Sideways Information Passing for Push-Style Query Processing

Zachary G. Ives, Nicholas E. Taylor

Computer and Information Science Department, University of Pennsylvania
Philadelphia, PA, U.S.A.
{zives,netaylor}@cis.upenn.edu

Abstract— **In many modern data management settings, data is queried from a central node or nodes, but is stored at remote sources. In such a setting it is common to perform "push-style" query processing, using multithreaded pipelined hash joins and bushy query plans to compute parts of the query in parallel; to avoid idling, the CPU can switch between them as delays are encountered. This works well for simple select-project-join queries, but increasingly, Web and integration applications require more complex queries with multiple joins and even nested subqueries. As we demonstrate in this paper, push-style execution of complex queries can be improved substantially via *sideways information passing*; push-style queries provide many opportunities for information passing that have not been studied in the past literature. We present *adaptive information passing*, a general runtime decision-making technique for reusing intermediate state from one query subresult to prune and reduce computation of other subresults. We develop two alternative schemes for performing adaptive information passing, which we study in several settings under a variety of workloads.**

I. Introduction

Today the database query processing field has expanded to consider a number of domains beyond those of traditional client-server, parallel, or even distributed settings. Data integration [18], publish-subscribe [11], [23], and middleware [6] systems pose queries over data that is autonomously held at remote data sources — possibly including Web services or XML data sources. Peer-to-peer and network query engines [15], [22], [19], [27] pose queries over highly distributed data, often stored outside of the query engine. One of the major lessons learned in processing queries for these settings is to use **flexible scheduling**: this enables the CPU to process different portions of the plan when it encounters a delay in waiting for a particular data source [30]. Rather than using traditional iterator-driven ("pull") query processing with deterministic scheduling, most systems for querying distributed data instead implement "push" query operators such as the pipelined hash join [16], [28] and the eddy [1]; these operators are typically implemented using threads [5], [29] and thus have nondeterministic scheduling.

As data management systems of this vein become increasingly sophisticated, one of the challenges is supporting more complex queries with aggregation, many-way joins, and nested subqueries. Applications that require such capabilities include data integration, middleware-based nested XQueries over externally controlled data, and distributed data exchange [12] — the underpinnings of many e-commerce, mash-up, and customer relationship management applications. Unfortunately, complex queries may **restrict** flexibility in push-based execution by introducing blocking operations and/or dramatically increasing the amount of state that must be maintained during query processing. In this paper, we develop new techniques for *sideways information passing* that allow state to be pruned within a query plan, even **across blocking operators** and **among multiple correlated join expressions**.

Our inspiration is the traditional relational DBMS context, where many techniques were developed to handle complex queries [20], [8], [25], [7]. Unfortunately, such techniques assume the presence of indexing (seldom present in the settings described above), fast LAN-speed communications links (not present with Web-based applications), and "linear" query plans in which all joins occur between a base relation and either another base relation or an intermediate result (inappropriate for push-style applications, which use more flexible "bushy" plans where joins may also be between intermediate results [14]). They prescribe a **fixed order of query evaluation**, which may explicitly or implicitly encode *sideways information passing* within the query plan. Sideways information passing, which includes techniques like Bloomjoins [20], two-way semijoins [8], and magic sets [25], is a means of sending information from one subexpression not simply to its parent expression, but also to some other correlated portion of the query computation, in order to prune irrelevant results.

In this paper, we perform sideways information passing *adaptively*, using what we term *adaptive information passing* (AIP). AIP is a general and flexible technique that can often provide the benefits of prior techniques like the Bloomjoin, hash filter [7], or magic sets rewritings, though AIP is also beneficial in many other settings. AIP is applicable across a broad variety of queries, subject to the constraints that (1) multiple subexpressions within a given query plan must be mutually correlated through predicates, and (2) the query plan must compute these subexpressions in parallel in a fashion that preserves intermediate results. These assumptions are well-suited to push-style query processing. Specifically, we:

- identify the opportunities for sideways information passing in push-style query processing and define *adaptive information passing* (AIP),
- propose two algorithms for AIP in single- and multi-site execution of push-style queries, one based on heuristics,

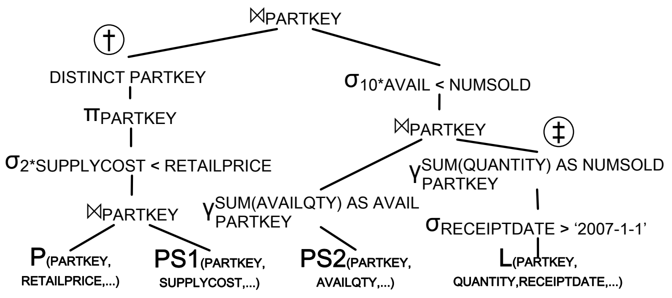

⋈PARTKEY
†
DISTINCT PARTKEY
πPARTKEY
σ2*SUPPLYCOST < RETAILPRICE
⋈PARTKEY
P(PARTKEY, RETAILPRICE,...)
PS1(PARTKEY, SUPPLYCOST,...)
σ10*AVAIL < NUMSOLD
⋈PARTKEY
‡
γSUM(AVAILQTY) AS AVAIL PARTKEY
PS2(PARTKEY, AVAILQTY,...)
γSUM(QUANTITY) AS NUMSOLD PARTKEY
σRECEIPTDATE > '2007-1-1'
L(PARTKEY, QUANTITY,RECEIPTDATE,...)

Fig. 1. Plan for example query.

and the other based on cost estimation,

- describe an implementation in the Tukwila data integration engine [16], [17], and
- experimentally demonstrate the benefits of our AIP algorithms using a variety of queries and workloads.

Section II reviews prior techniques for sideways information passing and explains why they are ill-suited to push-style query processing. Section III presents our model for adaptive information passing, which addresses these shortcomings. Section IV presents algorithms for adaptive information passing. Section V describes how these algorithms were incorporated into our Tukwila query processor. We analyze the performance of AIP in Section VI, and then conclude and discuss future work in Section VII.

II. Background and Related Work

The basic technique of *sideways information passing* — sending information from one query operator to another in a fashion not specified by the query evaluation tree — has been used in distributed [20] and deductive [2], [3] databases.

Example 2.1: We present a query that will be a running example throughout the paper, which has several opportunities for sideways information passing. Suppose we want to query the standard TPC-H benchmark schema for opportunities to break into the parts market by looking for parts that are available for much less than their retail price, but for which the stock on hand is low relative to sales so far this year.

```
SELECT DISTINCT p_partkey FROM part p, partsupp ps1,
 (SELECT ps_partkey AS partkey,
   SUM(ps_availqty) AS avail
   FROM partsupp ps2 GROUP BY ps_partkey) avail,
 (SELECT l_partkey AS partkey,
   SUM(l_quantity) AS numsold
   FROM lineitem l WHERE l_receiptdate > '2007-1-1'
   GROUP BY l_partkey) sold
WHERE p_partkey = ps_partkey
 AND p_partkey = avail.partkey
 AND p_partkey = sold.partkey
 AND 10 * avail < numsold
 AND 2 * ps_supplycost < p_retailprice
```

There are two conventional strategies for evaluating this multi-block query: (1) iterating over each value of the parent query and recomputing the subqueries, or (2) computing each block in parallel and then combining the results at the end. The former strategy repeats work in the subqueries, while the latter strategy may produce many results that are eliminated when the query blocks' outputs are combined.

We now review several previously studied means of increasing performance through sideways information passing.

Semijoins and Bloomjoins. The semijoin operator was initially studied in the context of distributed processing [4]. Given a primary source relation, it returns the subset that has a match in a second relation; this can pre-filter intermediate results before we perform an expensive operation (e.g., shipping them across a network). IBM's System-R* proposed a *two-way semijoin*: when two sites need to perform a distributed join, the first sends a projection of its join attributes to the second, which performs a semijoin and sends its matching tuples back to the original source. The original source then performs the final join. A variation is the Bloomjoin [20], where rather than sending projected attributes, the first source sends a Bloom filter summary of them. The second source may send back a few spurious tuples because of false positives in the Bloom filter — but the overall communication cost may be reduced due to the Bloom filter's small size.

Hash filters. The hash filter [7] uses Bloom filters within a bushy join query plan. The query plan is executed in stages, and Bloom filters are created and used to filter data in the next stage. Benefits are substantial if the optimizer does a good job in choosing execution stages. Unfortunately, in push query processing settings, an optimizer seldom has enough information to choose a good schedule, and hence it is likely to create a plan that computes and uses Bloom filters inefficiently.

Magic sets. Magic sets rewriting techniques define an order of evaluation across Datalog rules or SQL query blocks: values bound in the main query are propagated to restrict the computation being done in views or subquery blocks, in order to filter out tuples that fail the outer query's predicates. The information being passed from one block to another is termed a "magic set" or filter set. In essence, this set is computed in the outer query, then "shared" with the subquery, which performs a logical semijoin (on the relevant parent-child correlation predicates) between the subquery and the magic set. As initially presented in the context of deductive databases, the goal of magic sets rewritings [2], [3] was to use any constraints in the main query to more efficiently evaluate the views. In later years, many of these techniques were adapted to SQL views and nested SQL queries [21], [25].

Example 2.2: To create a magic set as part of the parent query, we determine the parts that a supplier sells for less than half of retail price. We feed this set of parts into the aggregate subqueries "in parallel;" they then run independently, computing answers restricted to possibly-relevant parts via a semijoin with the magic set. The parent then joins their results together and performs the selection to restrict the final result to parts with low availability.

Discussion. In traditional sideways information passing, the query optimizer makes an *a priori* decision about what information to pass, how to pass it, and where to pass it. It encodes this information in the form of query plan structure and choice of algorithms. Often, by choosing which information to pass, it must discard any alternatives, as the query plan can only pass information in one direction. Yet in the push model, multiple computations are going on in parallel, many subresults are being computed simultaneously, and several of these computations may produce information to pass across the query plan. The order of completion may not be known until runtime, so it is difficult for a query optimizer to take full advantage of the opportunities.

Prior adaptive techniques. An apparent solution might be techniques like eddies [1] or corrective query processing [17], which change the query plan on-the-fly in response to observed selectivities. In fact, the prior adaptive query processing techniques of which we are aware [10] can only change the **query execution plan** or the **dataflow** through it — whereas the technique we present next can prune against **many correlated expressions simultaneously** without creating additional intermediate state. Moreover, unlike these prior approaches, our techniques work across blocking operations like aggregation.

III. Adaptive Information Passing

In order to take better advantage of sideways information passing opportunities within a push-style query plan, we introduce *adaptive information passing*, which determines what intermediate state to pass across an executing query plan based on **runtime conditions**.

Typically, when a query optimizer chooses a query plan, it evaluates the query correlation predicates over the data in series of **binary** joins. Even in a pipelined query plan, joins in the upper part of the query plan may not "see" data from both inputs until late in plan execution — because they received their data from blocking operators (as in the join in the right side of Figure 1) or from joins that had slow inputs. Thus, a tuple may propagate through a series of join operators before it is found to not produce any output.

A. Basic Approach

If, while the query processor is processing a tuple t at some point in a query plan, it could look "holistically" at other subexpressions that have been fully computed at this point, it might be able to determine that t **cannot satisfy the predicates of the query** in combination with *any* tuples in the other relations (i.e., no tuple joins with t). Adaptive information passing takes advantage of the fact that in push-style query processing, intermediate results are **computed and buffered** in the hash tables of pipelined hash joins or hash-based aggregation operators. Hence once a subexpression is fully computed, there is state that can be correlated against arriving tuples from another subexpression; new tuples that do not satisfy the query conditions may be discarded early.

Specifically, we can create a *summary* (e.g., a Bloom filter, histogram, or hash set) of a completed subexpression's buffered data, and use a semijoin to probe arriving tuples against this summary. The benefits are **reduced state**, greatly reducing the memory footprint of a pipelined hash join plan, as well as **faster processing**, since intermediate results can be "pruned" early in the process and do not propagate through the plan.

Example 3.1: Refer to Figure 1 and its associated SQL query. Suppose we start executing both subtrees of the root node in parallel, and the left subtree completes first. We can create a hash set of the `PARTKEY` attribute from the state in the distinct operator or in the top-level join. Now, we can inject into the right subtree two *semijoins* with this set (based on equality on `PARTKEY`), after `PS2` is read and after `L` is read. These semijoins can discard any tuples that do not join with the left subtree — eliminating non-viable tuples and reducing the amount of state in the aggregation operators.

Example 3.2: Suppose, instead, that for the query of Figure 1, the aggregation over `L` completes first. We can create a Bloom filter of the `PARTKEY` attribute from the state in the aggregation operator; this may be very small due to the predicate over `L`. We inject a semijoin into the left subtree between this set of `PARTKEY`s and the filescans of `P` and `PS`. This allows us to prune tuples from the join. The Bloom filter may return false positives, but this only slightly reduces the number of tuples pruned. We can similarly add a semijoin to the scan of `PS2` to reduce the amount of state in the other aggregation operator.

Adaptive information passing exploits correlation predicates across different subexpressions within the same query plan, regardless of whether there are intervening blocking operators: in a sense, it bypasses the normal dataflow through the query plan in order to provide filtering. As subexpressions become fully computed, it may use them (or summary structures representing them) as as "upper bounds" on what tuples might viably produce output. We term the results of a subexpression (or the summary structure of a subexpression) an *AIP set*, since it is roughly analogous to a magic set.

B. Formal Justification

We briefly justify why adaptive information passing must always produce correct results within a select-project-join query block; the arguments are similar for queries with nesting and aggregation. Suppose a query plan Q is divided into three subexpressions that are joined as part of the query: E_A, the expression that produces the AIP set; E_P, the expression that is producing a subresult we would like to prune using the AIP set; E_R, the remaining expression.

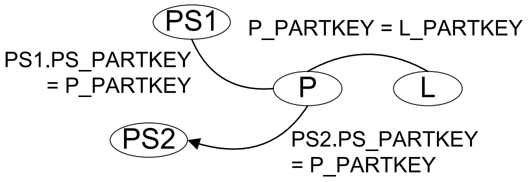
Assume the query is being executed in pipelined fashion. For E_A to produce an AIP set, all results from E_A must be fully computed, whereas E_P must only be partly computed: let E_{Pc} be the computed part of E_P and E_{Pu} be the uncomputed part. We can express the query as

$$(E_{Pc} \cup E_{Pu}) \bowtie E_R \bowtie E_A$$

For simplicity we omit the predicates on the joins. By algebraic semijoin equivalence, which holds under bag or set semantics, we may rewrite the query as

$$(E_{Pc} \cup (E_{Pu} \ltimes E_A)) \bowtie E_R \bowtie E_A$$

(a)



Source-predicate Graph

(b)

Loc.	Key	Predicates satisfied by AIP set
†	ProdKey	$2 \cdot \mathit{supplyCost} < \mathit{retailPrice}$
‡	ProdKey	$\mathit{receiptDate} >$ `'2007-1-1'`
⋮	⋮	⋮

AIP Registry

Fig. 2. AIP Manager structures for executing the query of example 1

This resembles semijoin optimizations [4]. Finally, it is easy to prove that we can rewrite the above expression to

$$(E_{Pc} \cup (E_{Pu} \triangleright_{\theta_A} E_A)) \bowtie E_R \bowtie E_A$$

where θ_A is *any conjunctive subset of clauses* from the join conditions relating E_P and E_A (i.e., it can be any less-restrictive set of predicates), and $E_{Pu} \triangleright_\theta E_A$ represents a probe of E_{Pu} against any *summary* of the tuples in E_A, which might return false positives (false matches) but never false negatives. Therefore, $E_{Pu} \triangleright_\theta E_A$ returns a superset of $E_{Pu} \ltimes E_A$.

C. Practical Considerations

The general definition of AIP, as described above, allows for arbitrary expressions in the semijoin, and therefore in the probe of the summary. In practice, neither the construction of the summaries nor the semijoin operations are free. Hence in the implementation we discuss next, we target the most beneficial, i.e., most selective, correlation conditions. We focus solely on conjunctive conditions that must hold over all query data, and in particular, equality/existence or non-equality/non-existence conditions. Such conditions can be evaluated with Bloom filters as well as hash sets. Range conditions and complex disjunctive expressions are in principle simple to implement, but in practice they are expensive to evaluate because they may require more expensive summary structures, such as histograms.

IV. AIP Algorithms

In this section, we propose two strategies. One makes greedy runtime decisions and is suitable for integration into a conventional push query processor; the second utilizes the query optimizer's cost estimator at runtime and is therefore only suitable when such features are available during query execution, such in as our Tukwila system [17].

A. Greedy Feed-Forward Filtering

Our first algorithm, which requires minimal runtime decision-making and no runtime statistics collection, optimistically creates and uses every potentially useful AIP set. During query optimization, the system creates a *source-predicate graph* describing the predicates (edges) between table variables (nodes), and whether these predicates are directional (i.e., when the correlated attribute is projected away). See Figure 2(a) for the source-predicate graph for our running example.

Query initialization. As each state-producing operator (join or group-by) is opened, it registers in a central AIP Registry a *candidate AIP set* for each attribute A it produces. Figure 2(b) shows some of the candidate AIP sets for the example query. Each operator also registers potential *interest* in other AIP sets, which it selects as follows: for each attribute A, it uses the source-predicate graph to find all attributes transitively equated to A by the query, but produced elsewhere. Once all operators have finished registering interest, any potential AIP sets without interested parties are then eliminated. Then, for each connected component in the source-predicate graph, we create in the AIP Registry a vector to hold associated and completed AIP sets. Finally, each source of an AIP set creates its own local "working copy" AIP set that it will construct incrementally.

Query execution. Upon receiving an input tuple, a join or group-by operator probes each attribute of this tuple against the appropriate vector of registered AIP sets (if any). Tuples that pass all filters are processed by the normal query operator, and are recorded in the operator's local AIP set. Space usage can be bounded by using Bloom filters; we discuss memory concerns further in Section V. Once an operator has finished reading all values from a given source, it decrements its *interest* in all the AIP sets it could have used, and then it sends its local AIP set to the AIP Registry. The registry appends the AIP set to the vector corresponding to its key attributes. If Bloom filters are used, they can be merged via bitwise intersection if they are of the same length and based on the same hash function. Finally, all other operators check if there is still interest in the AIP sets they are computing; if not, they discard their local AIP sets.

B. Cost-Based AIP

As we see in the experimental analysis, the Feed-Forward approach, in addition to being simple and easy to retrofit into an existing system, is often quite effective. However, it may incur unnecessary overhead by generating AIP sets that are useless as filters. This motivates a cost-based approach. It builds upon many of the concepts of the Feed-Forward approach, but rather than creating AIP sets incrementally, it proceeds with normal query processing until one of the input expressions to a stateful operator (i.e., join or group-by) completes. At this point, a global decision-making module, the AIP Manager, is invoked. This module evaluates the cost/benefit ratio of scanning the state within the operator, creating an AIP set, and adding the AIP set as a filter elsewhere in the query plan. This requires special support from the query engine: (1) query operators that maintain information about the cardinality of the results computed so far, (2) a cost re-estimator that can predict how expensive computation of

```
AIPCANDIDATES ($Q : qPlan, P : conjPredicateList$)
1  for all join or group-by nodes $n \in Q$ do
2      for $c \in children(n)$ do
3          for $p \in P \cap$ (preds over ATTRIBS($c$) $-$ preds in $n$) do
4              for $A \in$ ATTRIBS($p$) $\cap$ ATTRIBS($c$) do
5                  $Sources[A] := Sources[A] \cup \{n\}$
6              endfor
7          endfor
8      endfor
9  endfor
10 for every key $A$ in map $Sources$ do
11     for all join or group-by nodes $n \in Q$ do
12         if $\exists p \in P$ between EQ($A$) and EQ(ATTRIBS(n)) then
13             $Interested[A] := InterestedIn[A] \cup \{n\}$
14         endif
15     endfor
16 endfor
17 return($Sources, InterestedIn$)
```

Fig. 3. Precomputing candidate AIP set producers and users.

```
ESTIMATEBENEFIT ($Q : qPlan, A : AIPattrib, s : srcNode, R : srcResult$)
1  UPDATEESTIMATES($Q$)
2  $createCost :=$ COST(creating AIP set over R)
3  $savings := 0$
4  $used := \{\}$
5  for $n$ in $InterestedIn[A]$ in inverse order of depth in $Q$ do
6      $n' :=$ node joined with $n$ by $n$'s parent
7      $useBenefit :=$ COST($n \bowtie n'$) $-$ COST($(n \ltimes A) \bowtie n'$)
8      if $useBenefit > 0 \wedge n \notin used$ then
9          $savings := savings + useBenefit$
10         Propagate revised cardinality estimates to $n$'s ancestors
11     endif
12     if $useBenefit > 0$ then
13         Add to $used$ all ancestors of $n$ up to the common
14          ancestor of $n$ and s
15     endif
16 endfor
17 return($savings > createCost$)
```

Fig. 4. Estimating benefits of an AIP set

results will be, and (3) a means of re-invoking the cost and selectivity estimator on-the-fly for evaluating potential AIP sets.

Query initialization. During query optimization, the AIP Manager first traverses the current query plan and identifies possibly-useful AIP sets and sources. The basic algorithm, AIPCANDIDATES, takes the query plan and a list of conjunctive predicates that must hold over all contributing tuples, and it precomputes the potential sources and users of AIP sets. Pseudocode is provided in Figure 3; it references two functions that are not shown. Function EQ returns all attributes that are transitively equated by the query, and function ATTRIBS returns the set of attributes in a predicate or query plan node. AIPCANDIDATES first records the set of potential source nodes for each AIP set key (lines 1-9); note that the source nodes are the *children* of (i.e., inputs to) state-producing operators, whose results are stored within the operators. Lines 10-16 compute the sets of nodes whose output can be filtered by the AIP sets.

Query execution. Whenever an input subexpression to a pipelined hash join or hash group-by operator completes, it triggers the AIP Manager. The operator's internal state holds the relational result of the now-completed subexpression — next the AIP Manager must determine whether to construct any AIP sets from this. Algorithm ESTIMATEBENEFIT (Figure 4), given the query plan Q, a possible AIP set attribute A, source node s, and s's output result R, uses the optimizer's cost modeler to estimate the benefit of producing and using the AIP set.

ESTIMATEBENEFIT requires three functions to be provided by the query optimizer during execution. UPDATEESTIMATES updates the cardinality estimates to consider the amount of computation *remaining* in query processing. Then (lines 5-11) for each potential user n of the AIP set (evaluated in order from lowest to highest in the query plan tree), we call COST, which predicts cost of filtering n before it is joined against some other node n'. To avoid "double counting" the benefits of an AIP set, lines 12-15 ensure that, once we have seen that it is beneficial to filter node n against an AIP set, we record all of its ancestors so we do not *also* consider it beneficial to filter with them.

If an AIP set is judged to be beneficial, the AIP Manager makes the updated cardinality estimates permanent, constructs the AIP set, and injects the AIP filter into the appropriate stateful operators so n is prefiltered. In cases where there is an existing AIP filter over the same key attributes, that filter can either be intersected or, in the case of a filter with strictly weaker constraints, directly replaced.

The algorithm given above makes greedy decisions about individual potential AIP sets in isolation. This enables it to be very fast, which is key because it is invoked frequently during execution. The optimizer services invoked by the AIP Manager, namely cost estimation, do not search the plan space and therefore add little overhead.

V. IMPLEMENTATION

In this section, we discuss how we implemented our two algorithms within the Tukwila data integration engine. As mentioned previously, we consider two types of AIP sets: Bloom filters, which use limited space and are efficient to probe, but have false positives, and hash tables, which have no false positives but take more memory and are more expensive to probe. Preliminary experiments found that the added precision of a hash table was generally countered by its increased creation and probing cost. Hence, our implementation only employs Bloom filters, with a single exception which we describe in the Cost-based AIP case.

Memory overflow is not a focus in AIP because (1) AIP sets, especially if represented as Bloom filters, are small, and (2) if memory becomes an issue, an AIP set may be discarded, since it is a performance, not correctness, optimization. With a hash-based AIP set one can discard *portions*, on a per-bucket basis: any probe tuple that corresponds to a discarded bucket will simply be passed through the filter, and any probe tuple that corresponds to an existing bucket will be matched against the hash table.

A. Brief Overview of Tukwila

The Tukwila data integration engine [16], [17] is a push-style query processor whose focus is efficient processing of data integration queries, in which the remote sources may have little query processing capability of their own.

Query optimizer. Our cost-based query optimizer chooses maximally pipelined plans, emphasizing the pipelined hash join [24], [16], [28], hash-based aggregation, and bushy plans. The optimizer uses a top-down search strategy similar to Volcano's [13], and its cost modeler does not require histograms: instead, it relies on cardinality estimates and information about keys and foreign keys when estimating the selectivity of join conditions. Keys and foreign keys are useful for estimating the number of unique values of join attributes, and Tukwila's optimizer propagates this information assuming uniform distribution and uncorrelated attributes. For cases where the remote source *can* process queries, e.g., if data is located at a remote source running an instance of the Tukwila engine, the optimizer considers plans that "push" portions of the query from the "master" query node to the remote source, assuming all nodes have the same CPU costs and the network has 10Mbps bandwidth. The Tukwila optimizer and its subcomponents can be invoked at any time during execution.

Execution engine. In order to provide feedback to the query optimizer, the Tukwila query engine maintains and exposes state information. All query operators are supplemented with cardinality counters. The Tukwila query engine is heavily multithreaded, in that every pipelined hash join results in three separate threads (one for each input and one to produce join results). Finally, all stateful operators employ standardized data structures (hash table, Bloom filter, list) for preserving intermediate state, which they expose to the execution engine for use in AIP.

B. Query Processor Extensions for AIP

Feed-forward. The Feed-forward algorithm of Section IV required minimal extension to the Tukwila's query engine. The main new component, the AIP Registry, maintains a vector of completed AIP sets for each query attribute. It also determines which attributes are equated by the query. During initialization, a join reserves a container for a Bloom filter for each attribute that may be used in an AIP set, as described in Section IV-A. Finally, to enable on-the-fly pipelined query plan modification, we extended our join and group-by implementations to support registration of new semijoin operators "on the fly"; these semijoins are called when a tuple is received and before it is processed internally by the operator.

Cost-based AIP. The cost-based AIP algorithm required substantially more modifications, as it needs to be able to "look at" the entire query plan and selectively re-invoke optimizer cost and cardinality estimation. We achieved this through a global AIP Manager, which holds the data structures of Figure 2. The source-predicate graph, as in the Feed-forward algorithm, indicates equivalences of attributes of query atoms and is used to check transitive attribute equivalence. AIP sets are generated from particular portions of the query plan, and registered with the AIP Manager. The AIP Manager must also maintain cost estimate information about the query plan as different subresults are built. An AIP Registry (with more information than the version for the Feed-Forward algorithm) describes the essential characteristics of each AIP set, its key attributes and the predicates that have been evaluated in creating the set. Our cost-based algorithm only *creates* Bloom filter-based AIP sets. However, in some cases a hash table from an operator (e.g., a join) may be *directly* reused as an AIP set, if it has an appropriate key.

Distributed query extensions.

For this paper, we make only limited use of Tukwila's distributed computation capabilities: we consider how AIP sets can be sent to remote nodes to reduce data transfer rates, as in a Bloom join. We extended the cost-based AIP scheme to support distributed coordination and information passing among Tukwila query nodes. Our scheme relies on one AIP Manager with complete information about plan progress and intermediate result cardinalities. The "master" query node runs the AIP Manager and tracks the progress of the complete global query plan. Via TCP sockets, the AIP Manager periodically polls all secondary sites to discover execution progress and intermediate result availability. When a subresult becomes available at one site, the AIP Manager determines where it may be useful to inject into the global plan. We extended ESTIMATEBENEFITS's cost model with an additional factor, the cost of transmitting an AIP filter across the network. We only ship Bloom filters in our implementation, so we simply estimate the cost of shipping n bytes, where n is the size of the filter. When an AIP filter is estimated to be useful, the AIP Manager requests it from the source, relays it to the target node if necessary, and injects it into the appropriate query plan operator.

VI. Experiments

We conducted 3 classes of experiments: AIP as a means of executing subqueries when the data is arriving at a high rate; AIP with subqueries in the presence of delays, as in many wide area query processing settings; and AIP as a means of speeding up join query performance, including joins with a remote source.

Experimental workload. We elected to use the established TPC-H benchmark as a starting point, rather than designing our own data sets. For the base instance, we used the 1GB-scale TPC-H data. In some experiments we substituted a 1GB-scale TPC-D data set with the same queries, where the TPC-D data set was created by the Microsoft skewed data generator with a Zipfian skew factor z of 0.5.

We began with the basic TPC-H queries that include select, project, join, and grouping with multiple correlated predicates: these included queries 2, 5, 9, and 17 (where 2 and 17 include nested subqueries and 5 and 9 are single-block). Additionally, we added a query used previously to validate magic sets optimizations in [25]: this query somewhat resembles TPC-H query 2 but has slightly fewer joins. Finally, to compare the effects of variations in selectivity, we modified these basic queries along a number of axes — adding or removing predicates or weakening range conditions. Queries are listed in Table I.

To provide a point of comparison with adaptive information passing, we extended Tukwila to perform magic sets rewritings using the approach of [25]. We adopt [25]'s heuristics in pruning the optimizer search space: (1) the filter set is computed from the entire outer query, and (2) the filter set contains the largest number of attributes that can be joined. Our implementation performs full pipelining when computing the filter set: the filter set is computed simultaneously with the main query and the subquery. Experiments were repeated a minimum of 5 times and 95% confidence intervals are included. Our Tukwila engine is approximately 80,000 lines of C++ code. Experiments were conducted on a dual 3GHz Xeon machine with 2GB of memory running Windows Server 2003. For AIP, our Bloom filters use one hash function and are sized for a 5% false positive rate. Our cost estimates for transmitting Bloom filters assume 10Mbps data transfer rates.

A. AIP and Correlated Subqueries

Our initial motivation in developing adaptive information passing was to facilitate better multi-block query processing capabilities in push-style query engines. Thus our first experiment focuses on the performance of AIP for correlated, nested SQL queries: these are the queries that have previously been shown to be amenable to magic sets decorrelation. Figures 5 and 6 show the running times among normal query processing with no special optimizations (baseline), our pipelined magic sets implementation (Magic), and the two AIP approaches, Feed-forward and Cost-based. Figures 7 and 8 show the corresponding space usage. These queries were executed under "optimum" data transfer conditions to see how the algorithms perform when we are primarily CPU-bound. We streamed data directly from disk and without the presence of indices, since random access is unavailable in most push-based applications.

For most of our workload queries, the magic sets rewriting outperformed the baseline approach in terms of running time. Tukwila pipelines computation of the filter set and execution of the parent and child query blocks, and use of the filter set is beneficial. In one case, Q2E, the magic set is not useful as a filter, so the running time is slightly worse. Additionally, for Q2C, Magic's space usage was dramatically worse. In this case, we are seeing the effects of an optimization in Tukwila's pipelined hash join implementation: if one of the join inputs completes, the other input "short-circuits" and stops buffering input that will not be needed later. The dataflow in the Baseline plan allowed the LINEITEM table to be short-circuited early in execution, whereas the Magic plan did not.

We now discuss the AIP results, which validate that there is significant opportunity for filtering in a push-style query plan. Almost uniformly, both AIP methods outperform Baseline and Magic. Cost-based AIP is more conservative in creating AIP sets than Feed-forward, which sometimes avoids excess filtering and leads to better performance, as in Q3E; though it

TPCH-2: Q1A (normal), Q1B (skewed), Q1C (remote) select s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment from part, supplier, partsupp, nation, region where p_partkey = ps_partkey $\wedge$ s_suppkey = ps_suppkey $\wedge$ p_size = 1 $\wedge$ p_type like '%TIN' $\wedge$ s_nationkey = n_nationkey $\wedge$ n_regionkey = r_regionkey $\wedge$ r_name = 'AFRICA' $\wedge$ ps_supplycost = (select min(ps_supplycost) from partsupp,supplier,nation,region where p_partkey = ps_partkey $\wedge$ s_suppkey = ps_suppkey $\wedge$ s_nationkey = n_nationkey $\wedge$ n_regionkey = r_regionkey $\wedge$ r_name = 'AFRICA')
Q1D (child weaker) Q1A with child r_name $<$ 'S' and no p_type constraint
Q1E (parent weaker) Q1A with parent p_type $<$ 'TIN' and r_name $<$ 'S'
TPCH-17: Q2A (normal), Q2B (skewed) select sum(l_extendedprice) / 7.0 from lineitem, part where p_partkey = l_partkey $\wedge$ p_brand = 'Brand#34' $\wedge$ p_container = 'MED CAN' $\wedge$ l_quantity $<$ (select 0.2 * avg(l_quantity) from lineitem where l_partkey = p_partkey)
Q2C (parent stronger) Q2A with parent l_partkey $<$ 1000
Q2D (child stronger) Q2A with child p_partkey $<$ 1000
Q2E (parent weaker) Q2A with parent omitting predicate on p_brand
IBM [26]: Q3A (normal), Q3B (skewed), Q3C (remote) select s_name, s_acctbal, s_address, s_phone, s_comment from part, supplier, partsupp where s_nation='FRANCE' $\wedge$ p_size = 15 $\wedge$ p_type='BRASS' $\wedge$ p_partkey = ps_partkey $\wedge$ s_suppkey = ps_suppkey $\wedge$ ps_supplycost = (select min(ps_supplycost) from partsupp,supplier where p_partkey = ps_partkey $\wedge$ s_suppkey = ps_suppkey $\wedge$ s_nation='FRANCE')
Q3D (child weaker) Q3A with child n_name $>=$ 'FRANCE'
Q3E (parent-weaker) Q3A omitting parent p_size predicate
TPCH-5: Q4A (normal) select n_name, sum(l_extendedprice * (1 - l_discount)) from customer, orders, lineitem, supplier, nation, region where c_custkey = o_custkey $\wedge$ l_orderkey = o_orderkey $\wedge$ l_suppkey = s_suppkey $\wedge$ c_nationkey = s_nationkey $\wedge$ s_nationkey = n_nationkey $\wedge$ n_regionkey = r_regionkey $\wedge$ r_name = 'MIDDLE EAST' $\wedge$ o_orderdate $>=$ '1995-01-01' $\wedge$ o_orderdate $<$ '1996-01-01' group by n_name
Q4B (fewer suppliers) Q4A with l_suppkey $<$ 1000
TPCH-9: Q5A (normal) select n_name, o_year, sum(amount) from (select n_name, year(o_orderdate) as o_year, l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount from part, supplier, lineitem, partsupp, orders, nation where s_suppkey = l_suppkey $\wedge$ ps_suppkey = l_suppkey $\wedge$ ps_partkey = l_partkey $\wedge$ p_partkey = l_partkey $\wedge$ o_orderkey = l_orderkey $\wedge$ s_nationkey = n_nationkey $\wedge$ p_name like '%black%') group by n_name, o_year
Q5B (fewer nations) Q5A with n_nationkey $<$ 10

TABLE I

QUERIES USED IN EXPERIMENTS

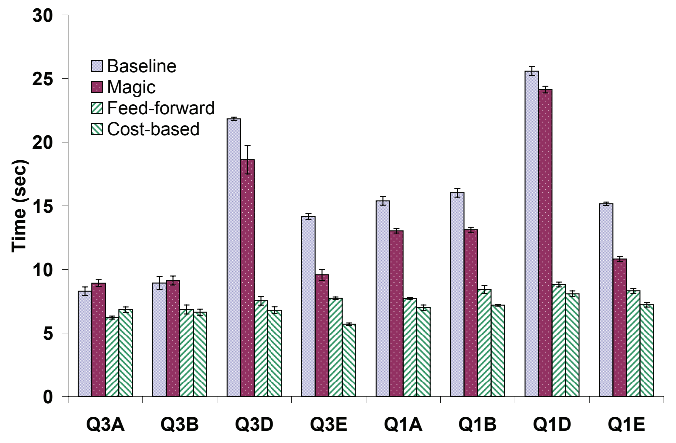


Fig. 5. Running times: Variations on TPC-H Query 2 and the IBM query.

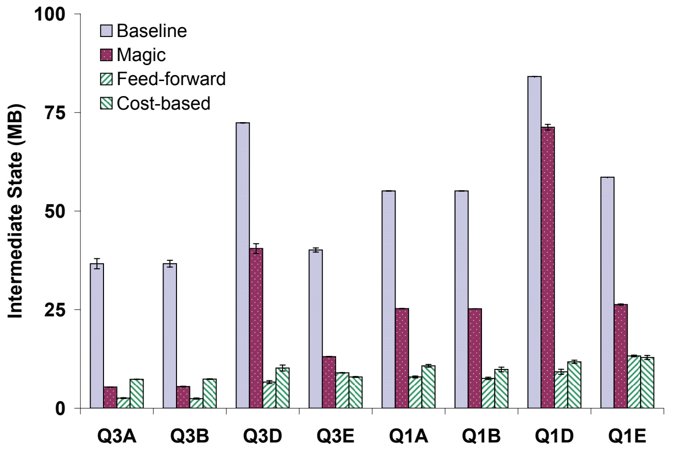


Fig. 7. Space usage: Variations on TPC-H Query 2 and IBM variant.

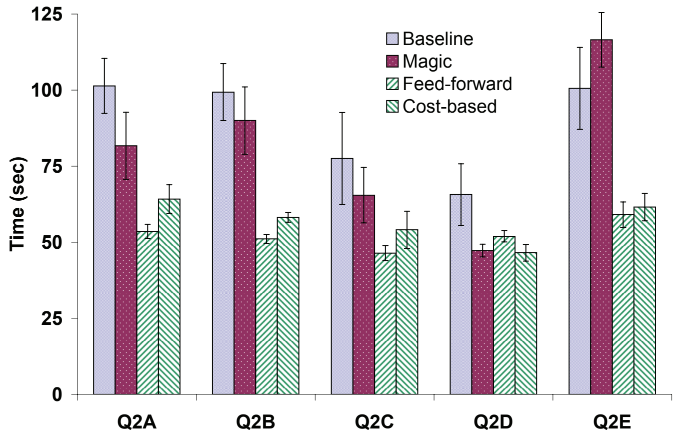


Fig. 6. Running times: Variations on TPC-H Query 17.

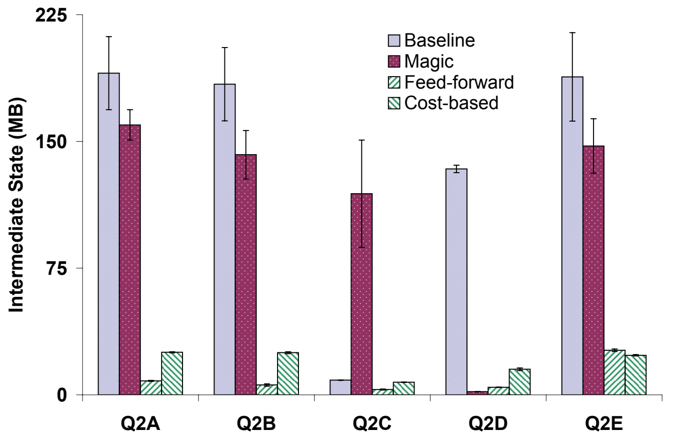


Fig. 8. Space usage: Variations on TPC-H Query 17.

may miss "borderline" filtering opportunities. Moreover, there is a delay in estimating costs and in creating AIP sets (we measured approximately 4% overhead for Q1A and 2.5% for Q2A); meanwhile unfiltered pipelined execution may continue elsewhere in the query plan. These factors mean that the cost-based approach typically performs in the same range as the naïve Feed-forward approach, which creates filters over-aggressively but does not suffer greatly because of their low overhead. Early experiments with using hash sets instead of Bloom filters showed a *significantly* greater disparity; however, Bloom filters proved to be superior in performance for all cases, so we only show Bloom filter results here.

B. AIP with Delayed Input

Since push query processing is oriented towards querying remote data, we next consider the impact of delays on overall performance. Naturally, in a pipelined query plan, as the input dataflow rates decrease, computational efficiency becomes less critical to individual query performance — I/O delays dominate. Hence one would expect that as relations are delayed, the running time disparities between different query processing methods will diminish. We repeated the previous set of queries in a setting where one of the larger input relations, PARTSUPP, was delayed by 100msec and rate-limited by injecting a 5msec delay every 1000 tuples. The space usage (Figures 11 and 12) is very similar to the previous experiment. However, as expected, the running time differences (Figures 9 and 10) have gone down. Still, there remains a noticeable performance benefit to using AIP: in fact, the Feed-forward approach becomes even more viable, as even expensive filters may provide some benefit. The Cost-based approach optimizes for CPU cost rather than query completion time, and hence it is less aggressive in generating AIP sets. It also incurs a propagation delay in receiving information about distributed AIP set availability, as well as the overheads mentioned in Section VI-A. Finally, we note that a reduction in both CPU cost and memory can be very useful in improving throughput if multiple queries are running concurrently, even if they do not decrease the latency of a single specific query.

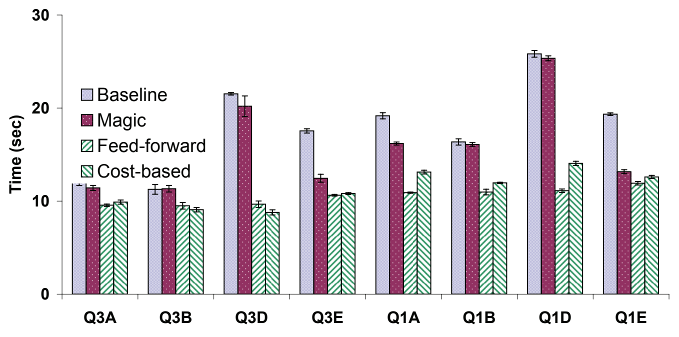


Fig. 9. Running times with delayed PARTSUPP relation: variations on TPC-H Query 2 and IBM variant.

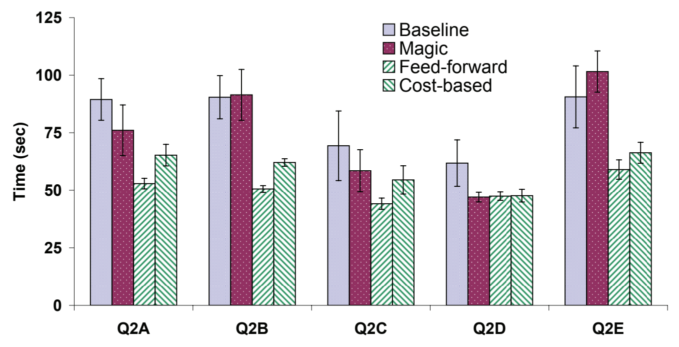


Fig. 10. Running times with delayed PARTSUPP relation: variations on TPC-H Query 17.

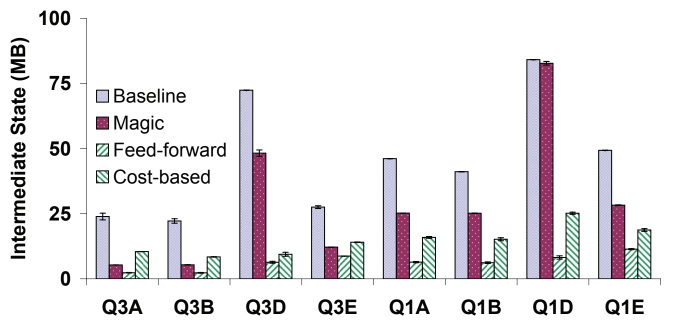


Fig. 11. Space usage under delay: variations on TPC-H Query 2 and IBM variant.

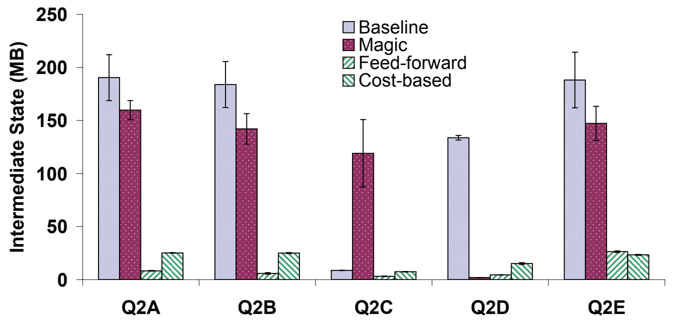


Fig. 12. Space usage under delay: variations on TPC-H Query 17.

C. AIP and Join Queries

Our final set of experiments focuses on join queries, which are seldom considered for sideways information passing. The leftmost two queries of Figures 13 and 14 show that for the base TPC-H benchmark queries, AIP reduces the amount of intermediate state and thus the running time. Not surprisingly, some of these benefits are dependent on the relative selectivities of the correlation predicates versus the predicates that are being evaluated by the joins in the query plan. For Q4B, we reduce the cardinality of the SUPPLIER relation and see the relative performance of AIP increases slightly. On the other hand, a similar change in Q5A, reducing the number of nations (Q5B), is detrimental to performance. This is because the NATION table was already being joined early in the query plan as a means of pruning state, and there are few other selection conditions in the query — meaning that few other useful filter sets exist (except against the final join, with LINEITEM, which reduces state but not running time). Here the Cost-based algorithm at least does not generate wasteful filters.

The final two queries in the figure show that in a truly distributed setting, where a remote query engine with AIP support is used to fetch data, we can derive many of the same benefits as Bloomjoins, but in an adaptive way. As discussed previously, we implemented a distributed version of the Cost-based algorithm, which can send filter sets to remote nodes "on the fly." Here, we see that the two queries, where all computation is done on the master node but the PARTSUPP relation is fetched across a 100Mb Ethernet, benefit substantially from the distributed AIP approach.

D. Summary of Experimental Results

Our experimental results show the benefit of both implementations of AIP in a variety of centralized and distributed settings. AIP offers significant savings in terms of both running time and memory usage against other execution strategies, including earlier sideways information passing techniques. The memory savings may be particularly imporant in a system that executes multiple queries simultaneously, as in such systems memory shortages can constrain performance. Furthermore, the experimental results show that use of AIP is safe: even when the query offers little or no opportunity for information passing, our techniques do not add a significant amount of overhead.

VII. Conclusions and Future Work

Adaptive information passing is a novel and general model for passing filters between portions of a query plan, which provides significant benefits in state size and running times, even when compared to magic sets, with minor overhead. We examined two implementation strategies, one of which can be easily retrofitted into an existing distributed query engine, and the other of which can be added to modern adaptive query processing systems. Experiments showed the benefits of both strategies in push-based query processing when data arrived rapidly or when it was delayed. The Feed-forward approach

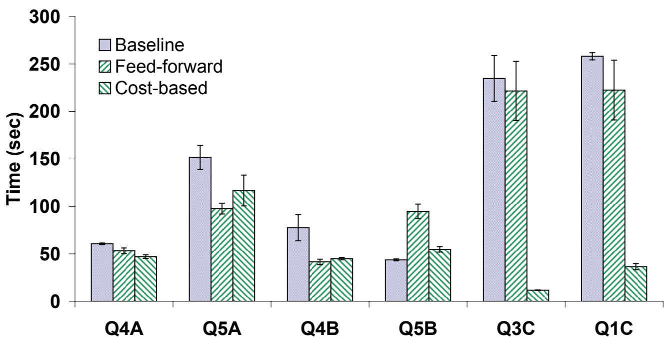


Fig. 13. Running times for join and distributed join queries.

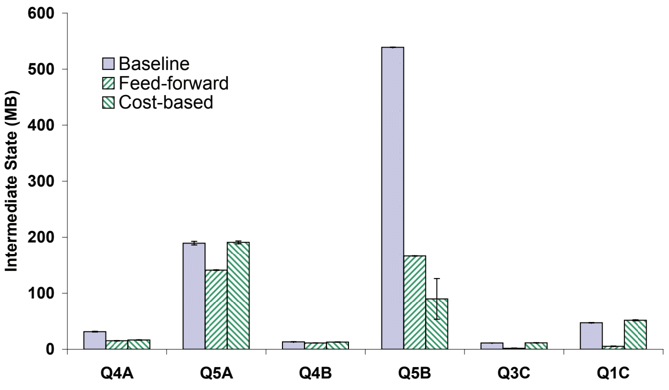


Fig. 14. Space usage for join and distributed join queries.

provides great benefits and is simple to implement; it therefore seems ideal for retrofitting into conventional push engines. The Cost-based approach provides slightly greater benefits (in particular, lower worst-case overhead) for systems that can be extended to support it, namely modern adaptive query processing engines.

We feel that adaptive information passing is a promising new technique for processing remote data. It reduces the performance and memory penalty of pipelined hash joins, which are standard in push query engines. It generalizes the concepts behind Bloomjoins, the two-phase semijoin, hash filters, and to a limited extent, magic sets. In the future, we hope to investigate the synergies between AIP and more general adaptive query processing techniques, such as STAIRS [9] or corrective query processing [17].

Acknowledgements

The authors would like to thank the anonymous reviewers of this and previous versions of this paper for many helpful comments and suggestions. Additionaly, we would like to acknowledge the assistance we received from Todd Green, J. Nathan Foster, and other members of the Penn Database Group in the preparation of this manuscript. Finally, we would like to thank Surajit Chaudhuri and Vivek Narasayya for providing us with their TPC-D skewed data set generator.

References

[1] A. Y. Levy, A. Rajaraman, and J. J. Ordille, "Querying heterogeneous information sources using source descriptions," in *VLDB*, 1996.
[2] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To, "YFilter: Efficient and scalable filtering of XML documents," in *ICDE*, 2002.
[3] O. Papaemmanouil, Y. Ahmad, U. Çetintemel, J. Jannotti, and Y. Yildirim, "Extensible optimization in overlay dissemination trees." in *SIGMOD*, 2006.
[4] M. J. Carey, "BEA Liquid Data for WebLogic: XML-based enterprise information integration," in *ICDE*, 2004.
[5] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica, "Quering the Internet with PIER," in *VLDB*, 2003.
[6] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron, "Delay aware querying with Seaweed," in *VLDB*, 2006.
[7] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan, "Declarative routing: extensible routing with declarative queries," in *SIGCOMM*, 2005.
[8] N. E. Taylor and Z. G. Ives, "Reconciling while tolerating disagreement in collaborative data sharing," in *SIGMOD*, 2006.
[9] T. Urhan, M. J. Franklin, and L. Amsaleg, "Cost based query scrambling for initial delays," in *SIGMOD*, 1998.
[10] Z. G. Ives, D. Florescu, M. T. Friedman, A. Y. Levy, and D. S. Weld, "An adaptive query execution system for data integration," in *SIGMOD*, 1999.
[11] T. Urhan and M. J. Franklin, "XJoin: A reactively-scheduled pipelined join operator," *IEEE Data Engineering Bulletin*, vol. 23, no. 2, June 2000.
[12] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *SIGMOD*, 2000.
[13] L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez, "Dynamic query scheduling in data integration systems," in *ICDE*, 2000.
[14] T. Urhan and M. J. Franklin, "Dynamic pipeline scheduling for improving interactive performance of online queries," in *VLDB*, September 2001.
[15] R. Fagin, P. Kolaitis, R. J. Miller, and L. Popa, "Data exchange: Semantics and query answering," *Theoretical Computer Science*, vol. 336, 2005.
[16] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *VLDB*, 1986.
[17] D. Daniels, "Query compilation in a distributed database system," IBM, Tech. Rep. RJ 3423, 1982.
[18] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan, "Cost-based optimization for magic: Algebra and implementation," in *SIGMOD*, 1996.
[19] M.-S. Chen, H.-I. Hsiao, and P. S. Yu, "On applying hash filters to improving the execution of multi-join queries." *VLDB J.*, vol. 6, no. 2, 1997.
[20] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang, "Optimizing queries across diverse data sources," in *VLDB*, 1997.
[21] Z. G. Ives, A. Y. Halevy, and D. S. Weld, "Adapting to source properties in processing data integration queries," in *SIGMOD*, June 2004.
[22] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic sets and other strange ways to implement logic programs," in *PODS*, 1986.
[23] C. Beeri and R. Ramakrishnan, "On the power of magic," *J. Logic Programming*, vol. 10(1/2/3&4), 1991.
[24] P. A. Bernstein and D.-M. W. Chiu, "Using semi-joins to solve relational queries," *J. ACM*, vol. 28, 1981.
[25] I. S. Mumick and H. Pirahesh, "Implementation of magic-sets in a relational database system," in *SIGMOD*, 1994.
[26] A. Deshpande, Z. Ives, and V. Raman, "Adaptive query processing," *Foundations and Trends in Databases*, 2007.
[27] L. Raschid and S. Y. W. Su, "A parallel processing strategy for evaluating recursive queries," in *VLDB*, 1986.
[28] G. Graefe and W. J. McKenna, "The Volcano optimizer generator: Extensibility and efficient search," in *ICDE*, 1993.
[29] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, "Complex query decorrelation," in *ICDE*, 1996.
[30] A. Deshpande and J. M. Hellerstein, "Lifting the burden of history from adaptive query processing," in *VLDB*, 2004.