[Departmental Papers (CIS)](#)   [Department of Computer & Information Science](#)

June 2004

Adapting to Source Properties in Processing Data Integration Queries

Zachary G. Ives
University of Pennsylvania, zives@cis.upenn.edu

Alon Y. Halevy
University of Washington

Daniel S. Weld
University of Washington

Follow this and additional works at: https://repository.upenn.edu/cis_papers

Recommended Citation

Adapting to Source Properties in Processing Data Integration Queries

Abstract

An effective query optimizer finds a query plan that exploits the characteristics of the source data. In data integration, little is known in advance about sources' properties, which necessitates the use of *adaptive* query processing techniques to adjust query processing on-the-fly. Prior work in adaptive query processing has focused on compensating for delays and adjusting for mis-estimated cardinality or selectivity values. In this paper, we present a generalized architecture for adaptive query processing and introduce a new technique, called *adaptive data partitioning* (ADP), which is based on the idea of dividing the source data into regions, each executed by different, complementary plans. We show how this model can be applied in novel ways to not only correct for underestimated selectivity and cardinality values, but also to discover and exploit order in the source data, and to detect and exploit source data that can be effectively pre-aggregated. We experimentally compare a number of alternative strategies and show that our approach is effective.

Comments

Adapting to Source Properties in Processing Data Integration Queries

Zachary G. Ives
University of Pennsylvania
zives@cis.upenn.edu

Alon Y. Halevy
University of Washington
alon@cs.washington.edu

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

ABSTRACT

An effective query optimizer finds a query plan that exploits the characteristics of the source data. In data integration, little is known in advance about sources' properties, which necessitates the use of *adaptive* query processing techniques to adjust query processing on-the-fly. Prior work in adaptive query processing has focused on compensating for delays and adjusting for mis-estimated cardinality or selectivity values. In this paper, we present a generalized architecture for adaptive query processing and introduce a new technique, called *adaptive data partitioning* (ADP), which is based on the idea of dividing the source data into regions, each executed by different, complementary plans. We show how this model can be applied in novel ways to not only correct for underestimated selectivity and cardinality values, but also to discover and exploit order in the source data, and to detect and exploit source data that can be effectively pre-aggregated. We experimentally compare a number of alternative strategies and show that our approach is effective.

1. INTRODUCTION

The cornerstone of relational database query processing is the optimizer's ability to exploit known properties of the input data (cardinalities, ordering information, histograms and other selectivity estimation aids, and dependencies and uniqueness constraints) and the operating environment (CPU speed, disk access time, data striping). Some query optimizers are even sophisticated enough to exploit information about distinct values or skew in the data, e.g., to choose more appropriate operators or to perform early aggregation [4].

Unfortunately, the rich summary and statistical information available to a traditional DBMS is simply nonexistent in many *data integration* applications, where queries are posed over a variety of heterogeneous, autonomous data sources that may not even be databases. Data integration systems are frequently constructed with little knowledge of the underlying data source properties (e.g., cardinalities, ordering, functional dependencies) or the current performance characteristics of the environment (e.g., network bandwidth, data provider's resources), and in fact any of these properties is subject to change without notice. Thus, the data integration *source*

descriptions for each data source are typically quite cursory: often, they merely describe the *semantic* relationship between relations (or XML trees) in a data source and the relations (or XML trees) in the globally integrated view of the data.

Given the limited knowledge available to a data integration system, many have proposed *adaptive* techniques that discover characteristics about the data and modify query execution to exploit this knowledge. Several techniques try to compensate for unexpected variation in data access rates [27, 26, 17], or they reconsider the query execution plan at materialization points [6, 19]. Still others, focusing on online-applications, prioritize certain portions of the data so these answers are presented earlier to the user [15, 12, 25].

Adaptive query processing techniques have shown significant potential for improving performance, but they are generally best described as a collection of techniques, each designed in isolation to address a specific need (e.g., delays, estimation errors, etc.) — rather than an integrated methodology for performing query processing. An open research challenge has been to develop a more comprehensive adaptive query processing framework and system that mirrors the breadth of capabilities offered by traditional RDBMS query processors, but for data integration applications with incomplete knowledge.

In this paper, we present a novel model of adaptive query processing that emphasizes *adaptive data partitioning* and captures most existing techniques. We show that by adaptively partitioning data into multiple plans we can achieve several benefits. First, we are able to react to cost under-estimates by efficiently changing plans in mid-stream. Second, we can employ more aggressive techniques that potentially *exploit* certain properties such as order and the ability to perform pre-aggregation of data. Importantly, our architecture is designed so partitioning data into plans can be done with very little overhead, and it reuses intermediate results as much as possible. The decisions about how to partition data are based on multiple considerations, including the semantics of each operator, statistics on data we have seen, and properties of competing plans.

Overall, we make the following contributions:

- We formalize the problem of adaptive query processing, defining four different, complementary classes of techniques.

- We present a framework for *adaptive data partitioning*, which encompasses a range of techniques for changing query plans in mid-execution: metaphorically, "transformation rules" for adaptive query processing.

- We describe an implementation in the Tukwila system, including novel components for adaptive data partitioning.

- We demonstrate that adaptive data partitioning can be applied to address several opportunities for improving query performance:

– new knowledge of selectivities, allowing for us to correct poorly performing query plans in mid-execution,

– order in the data, enabling the possibility of using more efficient merge-based join algorithms,

– and potential for early aggregation, reducing the overall number of tuples to be processed.

The remainder of this paper is organized as follows. Section 2 outlines the adaptive query processing problem space and outlines the benefits of our proposed adaptive methods. Section 3 presents the novel architectural features of our adaptive query processing system, and the subsequent three sections describe methods for using adaptive data partitioning to exploit particular source data characteristics. We discuss related work in more detail in Section 7 and conclude in Section 8.

2. ADAPTIVE DATA PARTITIONING

As discussed in the introduction, data integration and remote querying applications generally pose a query processing challenge because limited information is available for query optimization. In response, researchers developed *adaptive* query processing techniques, which may modify the plan based on information obtained during execution — instead of executing it blindly. Some techniques adapt *between* successive executions of queries [5, 23, 3]; these methods rely on significant commonality between subsequent queries, and the initial queries may show poor performance. Our interests are in *intra-query* adaptivity, which allows execution to be modified *during* execution.

In this paper, we focus on what we consider to be the most flexible and powerful type of adaptive query processing, which we term *adaptive data partitioning* (ADP). We begin by describing how ADP relates to other adaptive query processing methods.

2.1 Adaptive Query Processing Taxonomy

We classify prior work in adaptive query processing into four categories:

- *Scheduling-based methods* preserve the logical structure of the query plan, but seek to re-schedule the order in which operations are processed by the CPU. Typically this is done to mask delays or to cause certain answers to appear earlier. Examples include adaptive join operators [22, 15, 26], query scrambling [27], and dynamic rescheduling [25].

- *Redundant-computation methods* use multiple competing query plans to process the same data [1]. After some time interval or completion threshold is reached, all plans are terminated except for the one that has progressed the furthest.

- *Plan partitioning methods* attempt to change the physical plan at intermediate materialization points or after a blocking operator (e.g., an aggregate). The query engine can either choose from embedded alternative subplans [6] or trigger a runtime re-optimization [19].

- *Data partitioning methods* route different parts of the data to different plans. Thus far, the sole existing technique has been eddies [2, 20], which make *local routing decisions* and send each tuple along a potentially unique path through a set of query operators. State Modules (SteMs) refine the routing granularity so that two data items may also get different access methods and join algorithms [21]. (Eddies and SteMs also capture certain aspects of scheduling-based methods, in that they execute multiple operators concurrently and can thus compensate for delays.)

The main claim of this paper is that a more general architecture for data partitioning, employing global, long-term planning as well as local decision-making, can provide much greater opportunities for adaptivity. Our novel architecture, which we term *adaptive data partitioning* (ADP), provides a powerful framework for adapting complex query plans even in the midst of pipelined execution. Adaptive data partitioning dynamically divides query processing work across multiple *different* plans, where the plans may be running in parallel or in sequence. It not only forms a basis for developing new adaptive strategies, but it actually allows other classes of adaptive techniques to be used more effectively. Scheduling-based methods are only effective if the query is not CPU-bound; ADP can reduce the cost of the currently executing plan. Plan partitioning methods modify a query plan *after* each pipeline stage; ADP provides the ability to adapt a plan *within* a stage. Finally, redundant computation may be more useful when different sets of plans can be compared over different intervals of execution.

With the Tukwila system [17, 18], we have been investigating how adaptive data partitioning can be used as the focal point of an architecture supporting all four classes of techniques. In addition to ADP, our system includes adaptive scheduling capabilities and plan partitioning, and it can even perform redundant computation. We use experimental results from Tukwila to validate that our adaptive data partitioning methods can significantly improve performance, beyond the benefits provided by other adaptive techniques.

2.2 Uses of Adaptive Data Partitioning

A major reason for our interest in ADP is the fact it can allow us to detect and exploit certain properties (e.g., selectivity, order, and groups) that are discovered to hold within the source data. To illustrate, we provide three examples in which adaptive data partitioning can allow us to react to the discovery of data source properties.

Our first example (Figure 1) shows how ADP can recover from overly optimistic cardinality estimates by switching between query plans in the middle of pipelined execution.

EXAMPLE 2.1. *Suppose we have a schema with three relations, $F(fid, from, to, when)$, representing flights; $T(ssn, flight)$, representing travelers; and $C(p, num)$, indicating the number of children per traveler. Suppose that these relations are stored in randomly distributed order. Consider a query asking for the flight with the traveler who has the most children, the number of children, and the flight's origin:*

$Group[fid, from]_{max(num)}(F \bowtie_{fid=flight} T \bowtie_{ssn=p} C)$

Given little information about the data, a query optimizer may execute the query using the order of evaluation:

$Group[fid, from]_{max(num)}(F \bowtie_{fid=flight} (T \bowtie_{ssn=p} C))$

Once the optimizer of a traditional query processor has selected a physical query plan, the system will execute it until completion. In one use of adaptive data partitioning, the system starts *in the same way, executing the plan described above ("Phase 0" in Figure 1). However, the system monitors execution, collecting statistics. From these statistics, the system may determine that it would be better to join travelers and flights before children, replacing the original join expression under the Group operator with $((F \bowtie_{fid=flight} T) \bowtie_{ssn=p} C)$.*

Rather than restart the query using the new plan, the system simply suspends *the partially executed plan (which may have already returned some answers). We call this initial plan the 0th phase of execution and denote the sets of tuples that were processed in this phase by F^0, T^0, C^0.*

Adaptive data partitioning replaces the old phase's query plan with the new (phase 1) plan, which resumes reading the source re-
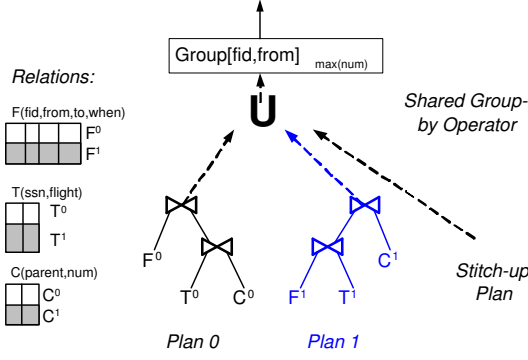
Figure 1: Executing an aggregation/join query with a combination of two plans.

lations — thus consuming all remaining tuples from the sources. However, the simple union of the 0th and 1st phase results returns only a subset of the desired answers. We must also join all combinations of relations between phases, feeding their results into the grouping operator. Omitting the join subscripts for conciseness, the remaining join expression is:

$$(F^0 \bowtie T^0 \bowtie C^1) \cup (F^0 \bowtie T^1 \bowtie C^0) \cup (F^0 \bowtie T^1 \bowtie C^1) \cup$$
$$(F^1 \bowtie T^0 \bowtie C^0) \cup (F^1 \bowtie T^0 \bowtie C^1) \cup (F^1 \bowtie T^1 \bowtie C^0)$$

Only when this final expression has been evaluated, in what we call the stitch-up *phase, will query execution complete. In general, adaptive data partitioning supports any number of phases, each correcting for a previous optimizer mis-calculation. Often, by applying ADP in such a sequence of phases, we will converge on the optimal plan. It may appear that the stitch-up phase would be expensive to compute, but this is not the case for several reasons, which we explain (and validate experimentally) in Section 4.*

We can exploit adaptive data partitioning to provide *speculation* about certain data properties, and *specialization* to these properties. Instead of running phase plans in sequence, we may run two or more plans in parallel, adaptively routing different source data to each plan.

EXAMPLE 2.2. *Suppose that we wish to join two relations that are "mostly sorted" (perhaps they were bulk loaded with some order that was not maintained by future updates). Knowing (or speculating) that this is the case, adaptive data partitioning can use one query plan specialized to joining sorted data, and a second plan to join unsorted data. As the query progresses, data from the relations will be routed to the appropriate plan depending on whether it maintains or violates order. In the end, a stitch-up plan will perform a limited amount of work to join data across the two plans.*

Early aggregation [4] is a well-known optimization that is exploited by several commercial systems. In SQL, the GROUP BY operation is typically performed last (in the absence of HAVING clauses). However, it is logically possible to perform the grouping operation before a join, since common aggregation operations such as min, max, sum, count, and average[1] distribute over union. Indeed, this transformation can sometimes reduce the amount of data to be joined.

There are two main cases when pre-aggregation is recognized to be a useful optimization. The first case holds when the join references only grouping attributes and outputs at most one tuple per

---

[1]Average is supported by pre-aggregating sums and counts for each value to be averaged, and then using these to compute the average at the end.

group. In this case, it is possible to directly commute the join and grouping operation. In the second case, when the above conditions do not hold, one may insert a *new* grouping operation ahead of the join. The new grouping operation creates smaller "partial groups" (including any join attributes, even if these are not part of the final groups), which are then fed into the join. Although the final GROUP BY must still be performed, it will now "coalesce" pre-grouped information instead of operating on original tuples.

EXAMPLE 2.3. *In the query of Example 2.1, if a traveler flies multiple times, the most efficient query plan may resemble that of Plan 0 in Figure 1, with an additional pre-aggregation operator after the $T \bowtie C$ expression (grouping on $flight$ and aggregating $max(num)$). If a traveler rarely flies, perhaps the original Plan 0 performs better.*
We can use data partitioning as a means of comparing the performance of the two plans: divide the data up into a series of subsets (say, k tuples apiece). Feed a few subsets into each of the alternative plans; compare performance on those, and then feed the remaining data into the "superior" plan.

Previously, we described four classes of adaptive techniques, including ADP. The three examples in this section are intended to give the intuition behind three ways data partitioning can be applied to query processing problems, and to illustrate its power. In the remainder of this paper, we present and validate more developed implementations of these basic applications of ADP.

While ADP provides the core of our adaptive model, we supplement it with other techniques, as appropriate. To mask I/O delays, we use adaptive scheduling — which also provides a further benefit that tuples are pipelined faster through the query plan, yielding more information about selectivities. In resource-constrained situations, we may use plan partitioning to break a query into stages and ADP to more effectively process each stage. Finally, in future work we hope to investigate how to use competitive, redundant-computation methods over ADP partitions to provide even further selectivity information.

2.3 Principles of Adaptive Data Partitioning

Now that we have provided the intuition behind adaptive data partitioning and several of its uses, we explain its algebraic underpinnings, which define a space of possible adaptive techniques, prove their correctness, and highlight their limitations. Informally, the algebraic basis shows how the answers to a query can be pieced together from multiple plans.

The key property enabling adaptive data partitioning is the ability to distribute relational union through operators such as select, project, and join. Stated generally, we can take any join expression over m relations, each divided into n partitions, and write it as:

$$R_1 \bowtie \ldots \bowtie R_m = \bigcup_{1 \le c_1 \le n, \ldots, 1 \le c_m \le n} (R_1^{c_1} \bowtie \ldots \bowtie R_m^{c_m})$$

where $R_j^{c_j}$ represents some subset of relation R_j. This is equivalent to the series of join expressions between subsets that have matching superscripts ($R_1^i \bowtie \ldots \bowtie R_m^i$, $1 \le i \le n$) plus the union of all remaining combinations:

$$\{t | t \in (R_1^{c_1} \bowtie \ldots \bowtie R_m^{c_m}), 1 \le c_i \le n, \neg(c_1 = \ldots = c_m)\}$$

Note that this two-part version of the overall expression exactly corresponds to the example of Figure 1. The two phases in the figure correspond to our join expressions with matching superscripts; the stitch-up phase performs the final union of combinations. It is straightforward to extend the above properties to include selection and projection: both operators easily distribute over union.

In fact, this property is already exploited in the hash ripple and pipelined hash joins [22, 15]. In that case, given relations R and S, of which some initial subsets R^0 and S^0, respectively, have already been joined, a ripple join can read an additional subset of relation R, R^1. It will join this with S^0, algebraically performing the operation $(R^0 \bowtie S^0) \cup (R^1 \bowtie S^0)$, but returning the same result as $(R^0 \cup R^1) \bowtie S^0$. The principles above can also be used to post-rationalize some earlier work on adaptivity, which applied them in limited settings (e.g, [2]).

An important benefit of our algebraic perspective is the ability to systematically determine additional places where these techniques may be applied. For example, we can apply adaptive data partitioning to aggregation operations in a manner that is more efficient than that of Example 2.3. We postpone the details of our *windowed* pre-aggregation scheme until Section 5, but the idea is to partition the data into a window of w tuples, and pre-aggregate elements in that window. If pre-aggregation is effective over the current window, increase the size for the next window; but if it is ineffective, reduce the window size.

3. THE TUKWILA PLATFORM FOR ADP

The Tukwila system [17, 18] integrates data from autonomous Internet and intranet data sources, where only limited information may be available to the query processor, yet the data to be queried is in the 10s or 100s of MB. Its emphasis is on (where possible) efficiently pipelining answers to the user; we have progressively added all four types of adaptivity described in Section 2.1. Our previous paper [17] discusses Tukwila's support for adaptive scheduling using pipelined hash joins [22, 26], as well as its integrated capabilities for plan partitioning and mid-query re-optimization [19]. Though Tukwila also supports pipelined processing of XML queries in a relational-like execution model [18], we focus here on issues that are not XML-specific.

In this paper, our focus is on using adaptive data partitioning to improve query processing: as alluded to in the examples of the previous section, we partition data across *parallel* subplans when the data has properties that can be exploited; we process regions or windows of the data using a *series* of plans as it becomes evident that the current plan is not performing well. Data partitioning allows us to perform adjustments that reduce the total amount of work required to answer a query. It complements adaptive scheduling, which ensures that the server is using its resources effectively, and plan partitioning, which divides work into logical units.

We now describe the novel architectural features required to implement ADP in a real query processing system. This section focuses on the execution-level mechanisms, and subsequent sections show different ways of using the framework to adapt to the properties of incoming data.

As might be expected, Tukwila has special operators for sharing information between subplans: (*split*, which partitions data across different plans; *combine*, which unions data from different plans; and a queuing operator that supports communication across concurrent threads). However, most of the novelty of the execution system lies in its support for *reusing* results from previous plan phases (avoiding the need to recompute certain query subexpressions), for monitoring the progress of execution, and for creating "stitch-up" plans. The remainder of this section describes these features: we begin by describing how we modify stateful operators to expose the intermediate results they record; then we explain how we ensure that these results are "compatible" with future plans; we describe Tukwila's execution monitoring routines; and we conclude by describing how a "stitch-up" plan is generated and executed.

3.1 Exposing State

To make multi-plan execution feasible, it is vital that we avoid re-generating work where possible — our goal is to enable sharing and reuse of subexpressions wherever effective. Hence, it must be possible to access the intermediate state (e.g., hash tables) internal to join and aggregation algorithms, in order to directly use those results in other expressions. This has required a redesign of the traditional query operators in an iterator model.

The join and aggregation operators been divided into two components, *state structures* and *iterator* modules. State structures store the relations to be joined or the intermediate aggregate results; they are similar in concept to STeMs [21], but they advertise certain *properties* (e.g., supports key-based access, requires sorted data), and they are components of pre-aggregation operators as well as joins. Tukwila includes the common data structures used in database systems: list, sorted list, hash, hash over sorted data (which allows us to perform a binary search over hash buckets), and B+ Tree.

For join algorithms, iterator modules define a creation and access pattern for state structures: nested-loops-style iteration (with buffering of the results of the inner loop); build-then-probe (as in a hybrid hash join); data-availability-driven (as in a pipelined hash join); or merge-driven (as in a merge join). There are two styles of iterators for our hash-based aggregation operators: a conventional, blocking iterator that reads the entire input relation and builds the aggregate relation in a hash table; and an "adjustable sliding window" iterator that partially pre-aggregates every w tuples, where w is variable (see Section 5). We also plan to add a "punctuated" iterator that takes input sorted by groups and outputs an aggregate tuple at the end of each group.

The goal of this decoupling of state structure and iterator is to allow Tukwila to *share* a state structure across join operators in different plans, yielding a common subexpression among different data partitions. However, for performance and consistency, we impose restrictions. For concurrently executing plans, we only allow one operator to be simultaneously adding tuples to the state structure and using the new tuples to probe another structure (e.g., in the outer loop of a nested loops join or the probe phase of a hybrid hash join)[2]. We also require that the joins' children be *compatible*, as we describe below. For executing plans in sequence (Section 4), we currently do not share results across plans, except between early and stitch-up plans, for efficiency reasons discussed in Section 3.4.1.

3.2 State Structure Compatibility

There is an important, though not immediately obvious, caveat when sharing trying to reuse intermediate results from another plan's data structures. The physical layout of the tuples for an equivalent subexpression might be different. First, the physical schema created by an expression $(A \bowtie (B \bowtie C))$ is likely to be different from that of $(B \bowtie (C \bowtie A))$, since the query processor is likely to concatenate the attributes of the joined relations in a different order. A second source of incompatibility is that a pre-aggregation operator changes the schema by projecting out non-grouping attributes and adding aggregated values.

We handle the first (tuple order-incompatibility) problem as follows. To minimize copying, tuples in Tukwila are implemented as vectors of pointers to individual attribute value "containers"; whereas state structures store a tuple's values directly, according to the tuple's physical ordering. We can read from a state structure

---

[2]This restriction prevents conflicts between different styles of iterators and it eliminates the need for message passing between the different "owners" of a structure.

into a differently-ordered tuple via a *tuple adapter* that has a mapping between the state structure's ordering and that of the tuple: it permutes the attributes within the tuple as it reads them. A similar idea allows us to efficiently map between schemas that have pre-aggregation and those that do not. Here we need not only a mapping between tuples, but a way to add placeholders for pre-aggregate values that do not exist in the non-aggregated tuple. We define a trivial *pseudogroup* operator that essentially performs pre-aggregation over each successive singleton tuple "set": for each tuple, it projects out non-grouping attributes and creates values for the aggregation attributes based on the current tuple. This converts each input tuple into a form that is schema-compatible with a pre-aggregated tuple, eliminating a source of incompatibility, but it costs little more than a conventional projection operation. Whenever Tukwila creates a plan that has a potential pre-aggregation point, it will either insert a pre-aggregation operator or a pseudogroup operator: this ensures that the same subexpression will have the same schema in any plan, regardless of whether pre-aggregation was applied.

State structure key compatibility Another constraint against structure sharing occurs when the state structure is accessed via a key, but we would like to use it in an operator that requires a different key. This situation arises occasionally in chain queries: relation C may join with relation A on attribute a, but it may join with relation B on attribute b. If there is no constraint that $a = b$, then we cannot share the state structure without converting its key.

3.3 Gathering Information for Adaptivity

At the finest level of granularity, we adjust the scheduling of operators as in prior adaptive query processing systems [26, 17, 2], where individual operators (e.g., pipelined hash joins) react to I/O and tuple availability delays to schedule work during idle cycles. Thread scheduling extends not only between operators in a single plan, but potentially across complementary query plans or subplans: if one subplan's computation is blocked by an I/O delay, perhaps another subplan can be computing a different subresult. In principle, this resembles some aspects of query scrambling [27].

More sophisticated forms of adaptivity, based on selectivity and cardinality values, are governed by a query optimizer/re-optimizer, which chooses the combination of query plans (which may ultimately run both in series and parallel) by which the query will be processed. A detailed discussion of runtime re-optimization is presented in Section 4; here we briefly describe the execution-engine mechanisms by which it gathers information.

Each of the query operators in Tukwila maintains certain information to aid the runtime decision-making modules (e.g., the query optimizer or tuple router). Every query operator maintains a counter indicating how many tuples it has output (unlike [23], we found that this had no measurable performance penalty in our workloads).

In addition to operator-level information, the query re-optimizer can make use of information exposed by the state structures of join and aggregation operators. Such information includes keys, order constraints, size and cardinality, and swapped-to-disk status. Moreover, hash tables provide an external interface by which they can be swapped to and from disk (enabling coordination of join overflow partitions, as in [10] and as discussed in Section 5).

The third adaptive component, used with complementary plans and subplans, is a router module that helps the *split* operator decide what subplan is most appropriate for an incoming tuple. The router is given a specification of each operator's constraints (e.g., order), and it may perform some additional pre-processing before routing (e.g., pre-sorting a window of the data).

3.4 Stitch-up Plans

Perhaps the most complex aspect of adaptive data partitioning lies in handling the stitch-up expression, which combines subsets of data that had previously been partitioned to different plans. In general, for a join of m relations in n plans, there are $n^m - n$ combinations of subsets that need to be stitched together.[3]

Note that one underlying requirement of adaptive data partitioning is that every plan must "buffer" the source data fed into it at the leaves, so this data can be joined with data in the other plans — this mirrors the requirement of the pipelined hash join, and in fact, since most data integration systems almost exclusively rely on pipelined hash joins, it is often trivially satisfied. In Tukwila we also extend the other join forms (nested loops, hybrid hash, and merge) to do buffering.

3.4.1 Strategies for Stitch-up

In principle, one could achieve the effects of stitch-up using a scheme similar to that of an n-ary pipelined hash join. Arrange the plans according to some arbitrary ordering (e.g., the sequence in which they are executed). For every source tuple, designate a single plan that "owns" the tuple. Feed the tuple into this owner and ensure that it buffers (e.g., in the hash table of a pipelined hash join) this tuple and all intermediate results generated by combining this tuple with others. Then feed the same tuple into all *previous* plans — these plans should join with all of their buffered data, buffering all resulting tuples — but to avoid duplicate answers, they must *avoid buffering the tuple itself*. This approach will generate correct answers, but we feel it is inadequate. First, if two or more of the plans are running in parallel and have different constraints on the sort orders of their inputs, the approach described above does not work, since subsets of the data will not simultaneously conform to both orderings. Second, if we change to a new plan because the current plan proves to be performing poorly, we would like to avoid sending future tuples into the poor plan.

A second alternative is to use an ordering scheme similar in flavor to that of the XJoin [26] (which supports multiple stages of joining input data and then joining overflowed data). As before, we assign an ordering on the plans; we also assign a *plan ID* to each plan. Next, we *share* state structures for common subexpressions across query plans. Every time an incoming tuple is fed to a particular plan, it gets annotated with the plan ID as its *lineage*, and every intermediate result tuple gets annotated with the lineages of all of its constituent source tuples. Plan ID k is responsible for all possible output values with a lineage consisting of (1) at least one source tuple from plan k and (2) the remaining tuples from plan k or earlier. This scheme has the desirable characteristic that every join combination is evaluated as early as possible, but has two important drawbacks: first, annotating tuples adds overhead to every step of query processing, and second, the optimal plan for combining the first k plans' worth of data may be different from the optimal plan for combining the data in partition k.

3.4.2 Implemented Approach

We elected to use a third approach that allows for sharing and reuse of subexpressions, while minimizing overhead and allowing

---

[3]The total number of possible tuple combinations to be evaluated remains the same as with a single join operator — we simply partition the work into smaller units. Moreover, we partition into a *new plan* that is expected to be more efficient over that data (either because the data had properties it exploits, or because the new plan is a new phase believed to improve over the old one). Ideally, the stitch-up expression can be computed in a way that does not incur additional cost beyond that of the single-plan expression.

for the most efficient means of performing stitch-up computation. We postpone the stitch-up computation to the end, after all prior plans have completed — we refer to this as a stitch-up phase.

During execution prior to stitch-up, every query plan attempts to maintain *all* of its intermediate results in state structures, since the stitch-up phase will attempt to reuse these intermediate computations wherever possible. (If memory is constrained, we use a heuristic that the state structures will be paged to disk in most-complex-expression to least-complex-expression order, based on the principle that larger expressions are less likely to be shared between plans than simpler expressions.)

Each plan "registers" its state structures in a *state structure registry* that records the plan ID, the expression, and the cardinality of the expression.

To create the stitch-up plan, the Tukwila query optimizer takes into account all existing state structures when comparing the cost of stitch-up. For every join operation, it estimates the cost of producing the unavailable intermediate results of the expression, rather than all answers[4]. Next, the optimizer creates an *exclusion list* that specifies which subexpressions have already been computed, can be reused, and should not be recomputed.

3.4.3 Stitch-up Joins

Finally, rather than taking all tuples from the existing state structures, combining them into a single structure, and annotating each with its lineage so duplicates can be discarded at the end, we provide a specialized variation of the join operator that checks tuple lineage more efficiently. The *stitch-up join* operator starts with an exclusion list provided by the optimizer (e.g., do not regenerate $A^0 \bowtie B^0$), plus sets of state structures containing existing results that are to be reused. The stitch-up join iterates over the combinations of existing state structures, and decides at a structure-to-structure level (rather than a per-tuple level) whether this combination is in the exclusion list or needs to be generated. Moreover, it decides on a pairwise basis which state structure should be scanned for tuples and which should be probed against; if necessary for performance, it will rehash one of the structures according to the join key. Finally, the stitch-up join combines data from its inputs with that from the existing state structures, checking on a per-tuple basis whether the tuple should be created. The final result is an operator that is much more efficient at producing precisely the results needed.

3.5 Implementation and Experimental Methodology

In the next three sections of this paper, we examine several different applications of ADP to data integration problems. Each requires the system to discover and exploit (or correct for) certain properties about the data: unexpected selectivity or cardinality, possibility of pre-aggregation, the presence of order. We describe how Tukwila's data partitioning framework can address each problem, and we provide experiments showing the effectiveness of our techniques.

The Tukwila query processor is approximately 80,000 lines of C++ code. Our query engine runs on a single central server but requests data from remote data sources. A Java-based front-end normally supplies queries to the system, and it also performs any final sorting (since we assume the user will want to see answers incrementally if possible). Many details of the execution engine are discussed in [17, 18, 16]. We have developed a System-R-based

query optimizer (Section 4.3) that shares the same memory space as the query execution engine; each component may trigger the other in order to provide certain adaptive behavior.

For our experiments, we use a 3.06GHz Pentium IV with the Windows XP operating system and 1GB of memory (of which 200MB is provided as a buffer pool) as the server. We disable query output to eliminate feedback effects from the client. Importantly, reflecting the common data integration paradigm, we limit access to the input relations to be sequential only, and assume that they may change between successive accesses.

Since there is no common data integration query benchmark, we use TPC-H (scale factor 0.1, 100MB) as the basis of our evaluation. While real integration workloads may be slightly different in nature, most, like TPC-H, are heavily reliant on key- and foreign-key joins. Since most data sets are not uniformly distributed, however, we supplement the TPC-H dataset with a similar one that has a skewed distribution: using a TPC-D data generator generously supplied by the Data Mining and Exploration Group at Microsoft Research, we created a skewed dataset of the same size, using a Zipf factor z of 0.5 on the major attributes.

Although in many data integration scenarios, single-query performance is limited by the speed of the network rather than computation cost, overall CPU usage is nearly as important as the running time of a single query: more efficient processing means that the data integration system can perform more concurrent queries. Hence, we include experiments using both local and remote data. All experiments are run a minimum of 4 times and 95% confidence intervals are provided.

4. CORRECTIVE QUERY PROCESSING

We begin by showing how adaptive data partitioning can be applied in a novel approach to a well-known and persistent problem: given incomplete and imprecise information (as is especially common in data integration scenarios, where even simple cardinality information may be difficult or impossible to obtain), the query optimizer is likely to choose a poor execution plan. "Poor" plans may result from inefficient CPU scheduling and/or from plan orderings that produce large intermediate results. The first problem can be addressed with pipelined hash joins (see [17, 26]); the latter problem is more challenging.

Most prior work has either attempted to improve estimates *between* query runs (e.g., [5, 23, 3]) or to partition the plan and then, at each materialization point, re-optimize the remainder of the plan [19, 17]. Although these approaches have very little overhead, they are fundamentally limited, since they cannot respond to a poorly performing query expression until *after* it has completed.

In contrast, eddies and STeMs [2, 21] perform *continuous* query re-optimization, using a data-flow heuristic that adds some per-tuple overhead but enables continuous simultaneous exploration of alternative query plans. Eddies will generally avoid "worst-case" query performance, but they spend a significant time doing "exploration" of options, and hence they devote fewer resources to "exploitation" — providing peak performance — than a traditional query engine.

In Tukwila, we aim to address the shortcomings of the plan-partitioning and inter-query adaptation approaches — namely, an inability to react to the poor performance of the *currently executing* query expression — but to dedicate the majority of computation to query answering rather than exploration of potentially-better plans. We seek to be *reactive* to bad plans, making minor course corrections, rather than *proactive* in finding better plans[5]. Tukwila occa-

---

[4]Because it only considers intermediate results that are part of our current query, and since such results are only a subset of all required data, this problem is somewhat simpler than that for optimizing using materialized views [14].

---

[5]We note that our focus is on minimizing the amount of work re-

sionally evaluates the performance of the executing query plan; if it seems to be roughly on track, execution continues. Otherwise, the system employs ADP mechanisms to halt the current query plan in midstream and to route the remaining source data to an alternate query plan that is hypothesized to be superior. Execution continues, potentially switching plans more than once; we refer to each sequential change of plans as a *phase*. Finally, the system performs a stitch-up phase at the end to compute the answers requiring data from *across* plans (as described in Section 3). We refer to this scheme as *corrective query processing*.

4.1 Basis of Decision-Making

Unlike eddies, which use a local heuristic, our approach is to try to perform a more global, cost-based evaluation of plan progress and potentially better plans. To facilitate this, our Tukwila system includes support for re-optimizing a query in a low-priority background thread, as query execution continues. If the optimizer determines that a plan appears to be proceeding well, it will simply terminate and allow execution to continue. If it finds a plan that is substantially better, it can interrupt the currently executing query plan, allow the plan to reach a *consistent* state (where all computation in the query plan, including blocking operations, has been performed on the source data that has been read), and switch to another plan using the ADP mechanisms described previously. After the input relations have been consumed, we perform the stitch-up phase of Section 3.4.

A natural question to ask is *how often* to make decisions. Several options have the potential to be effective: a triggering mechanism, in which the optimizer is invoked after some selectivity or cardinality threshold is crossed (as in [17]) would allow for reactivity at minimum cost. Alternatively, the optimizer could be run after certain milestones are passed (e.g., some percentage of data has been read). We elected to choose a more extreme approach, which is to have the optimizer poll the query plan according to some pre-specified interval. This has the benefit of allowing the optimizer to be "slightly proactive," taking advantage of obviously-better plans, rather than purely reactive. It also allows us to study how susceptible our information gathering and cost re-estimation routines are to noise and variation.

4.2 Cost Re-estimation

It is obviously trivial to determine whether the current query plan is proceeding with the expected cost, cardinality, and selectivity values. However, the challenge is in determining (1) how the remainder of query processing will proceed (especially if the data sources' cardinalities are unknown), and (2) the costs and selectivities of alternative query plans. One natural heuristic we and others (e.g, [1]) employ is to assume that query performance will be consistent throughout the lifetime of the query, and to extrapolate the overall cost and cardinality values from this. While this heuristic can be misled by variations in the data, we believe most of these situations to be unlikely to arise in practice: the majority of query cost tends to be in join operations, and the majority of joins are between keys and foreign keys, which may demonstrate skew but are fairly well bounded in the size of their output. (Later in this section, we demonstrate experimentally that our approach works well even with skewed data.)

Given these assumptions, we can fairly naturally incorporate information from the current query plan (made available by the query operators as described in Section 3) into the query optimizer's cost

---

quired to compute a query — we rely on adaptive rescheduling techniques, which are much more fine-grained, to mask delays and ensure the server is making efficient use of its resources.

estimator. However, the challenge is that the current plan only explores a very small piece of the (exponential) search space. We address this in several ways.

First, our concern is with determining *selectivities* and using those to better estimate costs. The search space is only over join orderings, not all physical query plans. We record only one *subexpression selectivity* that is shared across all logically equivalent subexpressions, regardless of algorithms used; we define this as the ratio of the subexpression's output cardinality over the product of all its input relations.

Even with this observation, the search space remains large, hence we adopt a number of heuristics that are similar in flavor to those in the System-R cost model. If a subexpression has not been computed previously, the system estimates its cardinality by averaging the following values:

- The estimate returned by standard System-R heuristics, given the estimated cardinalities of the subexpressions to be joined.

- The cardinality of each "parent expression" that joins this subexpression with a leaf-level relation. The system speculates that this parent expression may be a key-foreign key join, whose cardinality would match the size of the foreign-key relation.

While these heuristics may frequently be based on incorrect assumptions, we believe that averaging them will tend (1) to reduce the effects of a single heuristic making a poor decision, and (2) to generally provide a conservative estimate of the costs of subexpressions that are structurally similar to an expensive expression in the current plan.

Finally, we include one additional "conservative" heuristic: Tukwila also "flags" any join predicates that have been demonstrated to be "multiplicative" (i.e., joins in which the cardinality of the output exceeds the size of either input relation). Future selectivity estimates for any join expressions including these predicates will be multiplied by the same factor. This makes an independence assumption about the predicate, but it does so in a way that is likely to avoid expensive future plans.

4.3 Re-optimizer

Tukwila's re-optimizer is based on top-down enumeration (recursion with memoization, equivalent to dynamic programming but more flexible for sharing subexpressions *between* optimizer re-invocations) and mostly follows the System-R model, with a few significant variations. Our optimizer does bushy-tree enumeration, which has been shown to be important for data integration scenarios [11, 8], and it supports push-down of pre-aggregation operators, using the approach described in [4]. It supports select-project-join-aggregation queries (but not SQL subqueries). Its cost model incorporates the re-estimation features described above.

At every point, the optimizer operates in greedy fashion, re-estimating query costs based on the new estimates for source cardinalities and selectivities. Our goal is to find the plan that, based on what has already been computed, incurs the minimum cost. Hence, for every subexpression that existed in previous phases, the optimizer factors in the amount of computation that has already been performed.

4.4 Experimental Results

To examine the usefulness of our corrective query processing model, we started with the uniform TPC-H and skewed TPC-D datasets of Section 3.5, along with a query workload based on those TPC-H queries that conform to our select-project-join-aggregation

		Query 3A		Query 10		Query 10A		Query 5	
		Uniform	Skewed	Uniform	Skewed	Uniform	Skewed	Uniform	Skewed
No statistics	Phases	2	2	4	3	3	3	4	4
	Stitch-up time	5.1s	5.1s	7.5s	7.6s	36.2s	33.5s	15.4s	25.8s
	Reused tuples	754K	753K	637K	639K	794K	793K	640K	639K
	Discarded tuples	22.6K	20.8K	9.0K	0	0	0	37.4K	22.4K
Given cardinalities	Phases	1	1	1	2	1	1	3	3
	Stitch-up time	-	-	-	2.2s	-	-	15.4s	25.9s
	Reused tuples	-	-	-	631K	-	-	640K	639K
	Discarded tuples	-	-	-	7.9K	-	-	76.5K	65.8K

Table 1: Breakdown of number of stitch-up phases, time spent in stitch-up, total tuples reused from prior phases, tuples not reused by stitch-up phase.
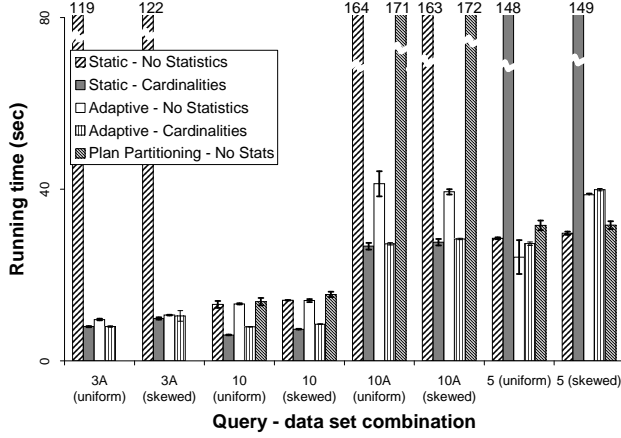

Figure 2: Static optimization, corrective query processing, and plan partitioning compared over uniformly distributed and skewed TPC data sets.

query model. These were queries 3, 10, and 5. Since query 3 was very inexpensive to compute (it took approximately 3 seconds in our system), we altered it to be more expensive by removing its date-based selection predicates (we called this query 3A). We also supplemented query 10 with a similar variation, which we called query 10A, that removed its date-based selection predicates. This left us with a workload with several levels of optimization complexity: a join of 3 relations (query 3A), two joins of 4 relations (queries 10 and 10A), and a join of 5 relations (query 5).

We configured the system to run all experiments completely within memory (using a 200MB buffer pool, as described previously): this allows us to isolate computation cost from disk I/O cost. Taking this one step further, we also configured the Tukwila engine to dynamically expand the amount of memory assigned to each hash table, should it run out of memory: this reduces the performance penalty imposed on query plans that are created with inaccurate estimates (though hash buckets in our system cannot be dynamically adjusted, meaning that an overly large relation will still suffer from many bucket collisions).

Our initial experimental comparisons were between three alternative strategies: static query optimization, our scheme for corrective query processing, and a plan-partitioning model along the lines of [19] that inserts a materialization after 3 joins and re-optimizes a query at this point. Where applicable, we consider these strategies in a case where the optimizer is not given cardinality estimates in advance — it makes a default assumption of 20,000 tuples for every relation, since that is roughly the median number of tuples in the

TPC datasets — versus a case in which the optimizer is provided with source cardinality information.

We make several observations about the results, shown in Figure 2. First, Tukwila's default query optimizer generally picks a good plan when it is provided with cardinality information, but there are times when the default selectivity estimation heuristics do not work well: for query 5, joins between CUSTOMER, NATION, and SUPPLIER produce a very large subresult when performed before the joins with the other relations. Under the "no statistics" assumption, the optimizer assumes that all relations are equally-sized, and this coincidentally works well in this situation, since it first joins the tables with selection predicates (REGION and ORDERS). In contrast, for queries 3 and 10A, which both generate a significant volume of data, the optimizer generally picks an ordering that yields an expensive intermediate result. Query 10 shows slightly less variation in costs simply because it processes less data and generates smaller intermediate results.

For plan partitioning with no statistical information, there is no good metric for deciding where to place a materialization point (unlike in [19]) — hence Tukwila inserts one after 3 joins have been performed. We see in the figure that this does not improve performance for queries 10 and 10A — both suffer from having the most costly subexpression *before* the materialization point. For query 5, plan partitioning succeeds in correcting the plan in a way that adds little overhead to the optimal case.

For corrective query processing, we set the query polling and re-optimization interval to be extremely short, 1 second, and we found that the scheme was stable, consistent, and effective at steering query execution away from bad plans. The performance numbers in Figure 2 show that without cardinality information, Tukwila generally recovers to find a superior query plan; likewise, with mis-estimated selectivities, our system shifts to a better plan. These results hold even in the presence of skew. Table 1 shows a breakdown of how many phases were used to execute each query, as well as the relative distribution of time in early phases versus stitch-up. We see several general trends: first, our system switches only a few times despite the frequent 1-second re-optimization interval. Second, the stitch-up phase is quite effective in reusing tuples from prior intermediate results — few results are computed and later discarded. Finally, the cost of stitch-up is often less than 50% of overall execution time (in the future, we hope to implement strategies such as those of [26] to precompute some of the stitch-up results during network delays). The implication of this is that our adaptive data partitioning mechanisms are roughly equivalent in memory consumption to a well-selected pipelined-hash-join-based query plan.

Finally, to validate that our corrective approach is not overly vulnerable to burstiness and network delays, we performed a similar

		Query 3A		Query 10		Query 10A		Query 5	
		Uniform	Skewed	Uniform	Skewed	Uniform	Skewed	Uniform	Skewed
No statistics	Phases	1	4.25	6	7	2	3.5	3.25	4
	Stitch-up time	-	10.4s	10.0s	10.2s	94.3s	48.2s	140.5s	20.2s
	Reused tuples	-	754306	624971	624483.25	871483.5	807589.25	673292.5	643214.5
	Discarded tuples	-	4886.25	3308.75	1654.25	28414.25	0	12424.75	11103.5
Given cardinalities	Phases	1	2	2	2	1	1	4	2
	Stitch-up time	-	6.9s	3.5s	5.1s	-	-	15.3s	114s
	Reused tuples	-	754K	628K	624K	-	-	640K	639K
	Discarded tuples	-	0	0	1.2K	-	-	22.2K	12.5K

Table 2: Breakdown of number of stitch-up phases, time spent in stitch-up, total tuples reused from prior phases, tuples not reused by stitch-up phase, for wireless-network experiment of Figure 3.
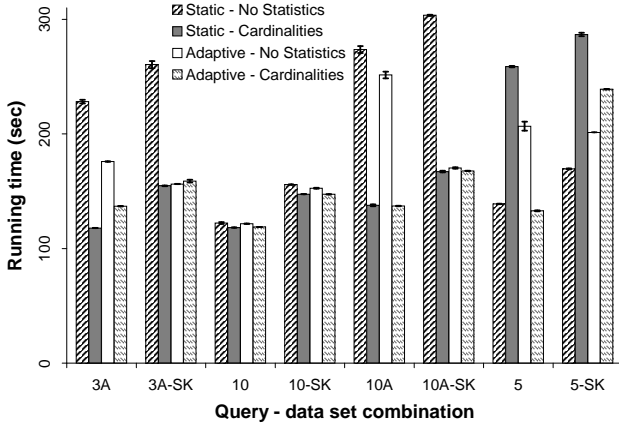

Figure 3: Corrective query processing applied on data transmitted across a wireless network, which has both limited bandwidth and burstiness. ("SK" suffix denotes skewed data set.)

evaluation using data sources accessed over an 802.11b wireless network that is known to be highly bursty. Figure 3 shows the results: in general the trends are very similar to those in the local case, with the exception that it does not change plans in query 10A until very late in the process. Here, we rely heavily on the pipelined hash join (adaptive scheduling) to overlap I/O and computation. Table 2 shows that our approach is only slightly more prone to switching through multiple plans than in the local-data case.

Overall, our evaluation shows that adaptive data partitioning can be used to effectively compensate for errors in query cost estimation, for both uniformly distributed and skewed data, where most of the joins are performed between keys and foreign keys. The next question we consider is whether join selectivities are predictable *in general* — a requirement for any adaptive form of query processing that attempts to improve the current plan.

4.5 Evidence that Selectivity Is Predictable

An argument can be made that adaptivity may be the *only* feasible solution when no statistics are available, even if it may sometimes make suboptimal decisions. However, in this section we show evidence that it may be possible to predict intermediate result sizes (and thus performance), although current summarization methods do not yet reach the desired level of performance. Histogram and order information can be gathered at runtime and used to predict intermediate result sizes early, even when the data is highly skewed.

We studied a query joining the 100MB TPC-H ORDERS table with a 100,000-row table on a Zipf-distributed attribute (using a random Zipf parameter), then joining on a *second* Zipf-distributed attribute with LINEITEM. In this experiment, we use two techniques to collect information: incremental histograms (using the Dynamic Compressed histograms of [7]) and order detection. (Uniqueness can also be quickly detected in the special case where the values are sorted.)

ORDERS is sorted by its key, which is also the join key; the Zipf-distributed attributes are in random order. We found that in isolation, neither of our detectors was adequate as a predictor of join output cardinality: histograms rely on having the data appear in a randomized order, and order detection only works if the data is sorted. Combining the two, however, produced reasonable estimates quite rapidly: we found that we could almost precisely estimate the 2-way join result having seen 75% of the data, and the 3-way join result when only 50-60% of the join has been completed.

Unfortunately, deploying incremental histograms adds significant overhead — when we add histogram generators to all three data sources (using 50 buckets), we see an increase of nearly 50% in query running times (from 6 sec to 11 sec). However, the combination of histograms and order detectors does yield good estimates of intermediate result sizes. Our preliminary conclusion is that traditional summarization techniques are currently impractical for most applications — hence Tukwila does not normally use them — but that there generally *is* enough information in a data stream to make predictions. We are currently investigating whether more recent stream-based summarization structures are similarly effective with less overhead.

5. EXPLOITING ORDER

A property common to many data sources, and frequently exploited in relational DBMSs, is ordering on the tuples in a relation. Yet in many data integration applications, the source descriptions are not detailed enough to specify whether a source is ordered; furthermore, a particular ordering may not be part of the "contract" to which a data provider is willing to commit. Despite these facts, the relations being joined in an integration scenario are likely to follow a particular ordering, especially if the joins are on primary key attributes.

In this section we propose an ADP-based technique for "speculating" that a particular pair of relations might be ordered and using a merge join (which is slightly more efficient than a pipelined hash join) to combine them as they stream in — but to include a "fallback" route should the sources *not* be ordered. (Sorting the relations would ensure an ordering, but at significant overhead.)

We initially studied an alternative implementation in which our optimizer generated "complementary" query plans for data following specific orderings, plus a plan for unordered data and a final
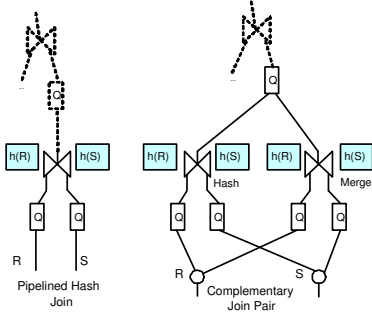
Figure 4: Internals of pipelined hash join vs. complementary join pair; "Q"s represent queues between threads.

Figure 5: Comparison of performance between pipelined hash join and complementary-joins strategies, using join between TPC-H LINEITEM and ORDERS.

	Naive			Priority queue		
Dataset	Hash	Merge	Stitch	Hash	Merge	Stitch
Uniform	-	600K	-	-	600K	-
Skewed	-	600K	-	-	600K	-
Uniform, 1%	600K	112	-	5.0K	80K	515K
Skewed, 1%	600K	7	26	4.8K	82K	513K
Skewed, 10%	600K	5	5	544K	7.8K	48K
Skewed, 50%	599K	-	1K	5K	80K	515K

Table 3: Distribution of processing in complementary joins

"stitch-up" plan: whereas Section 4 uses ADP to partition data in *sequence*, here we used it in *parallel*. Our *split* operator would route data to the most appropriate plan, and in fact the system could even share certain state structures across multiple query plans. However, this implementation tended to fragment data across many complementary plans, and the bulk of computation occurred during stitch-up, so the performance benefits were meager.

A smaller-scale approach (Figure 4) proved to be much more effective. For each possibly-sorted relation, we create a *complementary join pair* consisting of a merge join and a pipelined hash join. We share memory between the joins, dividing it into four hash tables (two for each relation, designated h(R) and h(S) in the figure), each with the same number of buckets. Data from input relation R is routed to one of the joins based on whether it conforms to the ordering of the merge join. If a tuple that arrives is not in the proper sequence with its chronological predecessor, it is joined within the pipelined hash join in standard fashion. If it is ordered, the merge join consumes and joins it, and then stores it in the merge join's local hash table for R. Data from relation S is processed similarly. Once all data is consumed, a mini-version of stitch-up is performed: hash table R from the pipelined hash join is combined with hash table S in the merge join, and vice-versa. (We describe a more sophisticated version of this approach shortly.)

This scheme enables us to handle overflow in the same fashion (and with roughly the same level of efficiency) as the XJoin [26] and Tukwila pipelined hash join [17]: if the complementary join pair runs out of memory, it lazily partitions all four hash tables along the same boundaries and swaps some of these regions to disk. During the stitch-up, those overflowed regions can be joined using the mechanisms suggested in those papers. (Note, however, we can avoid joining the two hash tables of the merge join with each other at stitch-up, even if overflowed, since the merge join performs this task prior to writing the tuples into the overflow partition.)

Figure 5, focusing on a single join between the LINEITEM and ORDERS relations over several different datasets, demonstrates that not only does the complementary join pair effectively utilize ordered inputs, but it can even be effective with "mostly ordered" data. We assume that this join operation is being done as part of a sequence, with both inputs and outputs running in separate threads (as in Figure 4), and we compare the relative performance of the conventional pipelined hash join with two versions of the complementary join pair, over our uniform TPC-H and skewed TPC-D data sets — as well as versions of the data in which we randomly swapped 1%, 10%, or 50% of the data. The "complementary joins" bar represents a naive implementation, which simply routes in-order data to the merge join. We compared this with a more sophisticated implementation, which uses a priority queue (holding up to 1024 tuples) to reorder recently received elements before routing them.
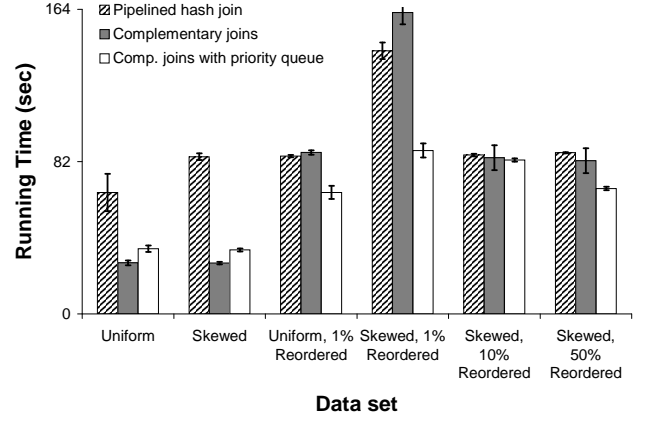
It is worth remarking that pipelined hash join performance was slightly impeded by both skew (as would be expected) but also by reordering. In the ordered case, many of the early tuples from LINEITEM can be inexpensively joined, as there are few tuples in the ORDERS hash table, few collisions, and only single matches (this is a foreign-key join). Reordering destroys this opportunity: the extreme case is for the skewed data set with 1% swaps, and we attribute the large degradation in performance to having many sequences of mostly-the-same foreign keys (in LINEITEM) whose primary key (in ORDERS) has been displaced to a late point in execution. It will need to be probed against many tuples in the appropriate LINEITEM hash bucket. As we further permute the data, this degenerate case diminishes due to randomness.

Versus the pipelined hash join, both complementary approaches had a clear advantage when the data was fully ordered, and no advantage when 10% of the data was permuted. Table 3 shows how the processing of tuples was divided among the different components of the complementary join (hash, merge, and stitch-up join). For purely ordered data, the naive implementation of the complementary join was clearly the most efficient (20% faster than the priority queue implementation). However, when the data had even a few permutations, the priority queue version was dramatically faster. At the 10% permutations level, the complementary join mostly degrades into a standard join and little difference is seen. Interestingly, with the data mostly randomized (50% data permuted), the priority queue has a greater chance of finding a sequence of relatively-contiguous data, so it sends more tuples to the merge join, with a slight performance benefit. Informal experiments suggest that shrinking the length of the queue does little to diminish overhead in the purely sorted case, whereas the queue becomes significantly less effective at reordering data for the merge join.

We also experimented with strategies for flushing and restarting the merge join (adding new hash tables each time), but we found that this typically added more overhead to stitch-up without significantly increasing the amount of data that was initially merged.

Using complementary joins: As we have seen, complementary joins are an effective use of adaptive data partitioning way to exploit order in data sets that are to be joined. Currently, we use them as a direct replacement for the pipelined hash join when joining leaf-level relations. We believe it may be effective to further generalize our strategy to support multi-way complementary joins.

6. PRE-AGGREGATION

Another data characteristic that can be exploited using adaptive data partitioning is repetition of items that are to be aggregated — if there is significant repetition, it is beneficial to *pre-aggregate* the values before they are fed into a join operator, thus reducing the cost of the join. While many existing database systems support pre-aggregation transformations, such capabilities must be applied conservatively (e.g., with knowledge that an attribute is a foreign key), as a poor choice of pre-aggregation can actually add overhead.

Data integration scenarios typically have little source knowledge, so it is difficult to estimate the effects of pre-aggregation, and hence it cannot be directly applied. However, we can leverage the key idea of adaptive data partitioning — dividing data into regions, performing sub-operations on these, and then combining the results — to perform pre-aggregation only where it is useful. Here, instead of dividing a data stream across separate plans, we use a single *adjustable-window pre-aggregation operator* to divide its input data into different "windows" and pre-aggregate each of those, outputting the results of each window in sequence. Since pre-aggregation distributes over union, we may adjust the window size dynamically. If pre-aggregation is effective over the current window, we can increase the size for the next window; if it is ineffective, we can reduce the window size, until we ultimately arrive at a window size of 1, which simply passes tuples through (with the appropriate creation of "aggregate" values over the singleton tuple).

The adjustable-window pre-aggregation operator adds very little overhead even in the worst case, so it can be systematically inserted by the query optimizer at *every* possible pre-aggregation point. Moreover, in contrast to a traditional grouping or pre-aggregation operator, it is pipelined, which is important both for potential parallelism of computation *and*, in our corrective query processing system, for acquiring information about join selectivities "above" pre-aggregation points. Figure 6 demonstrates these benefits: for our example queries over both uniformly distributed and skewed TPC data sets, it compares final aggregation operators only, adjustable-window pre-aggregation, and traditional pre-aggregation (applied only where it was beneficial, hence the omission of this data point for Query 5).

For queries 3A and 10 over both data sets, we see that both forms of pre-aggregation provide a slight benefit over a single final aggregation; this difference is much more pronounced for query 10A, which joins the entirety of the ORDERS table. In both cases, we see that the adjustable-window pre-aggregate operator shows a slight performance benefit over a traditional pre-aggregate operator, because of its support for pipelining. In Query 5 over the uniformly distributed data set, a pre-aggregate operator does not find any opportunity to coalesce input tuples, but it only adds 1.5% overhead to the running time. Over the skewed data set, there is some opportunity to pre-aggregate even in this query, so we see a performance benefit. Overall, adjustable-window pre-aggregation is a low-risk operator, adding minimal overhead in the worst case while provid-
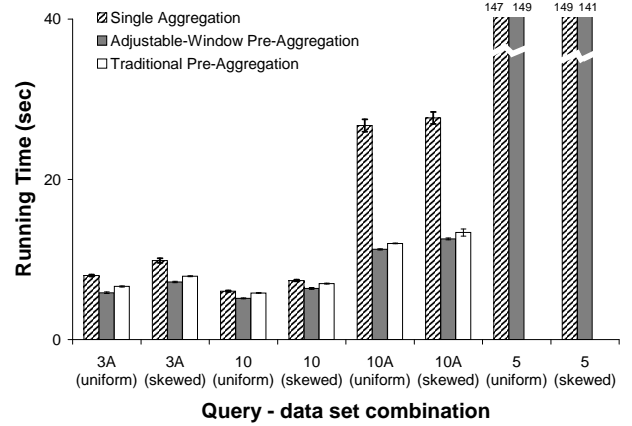

Figure 6: Comparison of strategies for pre-aggregation, using TPC-H queries.

ing significant potential for speedup. It complements corrective query processing quite well, especially since it pipelines its output data (allowing the selectivities of "downstream" operators to be monitored).

7. RELATED WORK

During our discussion, we have mentioned a number of related pieces of work in the adaptive query processing space. We use and supplement the adaptive scheduling-based algorithms, such as pipelined hash joins [22, 15, 26], query scrambling [27], and dynamic rescheduling [25], by focusing on reducing the amount of computation being performed in the query. Plan-partitioning techniques like choose nodes [6] and mid-query re-optimization [19] are an important complementary means of adapting at low cost, at points where pipelines must be broken.

Our work has similarities to Eddies [2] and STeMs [21], but these techniques embody a greedy approach that emphasizes constant exploration of alternatives. In contrast, our method emphasizes greater expressiveness and long-term decision-making, and it promises to make more globally optimal decisions. In this respect, our research is similar to the recent work on Eddy routing policies [24].

We attempt to create our initial query plans using any available statistical information, perhaps exploiting knowledge from *inter-query*-adaptive techniques like [23, 5, 3]; however, our focus is on adapting plans *during* execution.

Most previous work on adaptive query processing is restricted to the domain of SPJ queries, and cannot be easily generalized to more expressive query languages. In contrast, we have described the windowed pre-aggregation operator, which supports adaptive push-down of grouping operations.

There is a significant body of work in incremental sampling, synopsis, and estimation methods, e.g., [7, 9, 13], and we hope to study how some of the lower-overhead algorithms can be used to better predict intermediate result sizes.

8. CONCLUSIONS AND FUTURE WORK

This paper has focused on a fundamental problem in query processing for data integration, that of compensating for a lack of information about data sources by adaptively exploiting or reacting to the sources' unspecified properties using *adaptive data partitioning*.

We showed three different problems in which ADP techniques

could provide performance benefits: correcting mis-estimated selectivities, taking advantage of (even partial) ordering in the data, and adaptively pre-aggregating data. These capabilities can, of course, be used together to significant benefit: while space constraints prevent us from showing further results, our results in [16] show that pre-aggregation and corrective query processing can be combined to provide further performance improvements.

In our classification of adaptive query processing, ADP is one of several techniques. However, it is the one with the most opportunity to change the amount of *computation* done in processing a query (since it can change pipelines during execution), and hence we feel it is the most interesting. Note that we do not expect it to be used in isolation. ADP is most successful in situations where the query operators are fully non-blocking (i.e., the plan uses adaptive scheduling, as in our experiments): there are more opportunities for estimating selectivities. In resource-constrained situations, we may use plan partitioning to break a query into stages and ADP to more effectively process each stage. Finally, we believe there are interesting possibilities for using competitive, redundant-computation methods over ADP partitions, as a way of further evaluating the space of alternative plans.

Overall, we have made the following contributions in this paper:

- We formalized the problem of adaptive query processing into four classes of techniques and explained their relationship.

- We presented a framework for adaptive data partitioning, which encompasses a range of techniques for changing query plans in mid-execution.

- We described the novel components of the Tukwila system for implementing adaptive data partitioning.

- We demonstrated three applications of adaptive data partitioning: monitoring selectivity information and *correcting* query plans that appear to be performing poorly; exploiting *complementary* operators to take advantage of order present in the data; and using an *adjustable-window pre-aggregation* operator to apply pre-aggregation only where it is beneficial.

Our Tukwila platform provides a flexible framework for applying adaptive data partitioning techniques, and for adaptive query processing in general. We believe there is significantly more than can be achieved using it: our long-term goal is to try to find subsets of the adaptive query processing problem that can be exploited effectively. We see two clear directions for future development: first, to define a more comprehensive benchmark for evaluating data integration and adaptive query processing workloads, and second, to expand our techniques to support subqueries and ultimately XML queries.

ACKNOWLEDGMENTS

9. REFERENCES

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

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD '00*.

[3] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD '02*, 2002.

[4] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB '94*.

[5] C.-M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *SIGMOD '94*.

[6] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *SIGMOD '94*.

[7] D. Donjerkovic, Y. E. Ioannidis, and R. Ramakrishnan. Dynamic histograms: Capturing evolving data sets. In *ICDE '00*.

[8] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD '99*.

[9] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *TODS*, 27(3), 2002.

[10] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hashteams in Microsoft SQL Server. In *VLDB '98*.

[11] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB '97*.

[12] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD '99*.

[13] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB'95*.

[14] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[15] J. M. Hellerstein, P. J. Haas, and H. Wang. Online aggregation. In *SIGMOD '97*.

[16] Z. G. Ives. *Efficient Query Processing for Data Integration*. PhD thesis, University of Washington, August 2002.

[17] 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 '99*.

[18] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *VLDB Journal*, 11(4):380–402, December 2002.

[19] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD '98*.

[20] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD '02*.

[21] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE '03*.

[22] L. Raschid and S. Y. W. Su. A parallel processing strategy for evaluating recursive queries. In *VLDB '86*.

[23] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO — DB2's LEearning Optimizer. In *VLDB '01*.

[24] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB '03*.

[25] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive performance of online queries. In *VLDB '01*.

[26] T. Urhan and M. J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

[27] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD '98*.