An Adaptive Query Execution System for Data Integration*

Zachary G. Ives
University of Washington
zives@cs.washington.edu

Daniela Florescu
INRIA Roquencourt
Daniela.Florescu@inria.fr

Marc Friedman
University of Washington
friedman@cs.washington.edu

Alon Levy
University of Washington
alon@cs.washington.edu

Daniel S. Weld
University of Washington
weld@cs.washington.edu

Abstract

Query processing in data integration occurs over network-bound, autonomous data sources. This requires extensions to traditional optimization and execution techniques for three reasons: there is an absence of quality statistics about the data, data transfer rates are unpredictable and bursty, and slow or unavailable data sources can often be replaced by overlapping or mirrored sources. This paper presents the Tukwila data integration system, designed to support adaptivity at its core using a two-pronged approach. Interleaved planning and execution with partial optimization allows Tukwila to quickly recover from decisions based on inaccurate estimates. During execution, Tukwila uses adaptive query operators such as the double pipelined hash join, which produces answers quickly, and the dynamic collector, which robustly and efficiently computes unions across overlapping data sources. We demonstrate that the Tukwila architecture extends previous innovations in adaptive execution (such as query scrambling, mid-execution re-optimization, and choose nodes), and we present experimental evidence that our techniques result in behavior desirable for a data integration system.

1 Introduction

The goal of a data integration system is to provide a *uniform* query interface to a multitude of data sources. The data integration problem primarily arises in two contexts: organizations trying to provide access to a collection of internal autonomous sources, and systems that present a uniform interface to a multitude of sources available on the World-Wide Web (WWW). The key advantage of a data integration system is that it frees users from having to *locate* the sources relevant to their query, *interact* with each source independently, and manually *combine* the data from the different sources. The problem of data integration has received significant attention in the research community as evidenced by numerous research projects (*e.g.*, [10, 20, 25, 17, 9, 3, 6, 1, 25, 21, 4, 13]) and the emergence of several commercial products (*e.g.*, DataJoiner [23] and `jango.excite.com`).

Three main challenges distinguish the design of a data integration system from that of a traditional database system: query reformulation, the construction of wrapper programs, and the design of new query processing techniques for this more unpredictable environment. While the problems of reformulation and rapid wrapper development have been the focus of previous work (*e.g.*, [10, 17, 1, 9, 16, 3]), relatively little attention has been given to the development of query optimization algorithms and efficient query execution engines for data integration systems. These components are now the critical bottleneck to making such systems deployable in practice.

1.1 The Need for Adaptivity

To date, most data integration research has focused on the problem of integrating information from web-based data sources, where the amount of data returned by each source is generally small. The greater problem — that of querying over multiple autonomous data sources of moderate size, across intranets as well as the Internet — requires us to integrate novel query execution techniques.

Several characteristics of the data integration problem render existing database optimizers and execution engines (or simple extensions thereof) inappropriate in the context of data integration:

- **Absence of statistics:** statistics about the data (*e.g.*, cardinalities, histograms) are central to a query optimizer's cost estimates for query execution plans. Since data integration systems manipulate data from autonomous external sources, the system has relatively *few* and often *unreliable* statistics about the data.
- **Unpredictable data arrival characteristics:** un-

*This research was funded in part by ARPA / Rome Labs grant F30602-95-1-0024, Office of Naval Research Grant N00014-98-1-0147, by National Science Foundation Grants IRI-9303461, IIS-9872128, and 9874759.

like traditional systems, data integration systems have little knowledge about the rate of data arrival from the sources. Two phenomena that occur frequently in practice are significant initial delays before data starts arriving, and bursty arrivals of data thereafter. Hence, even if the query optimizer is able to determine the best plan based on total work, the data arrival characteristics may cause it to be inefficient in practice [22].

- **Overlap and redundancy among sources:** as a result of the heterogeneity of the data sources, there is often significant overlap or redundancy among them. Hence, the query processor needs to be able to efficiently collect related data from multiple sources, minimize the access to redundant sources, and respond flexibly when some sources are unavailable.

Since data integration systems are designed for online querying of data on the network, they have two other important characteristics. First, it is important to optimize the time to the *initial* answers to the query, rather than to minimize the total work of the system. Also, network bandwidth generally constrains the data sources to be somewhat smaller than in traditional database applications.

For all of these reasons, a data integration query processor should be *adaptive*. This is particularly true since a query optimizer is unlikely to produce good plans from bad metadata, and even a plan that may be good on average should be abandoned if unexpected situations arise. While runtime adaptivity has been shown to speed up performance even in traditional systems [15, 12], it becomes critical to performance in the data integration context (*e.g.*, [22]).

1.2 Adaptive Features of Tukwila

This paper describes the Tukwila[1] data integration system, designed with adaptivity built into its core. There are two levels at which Tukwila exhibits adaptive behavior: *between* the optimizer and execution engine, through a process of interleaved planning and execution, and *within* the execution engine, with operators designed for dynamic execution.

- **Interleaving planning and execution:** when Tukwila processes a query it does not necessarily create a *complete* query execution plan before beginning to evaluate the query. If the optimizer concludes that it does not have enough metadata with which to reliably compare candidate query execution plans, it may choose to send only a partial plan to the execution engine, and decide how to proceed after the partial plan has been completed, as in [7]. Alternatively, the optimizer may send a complete plan, but the execution engine may check for conditions that require incremental re-optimization.

[1]Tukwila is a scenic city near Seattle in the Northwest United States.

- **Adaptive operators:** Tukwila incorporates operators that are especially well suited for adaptive execution and for minimizing the time required to obtain the first answers to a query. Specifically, it employs an enhanced version of the double pipelined hash join [24] (a join implementation which executes in a symmetric, data-driven manner) and techniques for adapting its execution when there is insufficient memory. In addition, the Tukwila execution engine includes a *collector* operator whose task is to efficiently union data from a large set of possibly overlapping or redundant sources. Finally, Tukwila query execution plans can contain conditional nodes in the spirit of [12] in order to adapt to conditions that can be anticipated at optimization time.

Adaptive behavior in Tukwila is coordinated in a uniform fashion by a set of event-condition-action rules. An event may be raised by the execution of operators (e.g., out of memory, data source not responding) or at materialization points in the plan. The possible actions include modifying operator execution, reordering of operators, or re-optimization.

1.3 Example

A simple example demonstrates the breadth of Tukwila's adaptive behavior. Suppose that the same query (Figure 1a) is issued to the system under three extreme conditions: when the source tables are of unknown size, are small, or are large. Each time, assume that the relative statistics are such that a traditional optimizer would construct the join tree in Figure 1b. In a traditional query engine, the join implementations, memory allocations, and materialization points will be fixed at compile time, and the tree will be executed in a predetermined order. Tukwila implements mechanisms needed to behave more adaptively. Consider its response to the three cases:

No size information: With no information there is no point in traditional optimization. Instead, the optimizer may decide to compute a partial result that it chooses heuristically, such as the join AB, and decide afterwards what do next.

Small tables: Tukwila chooses the double pipelined join implementation for joins of small cardinality, and pipelines the entire query. When source latencies are high, this type of join has a large advantage over traditional joins, but it demands considerably more memory. To handle the "unlucky" case that memory is exceeded, the join operator has an overflow resolution mechanism.

Large tables: If the tables are sufficiently large, Tukwila's optimizer chooses standard hash joins, and breaks the pipeline, perhaps after join AB in Figure 1b. Now, depending on the rules in force, one of two things may happen during execution:

- **Rescheduling:** If all sources respond, and table AB has a cardinality sufficiently close to the optimizer's

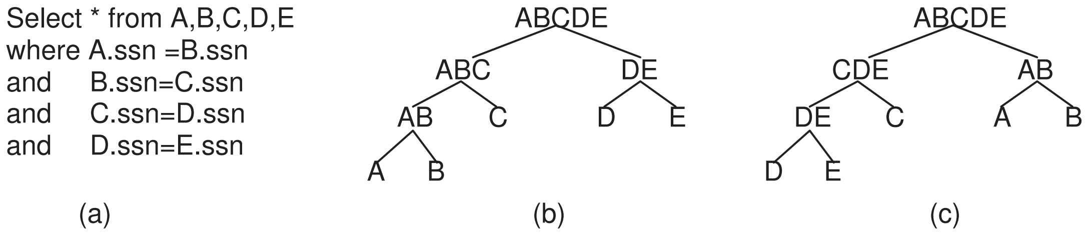


Figure 1: Sample query, initial join tree, and join tree produced by re-optimization.

estimate, execution continues normally. Should some sources respond slowly, however, Tukwila can reschedule as with query scrambling [22]. If the connection to data source A times out, join DE will be executed preemptively. Should that time out as well, the optimizer is called with that information to produce a plan reordered to use the non-blocked sources first.

- **Re-optimization:** After the AB join completes and materializes, Tukwila compares the actual cardinality with the optimizer's estimate. As in [15], if this value significantly differs from the optimizer's estimate, the optimizer is awakened to find a cheaper plan (perhaps the one in Figure 1c) given more accurate information.

The paper is organized as follows. Section 2 provides an overview of the architecture of Tukwila. Section 3 describes the mechanisms for interleaving of planning and execution. Section 4 describes the new query operator algorithms used in Tukwila. Section 5 discusses the implemented system. Section 6 describes our experimental results. Section 7 discusses related work, and Section 8 discusses several additional issues and concluding remarks.

2 Tukwila Architecture

This section provides an overview of the Tukwila architecture as illustrated in Figure 2.

Queries: A Tukwila user poses queries in terms of a mediated relational schema. The relations in the mediated schema are virtual in the sense that their extensions are not stored anywhere. The goal of the mediated schema is to abstract the details of the data sources' schemata from the user. In this paper we limit our discussion to select-project-join (conjunctive) queries over this mediated schema.

Data source catalog: The catalog contains several types of metadata about each data source. The first of these is a semantic description of the contents of the data sources. Second is overlap information about pairs of data sources (that is, the probability that a data value d appears in source S_1 if d is known to appear in source S_2) for use by collector operators, as in [8]. In the extreme case, overlap information can indicate that two sites are mirrors of each other. Finally, the catalog may contain key statistics about the data, such as the cost of accessing each source, the sizes of the relations in the sources, and selectivity information.

Query reformulation: The query over the mediated schema is fed into the Tukwila query reformulation component, which is based on an enhanced version of the algorithm described in [17]. In general, a query reformulator converts the user's query into a union of conjunctive queries referring to the data source schemata. This paper focuses on a limited form in which we have a single query that may include disjunction at the leaves. This limited disjunction, which is handled by our dynamic collector operator, is useful in handling multiple overlapping or mirrored data sources with the same attributes, *e.g.* in a query over bibliographical databases.

Query optimizer: The query optimizer transforms the rewritten query into a query execution plan for the execution engine. The optimizer has the ability to create partial plans if essential statistics are missing or uncertain, and also produces rules to define adaptive behavior during runtime.

Query execution engine: The query execution engine processes query plans produced by the optimizer. The execution engine emphasizes time-to-first result and includes operators designed to facilitate this. It includes an event handler for dynamically interpreting rules and supports incremental re-optimization.

Wrappers: the query execution engine communicates with the data sources through a set of wrapper programs. Wrappers handle the communication with the data sources and, when necessary, translate the data from the formats used in the sources to those used in Tukwila. We assume a location-independent wrapper model, where wrappers can be placed either at the data source or at the execution system.

3 Interleaving Planning and Execution

The query optimizer takes a query from the reformulator and uses information from the source catalog to produce query execution plans for the execution engine via a System-R style dynamic programming algorithm. The non-traditional aspects of the Tukwila optimizer include the following:

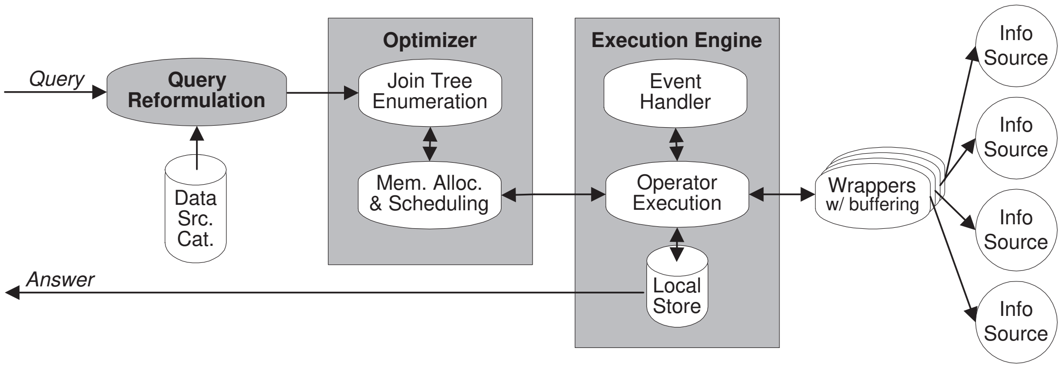


Figure 2: Architecture of the Tukwila information integration system.

- The optimizer does not always create a complete execution plan for the query. If essential statistics are missing or uncertain, the optimizer may generate a partial plan with only the first steps specified, deferring subsequent planning until sources have been contacted and critical metadata obtained.
- In addition to producing the annotated operator tree, the optimizer also generates the appropriate event-condition-action rules. These rules specify (1) when and how to modify the implementation of certain operators at runtime if needed, and (2) conditions to check at materialization points in order to detect opportunities for re-optimization.
- The query optimizer conserves the state of its search space when it calls the execution engine. The optimizer is able to efficiently resume optimization in incremental fashion if needed.

3.1 Query Plans

Operators in Tukwila are organized into pipelined units called *fragments*. At the end of a fragment, pipelines terminate, results are materialized, and the rest of the plan can be re-optimized or rescheduled. A plan consists of a partially-ordered set of fragments and a set of global rules. The partial ordering reflects constraints on the order of execution such as data flow dependencies. The global rules encode conditional execution policies, such as choosing among a set of alternative fragments after one completes. Fragments unrelated in the partial order may execute in parallel. For example, we may execute one CPU-bound fragment in parallel with other network-bound fragments as in [14].

3.1.1 Fragments and Operators

A fragment consists of a fully pipelined *tree* of physical operators, and a set of local rules. Each node in the tree is a physical operator specifying: (1) the algebraic operator at the node (*e.g.*, selection, join), (2) the chosen physical implementation of the operator (*e.g.*, hash join, double pipelined join), (3) the children of the node, (4) the memory allocated to the operator, as discussed in [5, 18], and (5) an estimate of result cardinality.

3.1.2 Rules

Rules are the key mechanism for implementing several kinds of adaptive behavior in Tukwila:

- **Re-optimization:** At the end of a fragment, if the optimizer's cardinality estimate for the fragment's result is significantly different from the actual size, the optimizer will be reinvoked (in the same spirit as [15]).
- **Contingent planning:** At the end of a fragment the execution engine can check properties of the result in order to select the next fragment (thus implementing choose nodes [12]).
- **Adaptive operators:** The policy for memory overflow resolution in the double pipelined join (Section 4.2) is guided by a rule. Collectors (Section 4.1) are also implemented using rules.
- **Rescheduling:** Rules are used for specifying when a plan should be rescheduled if a source times out (as in query scrambling [22]).

Tukwila rules have the form **when** *event* **if** *condition* **then** *actions*. For example, the following rule calls the optimizer to replan the subsequent fragments if the estimated cardinality is significantly different from the size of the result.

when closed(frag1)
if card(join1) > 2 * est_card(join1) **then** replan

Formally, a rule in a Tukwila plan is a quintuple $\langle event, condition, actions, owner, active\_flag\rangle$. An event can *trigger* a rule, causing it to check its condition. If the condition is true, the rule *fires*, executing the actions. The *owner* is the query operator or plan fragment which the rule controls or monitors. Only active rules with active owners may trigger. Firing a rule once makes it become inactive.

The execution system generates *events* in response to important changes in the execution state, such as:

- `open`, `closed`: fragment/operator starts or completes
- `error`: operator failure, *e.g.*, unable to contact source
- `timeout`(n): data source has not responded in n msec.
- `out_of_memory`: join has insufficient memory

• `threshold`(n): n tuples processed by operator

Once an event has triggered a set of associated rules, each rule's *conditions* are evaluated in parallel to determine whether any actions should be taken. Conditions are propositional formulae, with comparator terms as propositions. The quantities that can be compared include integer and state constants, states, values precomputed by the optimizer (*e.g.*, estimated cardinality or memory allocated), and various dynamic quantities in the system:

• `state`(*operator*): the operator's current state
• `card`(*operator*): the number of tuples produced so far
• `time`(*operator*): the time waiting since last tuple
• `memory`(*operator*): the memory used so far

After all rule conditions corresponding to a given event have been evaluated, *actions* are executed for those rules whose conditions are met. Most actions change some operator's memory allocation, implementation, or state. Tukwila actions include:

• set the overflow method for a double pipelined join
• alter a memory allotment
• deactivate an operator or fragment, which stops its execution and deactivates its associated rules
• reschedule the query operator tree
• re-optimize the plan
• return an error to the user

Naturally, the power of the rule language makes it possible to have conflicting or non-terminating rules. It is ultimately the responsibility of the optimizer to avoid generating such rules. However, in order to avoid the most common errors we impose a number of restrictions on rule semantics: (1) All of a rule's actions must be executed before another event is processed. (2) Rules with inactive owners are themselves inactive. (3) No two rules may ever be active such that one rule negates the effect of the other and both rules can be fired simultaneously. (This final aspect is a condition that can be statically checked.)

3.2 Query Execution

The Tukwila query execution engine is responsible not only for executing a query plan, but also for gathering statistics about each operation and for handling exception conditions or re-invoking the optimizer. The system takes a query execution plan from the optimizer and sends its rules to the event handler (Section 3.3). Then each plan fragment is processed in turn, as a single, pipelined execution unit.

The operator tree is executed using the top-down "iterator" model [11]. (Note that our implementation of the double pipelined join is an iterator-based adaptation, as described in Section 4.2). Control flows from the root node and makes its way down the tree. At the leaf nodes are file scans or requests for data from wrappers[2].

As operators within a fragment are executed, they perform two functions in addition to data manipulation: they gather cardinality statistics for the optimizer, and they invoke the event handler when significant system events (such as running out of memory, timing out on a connection, or completion of a fragment) occur.

3.3 Event Handling

The event handler is responsible for interpreting the rules that are attached to query execution plans, and thus it is the subsystem which enables most of Tukwila's adaptive behavior. The execution system may generate an event at any time. These events are fed into an event queue, which imposes an ordering on the rule evaluation process.

For each event in the queue, the event handler uses a hash table to find all matching rules that are in the active set. For each active rule, it evaluates the conditions; if they are satisfied, all of the rule's actions are executed before the next event in the queue is processed. Actions may change operator execution or cause the execution engine to terminate the current plan and re-invoke the optimizer, sending back statistics.

4 Adaptive Query Operators

Tukwila plans include the standard relational query operators: join (including dependent join), selection, projection, union and table scan. In this section, we highlight Tukwila's adaptive operators: the *dynamic collector* and the *double pipelined join* operator.

4.1 Dynamic collectors

A common task in data integration is to perform a union over a large number of overlapping sources [27, 8]. Common examples of such sources include those providing bibliographic references, movie reviews and product information. In some cases different sites are deliberately created as mirrors.

For these reasons, the Tukwila query reformulator will output queries using disjunction at the leaves. We could potentially express these disjunctions as unions over the data sources. However, a standard union operator has no mechanism for handling errors or for deciding to ignore slow mirror data sources once it has obtained the full data set, so it does not provide the flexibility needed in the data integration context. In Tukwila we treat this task as a primitive operator into which we can program a policy to guide the access to the sources.

An optimizer that has estimates of the overlap relationships between sources can provide guidance about

[2]Although several authors have considered wrappers that, in addition to accessing the data sources, may also apply relational operators to the data, in our discussion we assume that exploiting additional capabilities of the wrappers is done within the reformulator, and hence Tukwila submits atomic fetch queries to the wrappers.

the order in which data sources should be accessed, and potential fallback sources to use when a particular source is unavailable or slow (as in [8]). This guidance is given in the form of a *policy.* The query execution engine implements the policy by contacting data sources in parallel, monitoring the state of each connection, and adding or dropping connections as required by error and latency conditions. A key aspect distinguishing the collector operator from a standard union is flexibility to contact only some of the sources.

Formally, a collector operator includes a set of children (wrapper calls or table scans of cached or local data) and a policy for contacting them. A policy is a set of triples $\{\langle i, a_i, t_i \rangle\}$, associating with the ith child of the collector an activation condition a_i and a termination condition t_i. The conditions are propositional boolean formulas constructed from `true`, `false`, `and`, `or`, and four kinds of predicates on children: `closed`(c), `error`(c), `timeout`(c) and `threshold`(c). The policy is actually expressed in Tukwila as a set of event-condition-action rules, which are implemented using the normal rule-execution mechanisms.

In the example below, we have a fairly complex policy. Initially we attempt to contact sources A and B. Whichever source sends 10 tuples earliest "wins" and "kills" the other source. (Note that we take advantage of the fact that a rule owned by a deactivated node has no effect.) If Source A times out before Source B has sent 10 tuples, Source C is activated and the other sources are deactivated.

when opened(coll1)
 if true **then** activate(coll1,A); activate(coll1,B)
when threshold(A,10)
 if true **then** deactivate(coll1,B)
when threshold(B,10)
 if true **then** deactivate(coll1,A)
when timeout(A)
 if true **then** activate(coll1,C); deactivate(coll1, B);
  deactivate(coll1, A)

4.2 Double Pipelined Join

Conventional join algorithms have characteristics undesirable in a data integration system. For example, sort-merge joins (except with presorted data) and indexed joins cannot be pipelined, since they require an initial sorting or indexing step in this context. Even the pipelined join methods — nested loops join and hash join — have a flaw in that they follow an asymmetric execution model: one of the two join relations is classified as the "inner" relation, and the other as the "outer" relation. For a nested loops join, each tuple from the outer relation is probed against the entire inner relation; we must wait for the entire inner table to be transmitted initially before pipelining begins. Likewise, for the hash join, we must load the entire inner relation into a hash table before we can pipeline.

We now contrast these models with the double pipelined join (also known as the pipelined hash join), which was originally proposed in [24] for parallel database systems.

4.2.1 Conventional Hash Join

As was previously mentioned, in a standard hash join, the database system creates a hash table from the inner relation, keyed by the join attributes of the operation. Then one tuple at a time is read from the outer relation and is used to probe the hash table; all matching tuples will be joined with the current tuple and returned [11]. If the entire inner relation fits into memory, hash join requires only as many I/O operations as are required to load both relations. If the inner relation is too large, however, the data must be partitioned into smaller units that are small enough to fit into memory. Common strategies such as recursive hashing and hybrid hashing use overflow resolution, waiting until memory runs out before breaking down the relations.

In recursive hashing, if the inner relation is too large, the relation is partitioned along bucket boundaries that are written to separate files. The outer relation is then read and partitioned along the same boundaries. Now the hash join procedure is recursively performed on matching pairs of overflow files.

Hybrid hashing [11] uses a similar mechanism, but takes a "lazy" approach to creating overflow files: each time the operation runs out of memory, only a subset of the hash buckets are written to disk. After the entire inner relation is scanned, some buckets will probably remain in memory. Now, when the outer relation is read, tuples in those buckets are immediately processed; the others are swapped out to be joined with the overflow files. Naturally, hybrid hashing can be considerably more efficient than recursive hashing.

A hash join has several important parameters that can be set by an optimizer based on its knowledge of the source relations' cardinalities. Most important is the decision about which operand will be the inner relation: this should be the smaller of the two relations, as it must be loaded into a memory. Other parameters include the number of hash buckets to use, the number of buckets to write to disk at each overflow, and the amount of memory to allocate to the operator. In a conventional database system, where the optimizer has knowledge about cardinalities, and where the cost of a disk I/O from any source is the same, the join parameters can be set effectively. However, a data integration environment creates several challenges:

- The optimizer may not know the relative sizes of the two relations, and thus might position the larger relation as the inner one.
- Since the time to first tuple is important in data integration, we may actually want to use the larger data source as the inner relation if we discover that it sends data faster.

- The time to first tuple is extended by the hash join's non-pipelined behavior when it is reading the inner relation.

4.2.2 Double Pipelined Hash Join

The double pipelined hash join is a symmetric and incremental join, which produces tuples almost immediately and masks slow data source transmission rates. The trade-off is that we must hold hash tables for both relations in memory.

As originally proposed, the double pipelined join is data-driven in behavior: each of the join relations sends tuples through the join operator as quickly as possible. The operator takes a tuple, uses it to probe the hash table for the opposite join relation, and adds the tuple to the hash table for the current relation[3]. At any point in time, all of the data encountered so far has been joined, and the resulting tuples have already been output.

The double pipelined join addresses many of the aforementioned problems with a conventional hash join in a data integration system:

- Tuples are output as quickly as data sources allow, so time to first output tuple is minimized.
- The operator is symmetric, so the optimizer does not need to choose an "inner" relation.
- Its data-driven operation compensates for a slow data source by processing the other source more quickly. This also allows the query execution system to make more efficient use of the CPU, as it may process data from one join relation while waiting for the other.

On the other hand, the double pipelined join poses two problems as we attempt to integrate it into Tukwila. The first is that the double pipelined join follows a data-driven, bottom-up execution model. To integrate it with our top-down, iterator-based system, we make use of multithreading: the join consists of separate threads for output, left child, and right child. As each child reads tuples, it places them into a small *tuple transfer queue*. The join output thread then takes a tuple from either child's queue, depending on where data is present, and processes that tuple. For greater efficiency, we ensure that each thread blocks when it cannot do work (*i.e.*, when transfer queues are empty for the output thread, or full for the child threads).

The second problem with a double pipelined join is that it requires enough memory to hold both join relations, rather than the smaller of two join relations. To a large extent, we feel that this is less of a problem in a data integration environment than it is in a standard database system: the sizes of most data integration queries are expected to be only moderately large, and we may also be willing to trade off some total execution time in order to get the initial results sooner. Additionally, we expect an optimizer to use conventional joins when a relation is known to be especially large, or when one input relation is substantially smaller than the other. Nevertheless, we have identified several strategies for efficiently dealing with the problem of insufficient memory in a double pipelined join, and report on experiments with each of these methods (see Section 6).

[3] Once the opposite relation has been read in its entirety, it is no longer necessary to add tuples to the hash table unless the matching bucket has overflowed.

4.2.3 Handling Memory Overflow

When a hash join overflows, the only feasible recovery strategy is to take some portion of the hash table and swap it to disk. With the double pipelined hash join, there are at least four possibilities. First, it is possible to use statically sized buckets which are flushed and refilled every time they overflow, but this would not perform well if the relation were slightly larger than memory. Another alternative would be a conversion from double pipelined join to hybrid hash join, where we simply flush one hash table to disk.

The two algorithms we implemented in Tukwila are considerably more sophisticated and efficient. To give a feel for the algorithms' relative performance, we include an analysis here of a join between two unsorted relations A (left child) and B (right child) of equal tuple size and data transfer rate, and of the same cardinality s. For simplicity, we count tuples rather than blocks, and we further assume even distribution of tuples across hash buckets, and that memory holds m tuples. Note that our emphasis is on the disk I/O costs, and that we do not include the unavoidable costs of fetching input data across the network or writing the result.

Incremental Left Flush Upon overflow, switch to a strategy of reading only tuples from the right-side relation; as necessary, flush a bucket from the left-side relation's hash table each time the system runs out of memory. Now resume reading and joining from the left side. This approach allows the double pipelined join to gradually degrade into hybrid hash, flushing buckets lazily. If memory is exhausted before the operation completes, we proceed as follows. (1) Pause reading tuples from source A. (2) Flush some buckets from A's hash table to disk. (3) Continue reading tuples from source B, entering them into B's hash table, and using them to probe A's (partial) table; if a B-tuple belongs in a bucket whose corresponding A-bucket has been flushed, then *mark* the tuple for later processing. (4) If source B's hash table runs out of memory after A's table has been flushed completely, then write one or more of B's buckets to disk. (5) When all of B has been read, resume processing tuples from source A. If these tuples belong in a bucket which has been flushed, then write the tuples to disk; otherwise probe source B's hash table. (6) Once both sources have been processed, do a recursive hybrid hash to join the bucket overflow files. To avoid duplicates, the unmarked tuples from A should only be joined with marked tuples from B, whereas marked tuples should be joined with both unmarked and marked tuples. We calculate total costs for this algorithm as follows:

- Suppose $\frac{m}{2} < s \leq m$, so B does not overflow. We flush $s - \frac{m}{2}$ tuples from A, giving a cost of $2s - m$.
- Suppose $m < s \leq 2m$, so B is too large to fit in memory. In reading B, we overflow $(\frac{m}{2}) + (s - m)$ tuples. Reading the rest of A flushes $s + \frac{m^2}{2s} - \frac{3}{2}m$ more tuples. Our total cost becomes $4s - 4m + \frac{m^2}{s}$.

Incremental Symmetric Flush In this case, we pick a bucket to flush to disk, and flush the bucket from both sources. Steps to resolve overflow are as follows: (1) Upon memory exhaustion, choose a bucket and write that component of both A and B's hash tables to disk. (2) Continue reading tuples from both source relations. (3) If a newly read tuple belongs to a flushed bucket, mark the tuple as new and flush it to disk; otherwise, add the tuple to the appropriate hash table, and use it to probe the opposite hash table. (4) Once both sources have been processed, do a recursive hybrid hash to join the bucket overflow files. Note that the join must consider the tuple markings: unmarked tuples should only be joined with marked tuples; marked tuples should be joined with both unmarked and marked tuples. The disk I/O costs of this algorithm can be derived as follows:

Suppose $s \leq 2m$. After reading the entire contents of both tables, we have overflowed $2s - m$ tuples. After reading them back, we get a total cost of $4s - 2m$.

Our analysis suggests that incremental left-flush will perform fewer disk I/Os than the symmetric strategy, but the latter may have reduced latency since both relations continue to be processed in parallel. Section 6.3 evaluates this assessment empirically.

5 Implementation

The Tukwila system is an end-to-end platform for data integration research, from query reformulation through optimization to execution strategies and wrapper interfaces. To facilitate this, we use a component architecture with separate modules (wrappers, execution system, optimizer) communicating via well-specified APIs. Wherever possible, we leverage pre-existing standards, including TCP sockets, XML, and Unicode.

All communication between modules occurs over a socket interface. While this introduces a minimal performance penalty in cross-module calls on a single machine, it gives Tukwila several highly desirable characteristics. The first is that our system supports a limited form of scalability and distribution: all components can share a single machine or run on separate machines. A second major benefit of using sockets is that the system is language- and platform-independent. Our execution engine is written in C++ on a Windows NT/Pentium II platform; the optimizer and wrappers are written in Java, and can run on any platform supporting the language.

The query execution system accepts plans which are specified in an XML-based query plan language which is human-writable. At the end of its execution cycle (which may consist of an entire plan, or merely some subset after which the engine was directed to return to the optimizer), the execution system sends back information about operator state and cardinalities so the optimizer will have more accurate statistics.

The Tukwila query execution engine currently consists of approximately 25,000 lines of C++ code. The execution engine is designed with a multithreaded architecture in order to support prefetching and the double pipelined join and collector operators. Thread scheduling is done by the operating system, but it is controlled closely by the execution engine in order to prevent heavy contention for the CPU. We use a custom memory-management system optimized for efficient space usage in creating hash tables.

An early version of the query optimizer, implemented in Java and which includes the ability to save optimization state, was used in our experiments involving interleaving of planning and execution. For the other experiments, we used hand-coded query plans for greater control.

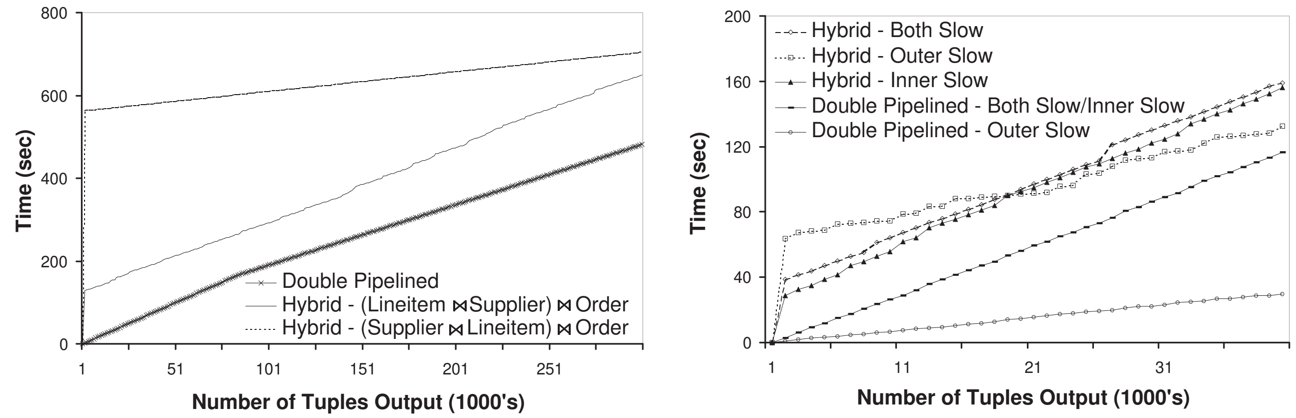
6 Experiments

We report the highlights of our experiments in four areas, showing that (1) the double pipelined join outperforms hybrid hash, (2) the preferred output behavior dictates optimal memory overflow strategy, (3) interleaved planning and execution produces significant benefits, and (4) having the optimizer save state in order to speed subsequent re-optimizations yields substantial savings.

6.1 Experimental Methodology

Experiments were performed using scaled versions of the TPC-D data set, at 50MB and 10MB, created with the **dbgen** 1.31 program. This data was stored in IBM DB2 Universal Database 5.20 on a dual-processor 450MHz Pentium II server with 512MB RAM, running Windows NT Server. The wrappers used IBM's DB2 JDBC driver, and were run directly on the server with JIT v. 3.10.93. The execution engine was run on a 450MHz Pentium II machine under NT Workstation with 256MB RAM. Our machines were connected via a standard 10Mbps Ethernet network.

For each of the experiments, we initially ran the query once to "prime" the database, then repeated it 3 times under measurement conditions. We show the average running times in our experimental results.



(a) Join Performance: Lineitem ⋈Supplier ⋈Order

(b) Wide Area Performance: Partsupp ⋈ Part

Figure 3: Double pipelined join produces initial results more quickly, is less sensitive to slow sources, and completes faster than the optimal hybrid hash join.

6.2 Performance of Double Pipelined Join

In order to compare the overall performance of the double pipelined join versus a standard join, we ran all possible joins of two and three relations in our 50MB TPC-based data set.

The results are very much in favor of the double pipelined join. In each of the experiments, we saw the same pattern: not only did the double pipelined join show a huge improvement in time to first tuple, but it also had a slightly faster time-to-completion than the hybrid hash join. This is explained by the double pipelined join's use of multithreading, which allows it to perform useful work as it is waiting for data to arrive. The exact performance gain of the double pipelined join varied depending on the sizes of the tables (since a small inner relation allows the hybrid hash join to perform well), but in all cases there was a measurable difference. Additional preliminary experiments suggest that adding prefetching to the hybrid hash join can almost remove the gap in total execution time between the two join methods, but that the double pipelined hash join still has an advantage in time-to-first-tuple.

Figure 3a shows a typical plot of tuples vs. time for the 3-relation join **lineitem** ⋈ **order** ⋈ **supplier** with different configurations of the join tree. **lineitem** is larger than the combined **order** ⋈ **supplier** result, so clearly it should be joined last. However, since the hybrid hash join is not symmetric, our assignment of inner and outer relations at each join impacts the performance for this join. In contrast, the double pipelined join performs equally well in all of these cases.

Next, we analyze the performance of the double pipelined join in a wide-area domain. In order to get realistic performance, we redirected wrapper data originating at the University of Washington to a Java "echo server" located at INRIA in France, which "bounced" the data back to the wrapper, which in turn forwarded the delayed data to the execution engine. A measurement of link bandwidth with the **ttcp** network measurement tool yielded an estimate of 82.1KB/sec, and **ping** returned a round-trip time of approximately 145msec.

Figure 3b shows the performance of a sample join, **partsupp** ⋈ **part**, under conditions where both connections are slow, the inner relation is slow, the outer relation is slow, and at full speed. As expected, we observe that the double pipelined join begins producing tuples much earlier, and that it completes the query much faster as well.

6.3 Memory Overflow Resolution

The first experiment assumed ample memory, but since double pipelined join is memory intensive, we now explore performance in a memory-limited environment. In order to contrast our double pipelined overflow resolution strategies, we ran experiments to measure the performance of these strategies under different memory conditions.

Figure 4 shows one such result. Here we are executing the join **part** ⋈ **partsupp**, which requires approximately 48MB of memory in our system. The graph shows how the number of tuples produced by a given time varies as we run the same join with full memory, 32MB of memory, and 16MB of memory.

From the figure it is apparent that the Left Flush algorithm has a much more abrupt tuple production pattern, as it runs smoothly only until the first overflow, after which it must flush and read in the right child before resuming fully pipelined operation. Note that this is still superior to the hybrid hash join, because our algorithm may still produce output as it reads the right child if there is data in the left child's hash table.

In contrast, the Symmetric Flush algorithm continues to pipeline as it overflows, but the number of buckets in memory decreases. The result is a a somewhat smoother curve which is dependent on the skew of the data.

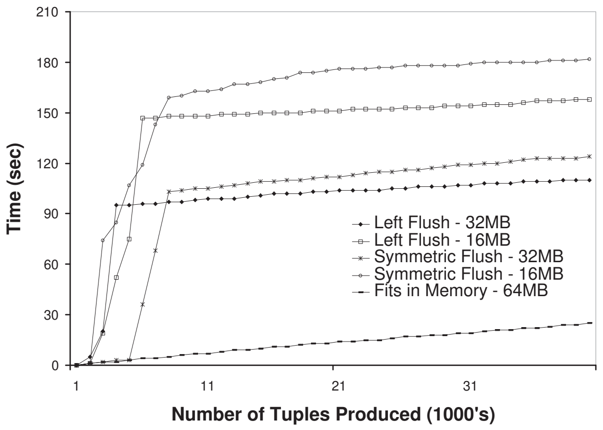


Figure 4: Symmetric Flush outputs tuples more steadily, but the rate tapers off more than with Left Flush. Overall performance of both strategies is similar.

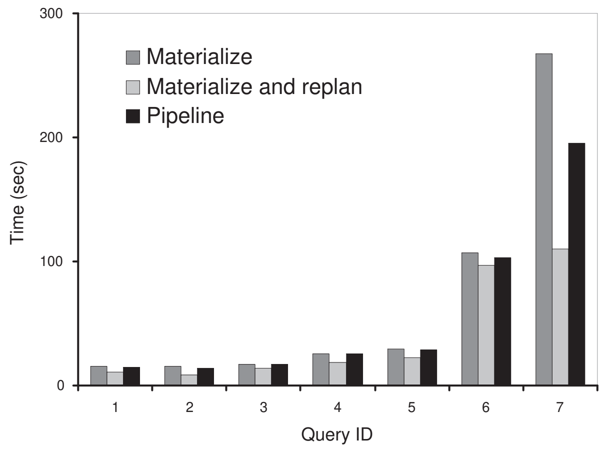


Figure 5: Even counting the cost of repeated materialization, interleaved planning and execution runs faster than a fully pipelined, static plan.

Our experiments suggest that overall running time for the two strategies is relatively close, and that the primary basis for choosing the overflow resolution strategy should be the desired pattern of tuple production. Left Flush must operate for a period in which few tuples are output, but after which it begins pipelining the left child against most or all of the right child's data. Symmetric Flush produces tuples more steadily, but its performance slows as memory is exceeded, up until the point at which the sources have been read and the overflow files can be processed.

The results also suggest that, while there is a noticeable penalty for overflowing memory with the double pipelined join, the operator's ability to produce initial tuples quickly may still make it preferable to the hybrid hash join in many situations.

6.4 Interleaved Planning and Execution

For complex queries over data sources with unknown selectivities and cardinalities, an optimizer is likely to produce a suboptimal plan. In this experiment, we demonstrate that Tukwila's strategy of interleaving planning and execution can slash the total time spent processing a query. We find that replanning can significantly reduce query completion time versus completely pipelining the plan.

For the 10MB data set, we ran all seven of the four-table joins that did not involve the **lineitem** table (which was extremely large). The optimizer was given correct source cardinalities, but it had to base its intermediate result cardinalities on estimates of join selectivities, since no histograms were available. We used the double pipelined join implementation in all cases.

In Figure 5 we see the comparison of running times for three different strategies using the same queries. The baseline strategy is simply to materialize after each join and go on to the next fragment. The second strategy added a rule to the end of each fragment, which replans whenever the cardinality of the result differs from the estimate by at least a factor of two. The third strategy is to fully pipeline the query.

In *every case*, the materialize and replan strategy was fastest, with a total speedup of 1.42 over *pipeline* and 1.69 over the naïve strategy of materializing alone. This is somewhat surprising, since the benefit of replanning based on corrected estimates overwhelms the costs of both replanning and extra materializations in each case. The most likely reason is that many of the join operations were given insufficient memory because of poor selectivity estimates, and this caused them to overflow. In practice, both cardinality and selectivity estimates of initial table sizes will be inaccurate, favoring replanning even more.

6.5 Saving Optimizer State

As the results from the previous experiment illustrate, re-optimization can yield significant performance improvements. Hence, it is common for the Tukwila execution system to re-invoke the optimizer after finishing a fragment. The optimizer then needs to correct its size estimate for the fragment's result, and update the cost estimate to reflect the cost of reading the materialization. A dynamic-programming optimizer can either replan from scratch each time, or save its state for reuse on the next re-optimization.

For the case of replanning from scratch, the query gets smaller by one operation after each join, thereby halving the size of the dynamic program. However, reuse has the advantage that any new information about the completion of a fragment can only impact half of the entries in the original table.

The advantage of saving state is that half of the useful entries in the rebuilt table have already been computed. Our stored-state algorithm visits none of these nodes. To facilitate this search strategy during re-optimization, we introduce *usage pointers* into the dynamic program from each subquery to every larger subquery that *can* use it as a left or right child. We also keep a usage pointer from every subquery to every subplan that *does* use it as a left or right child. In our final experiment, we compare replanning from scratch to re-optimization based on saved state as optimized with usage pointers. Here we realize a speedup of up to 1.64 over replanning from scratch. In separate experiments (not shown) we compare re-optimization using saved state without usage pointers and the resulting performance is worse than replanning from scratch.

7 Related work

The INGRES query optimization algorithm originally interleaved steps of constructing a query execution plan and executing it [26]. However, their approach was largely eclipsed by less flexible System-R style optimizers. Only recently have Kabra and DeWitt demonstrated the utility of runtime re-optimization for conventional database queries using a System-R style optimizer [15]. The Tukwila rule mechanism enables re-optimization as in [15] with two important advantages: (1) we do not necessarily create complete plans in advance, and (2) our optimizer was built to support efficient re-optimization while [15] used the standard Paradise optimizer.

Graefe and Ward's *choose nodes* allow the execution system to choose from a set of precompiled subplans based on runtime variables [12]. Oracle's Rdb [2] used a "dynamic optimization" strategy to deal with uncertainty by running alternative query plan subtrees in parallel competition. Although our query plans have a different structure, they can express similar choices over a set of fragments.

Tukwila provides dynamic collectors to organize access to redundant and overlapping information sources. Local completeness reasoning [9] can be used to generate policies for collectors when there are covering relationships between the sources. Probabilistic reasoning can be used whenever there is partial overlap, and completeness is not required [8]. Tukwila is the first data integration system to incorporate these techniques into a query processor.

Other researchers have investigated double pipelined joins (*e.g.*, [14, 24]), but in the context of parallel database systems as opposed to data integration. Bouganim *et al.* [5] consider adaptive scheduling techniques aimed at large queries, and Nag and DeWitt [18] investigate memory allocation strategies in the context of decision support queries.

The data integration context provides performance challenges unfamiliar in database systems. Urhan *et al.* explored replanning and rescheduling options for dealing with long source transmission delays [22] that may occur when sources are remotely located and autonomous. Our framework incorporates their adaptive algorithms.

Data integration also involves extending the query-answering problem to handle sources with varying capabilities. In contrast to our paper, much of this work has focused on either query optimization or query reformulation. The Garlic system [13] optimizes over sources that can perform joins. The work on *fusion queries* [27] optimizes queries for data that occur in multiple sources, while utilizing semijoins at the sources if possible. Systems such as TSIMMIS [10], the Information Manifold [17], Hermes [1], and Razor [9] have focused on the query reformulation component. Our research complements these projects by providing a general query execution engine.

8 Conclusions

This paper represents the first step in a larger research effort concerning query optimization and execution for data integration. Our main contribution is identifying several basic mechanisms for achieving adaptive behavior, incorporating them into a unified framework, and presenting evidence of their utility. In particular, we make the following contributions:

- We describe the architecture of the implemented Tukwila query processor. The key contribution is that adaptivity is designed into its core to facilitate interleaving of planning and execution. Furthermore, Tukwila provides a platform for incorporating hybrid optimization [19, p181] and important query optimization techniques that have been developed previously in isolation (e.g., query scrambling [22], choose nodes [12], runtime re-optimization [15], optimization of fusion queries [27]).
- We describe the design and implementation of query operators that are especially suited for adaptive behavior — the double pipelined join and the dynamic collector. We also demonstrate two useful techniques Tukwila uses to adapt the execution of a double pipelined join when there is insufficient memory for its execution.
- We use Tukwila to measure the impact of adaptive execution on data integration performance. We show that the double pipelined join outperforms the hybrid hash join. We demonstrate experimentally the efficiency gains of interleaving optimization and execution over the traditional approach of computing the entire plan before execution begins. We provide methods to efficiently resolve memory overflow for the double pipelined join. Our final experiment demonstrates the benefits of having the optimizer save state for subsequent re-optimization.

The success of our adaptive query execution system suggests a next course of action for the Tukwila project, which is to explore how the optimizer can best use our techniques in combination. We plan to discover effective strategies for generating rules and policies for dynamic collectors, as well as for combining interleaving of planning and execution and the double pipelined join to produce fast results. In addition, we plan to further extend the execution system to make use of optimistic prefetching and caching of source data.

Acknowledgments

We thank Corin Anderson, Luc Bouganim, Rachel Pottinger, and Oren Zamir for comments on the paper, and Neal Cardwell, Dennis Lee, and especially Andy Collins for assistance with wide area network simulation.

References

[1] S. Adali, K. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, 1996.

[2] G. Antoshenkov and M. Ziauddin. Query processing and optmization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.

[3] Y. Arens, C. A. Knoblock, and W.-M. Shen. Query reformulation for dynamic information integration. *International Journal on Intelligent and Cooperative Information Systems*, (6) 2/3:99–130, June 1996.

[4] J. A. Blakeley. Data access for the masses through OLE DB. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 161–172, Montreal, Canada, 1996.

[5] L. Bouganim, O. Kapitskaia, and P. Valduriez. Memory-adaptive scheduling for large query execution. In *Seventh International Conference on Information and Knowledge Management*, Bethesda, MD, Nov. 1998.

[6] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.

[7] C. Evrendilek, A. Dogac, S. Nural, and F. Ozcan. Multidatabase query optimization. *Distributed and Parallel Databases*, 5(1):77–114, 1997.

[8] D. Florescu, D. Koller, and A. Levy. Using probabilistic information in data integration. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 216–225, Athens, Greece, 1997.

[9] M. Friedman and D. Weld. Efficient execution of information gathering plans. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

[10] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[12] G. Graefe and R. Cole. Optimization of dynamic query evaluation plans. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Minneapolis, MN, 1994.

[13] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[14] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

[15] N. Kabra and D. J. DeWitt. Efficient mid-query reoptimization of sub-optimal query execution plans. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 106–117, Seattle, WA, 1998.

[16] N. Kushmerick, R. Doorenbos, and D. Weld. Wrapper induction for information extraction. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 1997.

[17] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[18] B. Nag and D. J. DeWitt. Memory allocation strategies for complex decision support queries. In *Seventh International Conference on Information and Knowledge Management*, Bethesda, MD, Nov. 1998.

[19] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 2nd edition, 1999.

[20] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[21] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to distributed heterogeneous data sources with Disco. *IEEE Transactions On Knowledge and Data Engineering*, 1998.

[22] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 130–141, Seattle, WA, 1998.

[23] S. Venkataraman and T. Zhang. Heterogeneous database query optimization in DB2 Universal DataJoiner. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 685–689, Aug. 1998.

[24] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the Int. Conf. on Parallel and Distributed Information Systems (PDIS)*, pages 68–77, Dec. 1991.

[25] D. Woelk, B. Bohrer, N. Jacobs, K. Ong, C. Tomlinson, and C. Unnikrishnan. Carnot and InfoSleuth: Database technology and the world wide web. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 443–444, San Jose, CA, 1995.

[26] E. Wong and K. Youssefi. Decomposition: A strategy for query processing. *ACM Transactions on Database Systems*, 1(3):223, 1976.

[27] R. Yerneni, Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Fusion queries over internet databases. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, pages 57–71, Valencia, Spain, 1998.