting for computing N-relational joins. *TODS*, 9(3), 1984.

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

[15] N. Kabra et al. Effi cient mid-query reoptimization of suboptimal query execution plans. In *SIGMOD*, 1998.

[16] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, 1986.

[17] S. Madden et al. Continuosly adaptive continuous queries over streams. In *SIGMOD*, 2002.

[18] K. A. Morris. An algorithm for ordering subgoals in NAIL! In *PODS*, 1988.

[19] H. Pang et al. Memory-adaptive external sorting. In *VLDB*, 1993.

[20] V. Raman. *Interactive Query Processing*. PhD thesis, U.C.Berkeley, 2001.

[21] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. Technical report, UC Berkeley, 2002.

[22] V. Raman and J. Hellerstein. Partial results for online query processing. In *SIGMOD*, 2002.

[23] L. Rashid and S. Su. A parallel processing strategy for evaluating recursive queries. In *VLDB*, 1986.

[24] M. A. Shah et al. Java support for data-intensive systems. *SIGMOD Record*, 4(30), 2001.

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

[26] The Telegraph project. http://db.cs.berkeley.edu/telegraph.

[27] T. Urhan et al. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engg. Bull.*, 23(2), 2000.

[28] T. Urhan et al. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, 2001.

[29] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *SIGMOD*, 1998.

[30] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, 1991.

[31] V. Zadorozhny and L. Raschid. Query optimization to meet performance targets for wide area applications. In *ICDCS*, 2002.

[32] W. Zhang and P. Larson. Dynamic memory adjustment for external mergesort. In *VLDB*, 1997.