Partial Results for Online Query Processing

Vijayshankar Raman*
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95136
ravijay@us.ibm.com

Joseph M. Hellerstein
University of California, Berkeley
387 Soda Hall #1776, Berkeley, CA 94720
jmh@cs.berkeley.edu

ABSTRACT

Traditional query processors generate full, accurate query results, either in batch or in pipelined fashion. We argue that this strict model is too rigid for exploratory queries over diverse and distributed data sources, such as sources on the Internet. Instead, we propose a looser model of querying in which a user submits a broad initial query outline, and the system continually generates *partial* result tuples that may contain values for only some of the output fields. The user can watch these partial results accumulate at the user interface, and accordingly refine the query by specifying their interest in different kinds of partial results.

After describing our querying model and user interface, we present a query processing architecture for this model which is implemented in the Telegraph dataflow system. Our architecture is designed to generate partial results quickly, and to adapt query execution to changing user interests. The crux of this architecture is a dataflow operator that supports two kinds of reorderings: reordering of intermediate tuples within a dataflow, and reordering of query plan operators through which tuples flow. We study reordering policies that optimize for the quality of partial results delivered over time, and experimentally demonstrate the benefits of our architecture in this context.

1. INTRODUCTION

A frustrating aspect of the traditional database user experience is the lack of interactivity during long-running tasks. It has often been noted that information seekers follow an exploratory, iterative process involving multiple query attempts [3, 20], and that early feedback during query execution can help speed up the process. Recently, a variety of work on *online* processing (*e.g.,* [13, 24, 1]) has attempted to address this problem by providing incremental, refining results during time-consuming tasks. An additional focus has been to support a user's exploratory querying by letting them *control* and refine ongoing queries based on their incremental view of the results [13, 23, 26].

This prior work is based on the idea of letting tuples stream through operators in a dataflow query plan, so that initial output rows – or statistical estimators based on initial rows – can be quickly delivered. An underlying assumption is that output rows can be generated at a reasonable rate, and that a *selection* of the output forms the basis for partial results. These assumptions break down in newer federated and Internet-based query environments, that are targeted by the Telegraph dataflow engine [25] and other systems that support queries over remote data sources (*e.g.,* [14, 19]). In these environments, different data sources of varying performance can be joined in a single query, and the production of a single complete output row may be blocked by delays or rate mismatches across sources. In such scenarios, useful results from fast sources could still be made available to a user, based on *projections* of the eventual output rows. In general, a mixture of projected and selected output tuples could be of interest to the user over time – the user could perceive the output table materializing one cell at a time, rather than the more restricted row-at-a-time experience provided by earlier work. As we will see in this paper, it is possible to produce meaningful partial results much more efficiently and robustly than the row-at-a-time model previously studied. It is also possible to give a user correspondingly fine-grained control over the *priority* of both "horizontal" and "vertical" partial results. The combination of this enhanced feedback and control can considerably enhance the interactivity of query processing, especially in unpredictable, networked query processing environments.

As an example, we consider a query over three websites: the Federal Election Commission's data on donors to the 2000 election campaign (FEC), Demographic data from Yahoo (Yahoo), and crime statistics by zip code from APBnews.com (Crime).

```
SELECT   F.name, F.State, F.contribution,
         Y.householdIncome, C.crimeRating
FROM     FEC as F, Yahoo as Y, Crime as C
WHERE    F.zip = Y.zip AND F.zip = C.zip
```

In response, Telegraph fetches donor records from FEC, and for each record probes Yahoo and Crime for matches. If Crime is not responding quickly, Telegraph can still output useful information from the other sites: donor names, states, contributions and household incomes. Such partial results can be quite useful – they may help the user decide whether the query as posed is worth pursuing, they may help the user guide the system to produce more interesting partial results more quickly, or they may help the user compose a related or contrasting query. But many questions arise:

1. **Partial result semantics**: If we display a partial result to the user, what can they assume about the final result?
2. **Rendezvous of early and late results**: How do the query engine and the client ensure that late-arriving attribute values are correctly displayed in the appropriate rows? In our example, if the Crime source subsequently starts returning matches to previous FEC tuples, how are these matches handled?
3. **Production of partial results**: How do we design a query engine that can generate partial results? Can this engine deal with the volatile nature of distributed sources?

*Work done while the author was a student at the University of California, Berkeley.

ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA

4. **User control**: To what extent can the user direct the system to prioritize the delivery of specific rows, columns, or even cells in the output?
5. **Performance metrics and optimization**: How do we quantify the performance of such a query model, and how do we optimize queries for such metrics?

In this paper, we address these questions in the context of the Telegraph dataflow engine [25]. The crux of our architecture is an integrated operator that combines two distinct forms of adaptive reordering from prior work. We use the *juggle* operator from [23] to reorder intermediate tuples within a dataflow, and we use the Eddy idea from [2] to reorder the query plan operators through which the tuples flow. We demonstrate that a tight integration of these two techniques – combined with a modest restructuring of the design of our query engine and client – can provide substantial improvement under metrics for online performance.

Roadmap: We begin with some background on Telegraph in Section 2. Then in Section 3 we formalize the notion of a partial result, and discuss how a client program can accumulate the continually arriving partial results to form the complete result. We next describe how user actions at the user interface are mapped onto preferences for different kinds of partial results in Section 4. In Section 5 we present a query processor architecture that is designed for partial result tuple generation. We then experimentally evaluate this architecture in Section 6. Finally, we discuss related work and conclude (Sections 7 and 8).

2. BACKGROUND

The work presented here was motivated by our experience using Telegraph to support structured queries over Web sources. In this section we describe this experience, and use it to motivate the need for giving partial results.

2.1 Telegraph FFF

Telegraph is an adaptive dataflow system for managing streams of networked data [25]. The first application we built with Telegraph is a query engine called Federated Facts and Figures (FFF) for composing data from Web data sources and local files. The Web sources comprise not only web sites that allow direct download of data, but also also those that have complex form-based interfaces (the so called "Deep Web" or "Hidden Web", which is not currently indexed by search engines [21, 16]). Like other heterogeneous databases, we model such web sites as tables that can be accessed only by binding in values for particular fields (*e.g.,* [11]). FFF uses the Telegraph Screen Scraper (TeSS) [25] as a gateway to traverse and fetch data from the web site, and parse it into a tuple format. As an example application of FFF, Figure 1 shows a Java client running an aggregate query over the three Web data sources mentioned in the introduction.

Our initial prototype of Telegraph FFF combined and broke down information about the 2000 United States presidential election: campaign donations, personal and corporate information about the donors, information about neighborhood crime rates, matches to lists of celebrities, and so on. The resulting demo was placed live on the Web, and was accompanied by press coverage, discussion with staff at the presidential campaigns, and other public activities. Thousands of users tried the demo in the month before the election. Our experience with the prototype drove the work described in this paper, and this scenario forms the basis for the experimental results we report in Section 6.

2.2 The Need for Partial Results

Partial result generation is motivated by two properties of queries over such distributed data sources: diverse and volatile performance of distributed data sources, and the inherently imprecise nature of exploratory querying.

2.2.1 Diversity and Volatility of Distributed Data Sources

Web-based data sources, unlike tables in centralized databases, are autonomously maintained, widely shared, and accessed across wide-area networks (WANs). This results in significant diversity of performance across sources, and unpredictability of performance per source. The heterogeneity and volatility of Internet performance is well known, as is the bursty nature of Web server loads. These problems are especially bad for "deep Web" data sources, since performance enhancements like "edge" caching are not available for dynamically generated data.

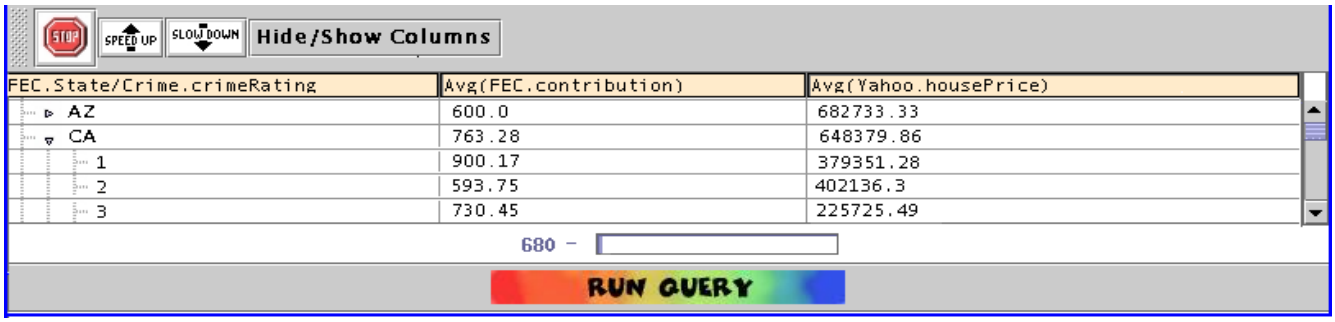
With multiple Web sources running at different rates, a query engine that outputs full result tuples is limited by the slowest of the sources in a query, even if that source is of limited importance. Moreover, if one source becomes unavailable or is delayed, result tuple generation can stall completely. We saw both of these problems in our initial Telegraph FFF implementation for the 2000 presidential election: some of our data sources were reliable commercial servers, while others were governmental servers and small companies, which exhibited less predictable performance. Such problems can be sidestepped by delivering data from faster sources to the user without waiting for slower sources. Note that volatility means that the relative order of server speed can change even within the execution of a query.

2.2.2 Imprecision of Exploratory Web Queries

Query specification over the Web is a much different problem from that over traditional databases, because the data sources are much more numerous and diverse, and are autonomously created and maintained. Relative to traditional decision-support environments, a Web user is far less likely to understand either the content or schemas of the sources, and is especially unlikely to know about the semantic relationships between sources.

As a result, public interfaces for Internet query processors typically allow queries to be specified imprecisely. In the FFF demo we provided a set of broad, canned queries that we believed would satisfy a large number of users. The Niagara Internet Query System [19] suggests an interface where sources in a query are automatically generated by looking up user-specified keywords in a catalog of all sources. The cooperative database literature has long argued for looser modes of query specification (*e.g.,* [28, 18, 7]) where the system aids the user in specifying queries. For example, the CoBase project [28] automatically adds other relevant sources to a query even if the user has not asked for them. A common denominator among these approaches is a broad, generalist approach to query specification.

Such query imprecision makes the performance problems of full result tuple generation especially egregious, because the user may not care equally for all the output columns. The delayed sources may augment the result with attributes that the user does not care very much about; or the user



QUERY: SELECT Avg(F.contribution), Avg(Y.householdIncome), F.State, C.crimeRating
FROM FEC as F, Yahoo as Y, Crime as C
WHERE F.zip=Y.zip and F.zip=C.zip CUBE BY F.state, C.crimeRating

Figure 1: Intermediate results displayed on a client interface for a query about election campaign contributions

may care for results from slow sources only for some rows. *E.g.,* our first example query analyzed political campaign contributors along five attributes, but a given user may only be interested in one or two of these attributes. Likewise each user may want only results matching specific predicates of interest to them. If the results of this query were cubed by the contributors' states of residence, and crime levels around their residence (Figure 1), one user may want drill-downs by Crime ratings for their state of residence only, and another user might not be interested in the drill-down at all.

2.3 Partial Result Generation and Dynamic Query Refinement

The above problems seriously constrain the interactivity of any query processor that produces only full result tuples. In this paper we tackle these problems though a looser model of querying that is based on user-controllable generation of partial result tuples.

Users specify an initial query, and the query processor continually returns partial result tuples, as shown in Figure 2. As a concrete example, we consider a client program that gradually accumulates these partial result tuples to form the complete query result, and also displays intermediate results to the user on a multi-resolution (roll-up/drill-down) spreadsheet interface (Figure 1). These intermediate results are typically sorted by some column, and users can explore the results by scrolling. The results are also typically aggregated and grouped by one or more columns, and users can dynamically drill down or roll up this GROUP BY hierarchy. Therefore different rows in the spreadsheet could correspond to results at different resolutions, with different detail columns being displayed on screen for different rows. For example, in Figure 1, the crime ratings are shown only for the contributors from California.

While the query is running, the user navigates through the intermediate results at the user interface. These navigational actions can be used to infer the user's interests, which are translated into *refinements* to the query processor. For example, in Figure 1, the results are sorted by state, the user is scrolling over the contributions from Arizona and California, and has drilled down into California to see contributions broken down by each neighborhood crime rating. The client tool can infer that the user cares most about Arizona and California, and that they care about the crime ratings for California contributors alone. Accordingly, the client can inform the server to adapt its processing so that the updating of the house price aggregate is prioritized for contributors from these two states, and the crime rating calculation is prioritized for the California contributors alone.

3. SEMANTICS OF PARTIAL RESULTS

An unusual aspect of our system is that the client must handle partial query results. This introduces some complexities in the handling of query results that do not arise in traditional DBMSs. Our client must accept incomplete (partial) result tuples generated by the query processor, and continually "compose" them to form the complete query result. At the same time, the client must display a continually updated spreadsheet interface, filling up the table cells over time. This architecture is outlined in the schematic of Figure 2. In this section we first formalize the notions of partial result tuples and intermediate results. We then discuss what these definitions imply for the query processor (Section 3.2), and how a client interface can compose partial result tuples into intermediate results (Section 3.3).

3.1 Partial Result Tuples and Intermediate Results

So far, we have been vague in our description of partial results and related terms. We now formalize these ideas. We introduce a special attribute value DEFERRED, which is included in every type, and indicates that an attribute value will be delivered by the server at a later time in this query's processing. DEFERRED is a special kind of NULL. However, DEFERRED is only used for server-client communication, and does not cause the problems associated with multiple NULL values in the database or query language [10].

Definition 1. A tuple $r = (r_1, \ldots, r_n)$ is a *subtuple* of a tuple $t = (t_1, \ldots t_n)$, if for all $1 \leq i \leq n$, either $r_i = t_i$, or $r_i =$ DEFERRED. An *approximate subtuple* s of a tuple t is a subtuple of t, with the exception that for aggregate output columns, values in s can be estimates of the corresponding values in t.

Approximate subtuples are needed for giving estimates as in online aggregation [13]. Ideally such estimates are produced over random samples of the data, and are accompanied by robust confidence intervals. Statistical robustness can be hard to guarantee over Web sources, but users often find "quick and dirty" estimates useful nonetheless.

Definition 2. For a query with result set QR, a *partial result tuple* is an approximate subtuple of any tuple in QR.

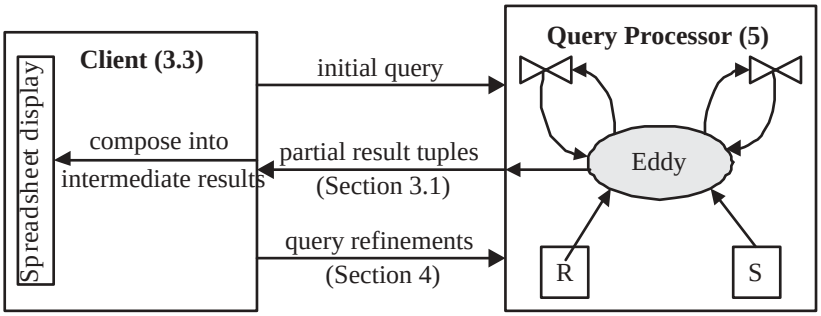


Figure 2: Data and control flow in our partial result generation architecture. Each component is explained in the section given in parentheses

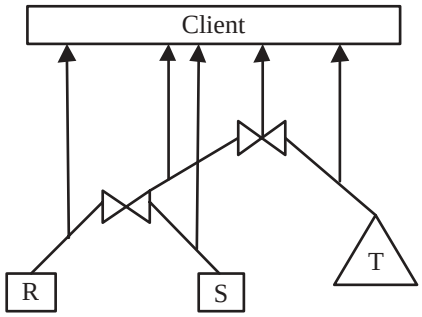


Figure 3: Partial results being output in a statically chosen query plan

An *intermediate result* is a set I of tuples $\{i_1, i_2, \ldots i_n\}$, such that there exists a one-to-one mapping $f : I \rightarrow QR$, where $\forall \quad 1 \leq u \leq n$, i_u is an approximate subtuple of $f(i_u)$.

In other words, an intermediate result is a set of subtuples of the query result tuples, with each subtuple associated with a *distinct* query result tuple. This distinctness condition is important, since it ensures that each tuple can be displayed independently – in a separate row on screen – by a client program. Without this condition, the client program cannot determine whether a pair of tuples in the intermediate result must be displayed on the same row or on different rows. Since query results are typically displayed using a tabular interface (such as Figure 1), we will henceforth refer to the tuples in the intermediate result as *rows*.

3.2 Implications for the Query Processor

Though the above definitions may appear to be natural, they have carefully been chosen to be quite conservative, and have important implications for the query processor.

Query processing is typically implemented as a dataflow, with intermediate tuples going through join and selection operators to accumulate columns from all the tables involved in the query and verify whether they pass all query predicates. A natural way of generating partial result tuples in this scheme is to apply the expressions in the query's SELECT clause on the intermediate tuples (we describe this process in detail in Section 5) flowing inside the query processor, and pass the results to a client.

However, our definition of partial result tuples requires the query processor to verify that a certain intermediate tuple will find matches when it passes through every join operator, and will pass all the selection operators, before it can be used to produce a partial result. Some of these joins might involve slow data sources, and could delay partial result generation. Fortunately, two factors alleviate this problem, and provide the query processor with advance knowledge that an intermediate tuple will find matches when joined with a source, even without actually performing the join.

Outerjoin semantics: In general, outerjoin semantics are often preferable in exploratory querying because the user may be testing hypotheses and may not completely understand the relationship between various data sources. For example, a user may want to extend the query of Figure 1 by joining contributors with home sales they have made, even though the user is not sure that all contributors have sold homes. Outerjoin semantics are especially important for predicates involving sources that were not directly specified by the user but were instead automatically added through one of the mechanisms discussed in Section 2. That source might merely provide some "embellishing columns" that the user does not care about.

Referential integrity constraints: Alternatively, if two sources A and B satisfy a referential integrity constraint that any value in a column A.p exists in some B tuple's column B.q, then the query processor can give out an intermediate A tuple without checking the equality predicate A.p=B.q.

Such referential integrity constraints are seldom known for Web sources, especially across sources that are administered independently. However we found that they arose naturally in one common setting we observed in our FFF scenario. Web sources are often incomplete, and contain only a subset of values from the natural domain of the data they serve. For example in Figure 1, Yahoo may not provide the household incomes for some contributors' neighborhoods, but that does not mean that their household does not have an associated income. We avoid this problem, by specifying that a wrapper over such a source should always deliver a matching tuple full of NULLs, rather than returning no match. Such wrappers provide implicit referential integrity constraints for foreign keys that refer to them, by effectively turning all equijoins involving this column into outer joins – without requiring an outer join to be specified in the query.

Thus, our definition of partial results guarantees *monotonicity* at the client: if a subtuple is passed to the client as part of a partial result, it will never need to be removed from a subsequent partial result. This conservative definition simplifies the client implementation, and avoids confusing a user. It does limit interactivity for queries that perform standard ("inner") joins across sources without referential integrity; we believe that such queries will be rare in exploratory Web querying.

3.3 Composability of Partial Result Tuples

Partial result tuples could arrive from the query processor in a staggered and unpredictable manner, depending on the relative speeds of the data sources and the query plan used for executing the query. The client must continually compose such incoming partial result tuples to form intermediate results. The effectiveness of this operation depends entirely on the sophistication of the client, and the nature of the partial result tuples.

At one extreme, if the client can run on powerful machines, it can be implemented using a *continuous query pro-*

cessor (*e.g.,* [6]). Such a client can handle any partial result tuple. In fact, the "server-side" query processor need do no work and can merely ship base-table tuples over to the "client"!

At the other extreme, a low power client (such as a browser or a DHTML application) can be a Unix-style *pipe*, merely appending partial result tuples to a standard output stream as they arrive. This client is analogous to a non-scrollable cursor interface, and the only partial result tuples it can accept are full result tuples.

We have found neither of these extremes to be satisfactory. The first solution is too heavy-weight since we want our client to run on zero-administration, low-power machines (our FFF client is implemented as a Java applet). The second solution cannot benefit from partial result tuples.

As a middle ground, we use a Hash client that works as follows. We choose a minimal subset of the result columns as the *key columns* for the partial results, such that the value of these key columns uniquely determines all other result column values. The Hash client maintains the intermediate result in a main memory hash table hashed by these key columns. Each incoming partial result tuple is hashed on the values of its key columns to determine which tuple of the intermediate result it matches. If no match is found, the partial result tuple is added to the hash table as a new tuple in the intermediate result. If a match exists, each of the column values in the partial result tuple is used to fill in (or update, in the case of aggregate columns) the corresponding column in the matching intermediate result tuple. We next discuss how the key columns can be identified.

3.3.1 Choice of Key Columns

Since our client uses the key column values to join partial result tuples together, every partial result tuple must span[1] the key columns. We choose these key columns to be a minimal set s such that there is no other set $s' \subset s$ that can uniquely determine the values of all result columns. This ensures that we constrain the query processor minimally, and allow it to generate partial result tuples at high throughput.

In the case of GROUP BY queries, the GROUP BY columns themselves form the key columns. The same applies to CUBE BY queries as well, except that the cubing column values will be "ALL" for rolled up groups. For other queries, the key columns of the output can be easily derived from the table schemas and any referential integrity constraints over the tables (see [22] for details).

4. DYNAMIC QUERY REFINEMENT

As mentioned in Section 2.3, a user's actions at the user interface map into refinements of the query, which specify their interests in different kinds of intermediate results. These actions can be captured via a simple API expressing three kinds of preferences:

ColumnPriorities (CPs): Preferences for particular columns in the intermediate result

RowPriorities (RPs): Preferences for intermediate result rows that satisfy particular *row predicates*

RowColumnPriorities (RCPs): Preferences for particular columns in rows that satisfy particular *row predicates*

[1]We say that a tuple spans a column if that column is a part of the tuple's schema

4.1 Specifying Query Refinements

The most direct way for a user to refine a running query is by specifying CPs for particular columns in the display. These can be set in the client of Figure 1 by selecting the desired column, and clicking on the "speed-up" or "slow-down" buttons. The user can also ask to hide a column on the screen, assigning a low priority to that column.

Users can also submit preferences implicitly, by navigating through the intermediate results being displayed on the screen. The client monitors the user's navigation at the interface and accordingly judges what portions of the data they are interested in.

Users primarily navigate in the client interface by scrolling, or by rolling up and drilling down the GROUP BY hierarchy. Both these actions change the results that are displayed on screen. The system infers that this indicates user interest in the displayed rows, and prioritizes their processing at the expense of other rows. The visible portion of the results are mapped onto a *row predicate* that is given a high RP[2]. If the user has asked to sort by a non-categorical[3] column, the range of values displayed on screen is used as the row predicate. However if the sort is on a categorical column (or there is no specified sort order), a row predicate is inferred for each of the rows on the screen, based on the key column values for these rows. *E.g.,* in a GROUP BY query the row predicates are equality predicate on the GROUP BY columns.

For CUBE BY queries, the drill-down resolution on each row determines the columns visible in that row. Therefore the drill-down resolution is mapped onto a RCP for that row, prioritizing the visible columns over the hidden ones.

4.2 Benefit of a Partial Result Tuple

The query processor uses the priorities specified by the user to gauge the benefit of a partial result tuple. In terms of the priorities described above, the benefit of updating a single cell in column c and row r of the client interface is:

$$Benefit(cell[r,c]) \quad = \quad RP(r) \ \times \ CP(c) \times RCP(r,c)$$

The first component of the right hand side accounts for the user's preference for the row containing the updated cell. The second component handles the preference for the column in which the cell falls. We take the product of the CP and the RCP so that both these priorities can be used to arbitrarily influence the effective cell priority. A single partial result tuple can update multiple cells at the user interface, and the benefit of the partial result is computed as the sum of the benefits of updating each of these cells.

5. A QUERY PROCESSOR FOR GENERATING PARTIAL RESULT TUPLES

We now shift focus to the generation of partial results by a query processor. Traditionally, query processors optimize queries to form query plans that generate the complete query results as quickly as possible. If pipelining operators are used, full result tuples will be gradually produced as the query is executed.

[2]All rows are given a RP of 1 when query execution starts. When a row predicate is prioritized, we raise its RP to be 10. We have not found higher values of RP to affect the prioritization, since the prioritization is limited by the speeds of the underlying data sources.

[3]An attribute is considered to be *categorical* if there is no natural sort order defined on it. This information must be provided in the database catalog.

A simple way to enhance such a pipelined system to generate partial result tuples is to copy the intermediate query processing tuples to the client as shown in Figure 3. But this approach has many problems:

1. The pipelining nature of the plan means that tuples flow through the plan at the rate of processing of the slowest operator in the plan. Thus partial results can be produced no faster than complete results can be.
2. The dataflow in this plan is rigid and statically determined, and may turn out to be sub-optimal when user interests or the properties of the data sources change. For example, in the query plan of Figure 3, if the user loses interest in the S column, or if data arrival from S-source happens to slow down, the query processor has no way of adapting to generate RT results instead.
3. In fact, different kinds of dataflows may be optimal for different kinds of tuples, even when user interests and data source properties remain constant. For example, if the user is cubing by R.a and S.b, and wants to drill down into the S component only for R tuples with certain values of R.a, the plan of Figure 3 is still inefficient because the join with S is needed only for those R tuples. In fact, *any static query plan will be inefficient for such mixed-preference scenarios.*

The first problem can be avoided by inserting buffers into the plan so that query operators can be run as independent threads. But the extent of buffering, and the scheduling of these threads, must be adaptive. For example, one could completely materialize the RS results in a temporary table. This materialization drains RS tuples, so the RS join can proceed rapidly. But if the user loses interest in the S column after the query starts, they will get no useful partial result tuples until the RS join completes. In general, if the buffering is not dynamically adaptable, the second and third problems are aggravated – a bad choice of plans may prevent any useful partial result from being generated until a materialization point is reached.

In general, the key requirement in a query processor generating partial results is adaptivity – the query processor must be able to adaptively choose the best way to route different kinds of tuples through various query plan operators, in response to changing user interests and properties of the data sources and query operators. This adaptivity breaks down into three design requirements:

Adaptive Operator Ordering: The order in which a tuple goes through query operators must be adaptable, according to the nature of the desired partial result, as user interests and source properties change.

Adaptive Data Ordering: The order in which different kinds of tuples get processed must be dynamically adaptable, according to current user interests.

Buffering: The above ordering goals have an implied requirement: intermediate tuples must be buffered until they are ready to be processed, so that each operator can run independently of other operators.

Given this motivation and requirements, we proceed to describe a query processor architecture that enables such adaptive ordering and buffering (Section 5.1), and follow that with a discussion of policies for the ordering (Section 5.2).

5.1 Architecture for Adaptive Ordering

Dynamic operator and data reordering cannot be achieved within a statically chosen query plan. Instead, we use the approach introduced in [2] where tuples are routed between query operators using a separate routing operator called an *Eddy*.

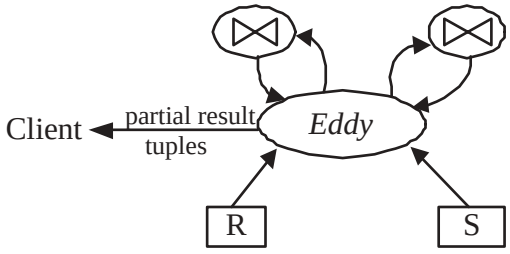


Figure 4: Dynamic operator ordering using an Eddy

Given a query, the system first checks that it can be executed under the access restrictions imposed by the data sources (as in [17] (this is not an issue for local database tables, but does matter for Web sources that may permit data access only if some fields are bound). Then it creates query operators for this query, using a pre-optimizer that chooses access methods to access data from each source, a spanning tree of the query's join graph, and a join operator for each edge in the join graph [2]. Finally the system creates a separate Eddy routing operator.

Each query operator runs in a separate thread. The Eddy's role is to route tuples between the these operators, as shown in Figure 4. As intermediate tuples are routed through operators, they accumulate columns from the various query sources and pass through the query predicates, until they form full query result tuples. If a tuple entering the Eddy spans the key columns of Section 3, it is copied to the client as a partial result tuple. Tuples leave the dataflow when they have passed all predicates and spans all tables.

To ensure that the Eddy routes tuples only to needed operators, each tuple is tagged with a *TupleState* – a bitmap indicating the tables it spans, and the query predicates it has passed. This TupleState determines at each stage the operators that a tuple needs to be routed to, and is analogous to the Ready and Done bits of [2]. When a module has finished returning all matches for a particular tuple, it sends back a *Done* message to the Eddy. The Eddy uses this to track the number of outstanding tuples at each module. The query is terminated when there are no tuples in the dataflow, and there are no outstanding tuples at any module.

In this architecture, the Eddy's routing controls the operator ordering at a per-tuple granularity. However, unlike in [2], our routing policy should not be designed to minimize the completion time of the query when data source properties change; in Section 5.2 we will develop a routing policy that is geared to generate partial results flexibly.

The next requirement is data ordering. The simplest solution for data ordering would be to utilize indexes in the data sources to choose tuples of interest, as in the Index Stride method of [13]. However, in FFF we seldom have control over the Web sources' access methods – the index needed for satisfying a user's priorities may not be available. Instead we order the tuples *within* the query processor itself, through the *Juggle* online reordering operator of [23]. Juggle reorders tuples within dataflow pipelines in a best-effort fashion, by exploiting throughput differences between the pipeline operators. While slow operators are processing

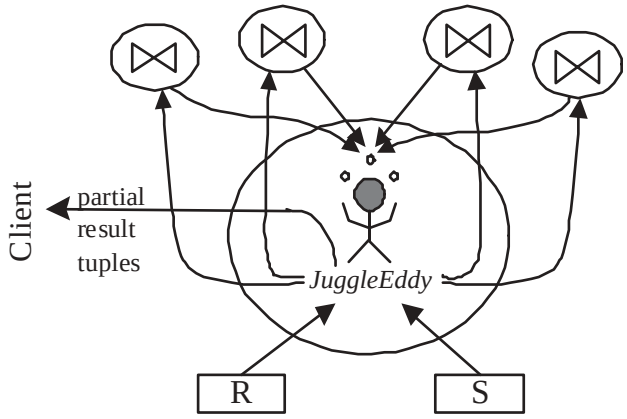


Figure 5: JuggleEddy query processor architecture

tuples, Juggle fetches and buffers tuples from other operators, and reorders them through an internal memory buffer – which may, if needed, spill to disk.

Tuple flow through the JuggleEddy

As discussed in [23], the effectiveness of Juggle is entirely dependent on the throughput differences between the operators flanking it in a query plan. In [23], Juggle is placed above a scan on the table being reordered; this assumes that user preferences are on attributes from a single scannable table alone, and that the scan has higher throughput than other operators.

This approach is not applicable in our setting, because the user preferences might be specified on attributes from multiple tables, and these tables may not have scan access methods. In general, it is difficult to determine an optimal location for reordering in a dataflow because the operator throughputs could vary dynamically.

We tackle both these problems by not using a separate operator for reordering tuples in the dataflow. Instead we combine operator and data ordering, using the tuple reorderer itself as the internal buffer for the Eddy. This enhanced Eddy, which we call a *JuggleEddy*, is depicted in Figure 5. All tuples coming into the query dataflow, whether they are input tuples from the sources or intermediate tuples from other operators, are placed in the reorderer. If any of these tuples also spans the key columns described in Section 3, they are copied to the client as partial results. The JuggleEddy primarily functions as a router, sending tuples from the reorderer buffer to modules, and receiving results back from them. Both the tuple to process next, and the query operator to send it to, are chosen based on the user interests and data source properties.

This architecture allows the system to control the order of tuple delivery to each operator, and also the order in which each tuple flows through the various query operators. It also implicitly solves both the buffering requirement for partial result generation, and the reorder placement problem. Since all intermediate tuples are buffered in the reorderer, query operators can proceed at varied speeds. As the same time, the reordering performance adjusts itself to the speed of each query operator. Fast operators do not give enough opportunity for the reorderer, and so many of their input tuples might be unprioritized. But this is not much of a problem, since these operators process their inputs quickly. Slow operators will not have high processing throughput, but will concentrate their efforts on the most important tuples — by giving more time for the reorderer to work.

5.2 Policies for Operator and Data Ordering

In the architecture described above, the JuggleEddy's routing policy determines the partial result tuples generated over time. This policy involves two interdependent decisions that the JuggleEddy continually makes:

Data ordering: Among all the tuples in its tuple pool, which tuple should be routed next?

Operator ordering: Among all the operators that this chosen tuple can go to, who should it be routed to?

Since we aim for the intermediate result to have maximum value to the user, we gauge the benefits of each partial result tuple in terms of the user preferences, as discussed in Section 4.2. We now discuss how to quantify the benefit of routing a particular kind of tuple to a particular operator, and then present ordering policies that can generate partial result tuples with high benefit.

5.2.1 Benefit of Sending a Tuple to an Operator

Suppose that the eddy chooses a tuple t and routes it to a operator M, which generates a set of tuples $O = \{o_1, o_2, \ldots o_f\}$ in response. The number of tuples f output by M is called the *fanout*. Suppose further that the fields of t can be used to update the values of a set τ of output columns (by applying the expressions in the SELECT clause), and that the fields of each o_i can be used to update the values of a set θ of output columns, with $\tau \subseteq \theta$. If t (or o_i) does not have the key result columns as discussed in Section 3.3, then τ (correspondingly θ) $= \phi$. Thus $\theta - \tau$ is the set of output columns whose values are generated by M (*e.g.,* if M is a selection operator, $\theta = \tau$ and $f \leq 1$). When the expressions in the query's SELECT clause are applied on o_i, a partial result tuple is generated. Let $\rho(o_i)$ denote the row in the intermediate result that this partial result tuple affects. Now, $B(t, M)$, the total benefit of routing t to M, is defined as the sum of the benefits of all the partial result tuples generated from $\{o_1, o_2, \ldots o_f\}$. This can be written as,

$$B(t,M) = \sum_{1 \leq i \leq f} \sum_{c \in \theta} RP(\rho(o_i)) \times CP(c) \times RCP(\rho(o_i), c) - \sum_{c \in \tau} RP(\rho(t)) \times CP(c) \times RCP(\rho(t), c)$$

The $f|\theta|$ positive components of the above expression are the benefits gained from $o_1, o_2, \ldots o_f$, gauged as in Section 4.2. The negative component serves to prevent double counting; since $\tau \subset \theta$, the cells at the client updated by the $o_1, o_2, \ldots o_f$ subsume those that have already been updated by t. For a given t, the JuggleEddy must estimate these total benefits for various operators M, *before* routing t, so that it can choose the best operator to route t to.

This involves an estimation problem. The JuggleEddy must identify the output rows that will be affected by the results returned by M, *i.e.,* it must identify the row predicates that the results will satisfy, in order to estimate the RPs and RCPs. This estimation is not direct because, in general, the row predicates in the user's prioritization may not be evaluable using the columns in t alone. Currently we approximate the RPs (and RCPs) in the above equation by the average of the RPs (and RCPs) of all the row predi-

cates that t could satisfy. A more sophisticated alternative would be to use a weighted average, perhaps using a dynamically updated histogram of the number of intermediate tuples that satisfy each row predicate, or even a joint distribution on attributes of t referenced by the row predicates.

5.2.2 A First-Cut Routing Policy

The JuggleEddy's goal in routing is to increase the total benefit of the partial result tuples given out over time, as quickly as possible. Hence a natural policy is to continually choose tuples and route them so as to maximize the gradient of this total benefit at any time. When a tuple t is sent to an operator M, the tuples returned by M provide an incremental partial result benefit $B(t, M)$, but M will take a time $C(t, M)$ to process t. The JuggleEddy's policy then is to continually choose tuples t and route them to operators M so as to maximize $B(t, M)/C(t, M)$, the *payoff* of routing t to M. The payoff is modeled as a vector $\overrightarrow{Payoff(t, M)} = \langle C(t, M), B(t, M)\rangle$; keeping the cost and benefit as separate components helps us later, when we improve upon this routing policy in Section 5.4.

5.3 Reordering and Routing Mechanism

Obviously, computing $B(t, M)/C(t, M)$ for all pairs of tuples and operators and reordering tuples accordingly is prohibitively expensive. Fortunately however, $\overrightarrow{Payoff(t, M)}$ as derived in the last section does not vary with each tuple. Instead, tuples fall into a relatively small set of groups.

$\overrightarrow{Payoff(t, M)}$ depends only on M, the fanout and cost of sending t to M, the columns spanned by t, and the row predicates that t satisfies. Recall from Section 5.1 that each tuple has an associated TupleState that denotes the sources it spans, and the query predicates it has passed. Clearly the set of columns spanned by a tuple depend only on the sources it spans, assuming that projections are done as soon as possible, as is standard. Hence the set of columns spanned by a tuple is captured in its TupleState. The JuggleEddy maintains a running estimate of the fanout and cost of routing tuples to each operator M, by tracking the number of tuples sent to M, the number of tuples returned by M, and the number of *Done* messages it returns. Individual operators can also choose to provide their own cost and fanout functions at a finer granularity of the TupleState of the input tuples. Thus the fanout, cost, and set of columns spanned by t are all determined by its TupleState.

The total number of row predicates that a tuple could satisfy is bounded by the number of distinct rows that the user prioritizes at any given time, which is likely to be small. We define the *TupleGroup* g of a tuple as an ordered pair of its TupleState and the row predicate it satisfies[4].

From the above discussion, the TupleGroup g alone determines the payoff of a tuple for each operator M. We call this payoff $\overrightarrow{Payoff(g, M)}$. Therefore the reorderer needs to reorder tuples only at the granularity of TupleGroups.

5.4 Gauging the Multi-Step Benefits of Routing

The routing and reordering policy developed so far is a greedy policy. It chooses a tuple and routes it to an operator so as to maximize the payoff from the partial result tuples generated by that routing. However this payoff is only the immediate, one-step payoff of a single routing decision. In some queries, a particular routing may not generate useful partial result tuples immediately, but may instead generate a tuple that can be used, *subsequently*, to generate highly beneficial partial result tuples.

For example, in a four-table join with the join graph being a chain $P - R - S - T$, suppose that S and T are index sources whose lookup values are provided by attributes of R and S respectively. Suppose also that the columns provided by T have high CPs. If the columns provided by S have low CPs, there is little immediate payoff in probing the the S index with R tuples. However this probe will generate an RS tuple, which can subsequently be used to generate a highly beneficial RST tuple. In order to factor in such subsequent benefits, we modify the reordering and routing policy as follows.

Since the payoff only considers the immediate benefit of the routing, we introduce another quantity $\overrightarrow{Value(g)}$, the *value* of a TupleGroup, as the best gradient that could be obtained by routing tuples of that TupleGroup to all possible modules. Suppose that when tuples of TupleGroup g are routed to M, M returns tuples of TupleGroup g'. As Figure 6 illustrates, the value of the g' tuples could be better or worse than the immediate payoff gained from this routing. The ability to buffer tuples in the JuggleEddy allows us to dynamically decide whether to make use of the g' tuples or not. So we combine the immediate payoff of routing g-tuples to M with the value of the resulting g'-tuples, as follows:

$$\overrightarrow{\text{TotPayoff(g,M)}} = \max(\overrightarrow{\text{Payoff(g,M)}},\ \overrightarrow{\text{Payoff(g,M)}} + f.\overrightarrow{\text{Value(g')}})$$

$$\overrightarrow{\text{Value(g)}} = \max_{\text{all M}}(\overrightarrow{\text{TotPayoff(g,M)}}) \text{ where,}$$

max chooses vector with higher gradient (benefit per unit cost), and breaks ties by by choosing vector with smaller cost (to improve interactivity). When a query starts, $Value(g)$ is set to 0 for all TupleGroups g, and is updated continually as tuples are routed, per the above recursive definition.

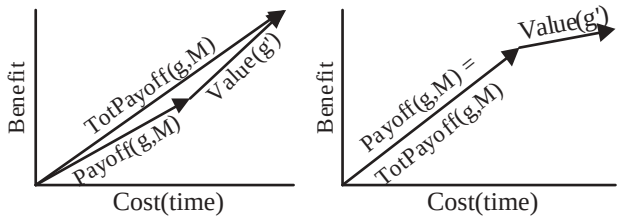


Figure 6: Payoff, TotPayoff and Value of routing a tuple in two situations

Query 1	SELECT	F.Name, C.Crime, Y.income
	FROM	FEC as F, Crime as C, Yahoo as Y
	WHERE	F.zip = Y.zip and F.zip = C.zip
	SELECT	Avg(C.crimeRating), Avg(F.Contribution), F.State, Y.College DIV 25
Query 2	FROM	FEC as F, Crime as C, Yahoo as Y
	WHERE	F.zip = Y.zip and F.zip = C.zip
	CUBE BY	F.State, Y.College DIV 25

Figure 9: Queries used in our experiments. CUBE BY is the Telegraph syntax for specifying data cubes.

[4]If t could satisfy more than one row predicate, $B(t, M)$ depends on the common subset of all these row predicates that we know t will satisfy. If the row predicates are equality predicates on group-by columns, we express this set concisely as the set of (prioritized) group-by column values of t.

Name	Description	Location
FEC	List of contributors to the 2000 presidential election campaign of the Republican party candidate	Federal Election Commission (www.fec.gov)
Crime	Crime ratings in each zip code region	APBNews (www.apbnews.com)
Yahoo	Demographic information about each zip code region	Yahoo (realestate.yahoo.com)

Figure 7: Sources used in our experiments

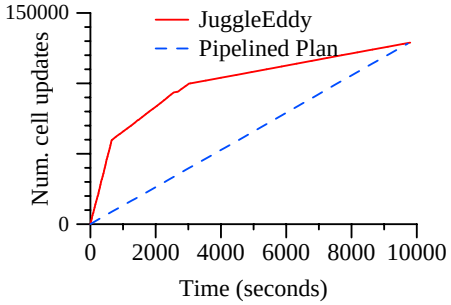


Figure 8: Partial result generation for FEC ⋈ Yahoo ⋈ Crime

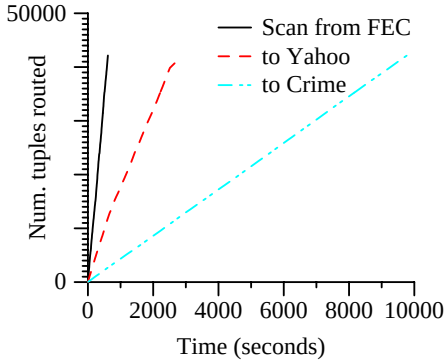


Figure 10: Tuple routing for FEC ⋈ Yahoo ⋈ Crime

6. EXPERIMENTAL RESULTS

In this section we experimentally evaluate our architecture for partial result generation. We study the effectiveness of our system in producing partial results that match user interests, and the role played by various components of the query processor.

Our experiments involve queries over various Web data sources providing campaign finance information. Figure 7 describes these sources. FEC is a scannable (*i.e.,* bulk-downloadable) source containing information about contributors to the 2000 campaign of the Republican presidential candidate. Crime is an index source that accepts a zip code as an input binding, and returns a crime rating for that area. Likewise, Yahoo maps zip codes onto demographic information such as average annual household income and percentage of college-educated people in that area. These indexes are all implemented as asynchronous indexes — ones that accept multiple bindings at a time and return matches asynchronously, so as to exploit the good throughputs and poor response times of distributed sources [9]. Our experiments were run on a lightly loaded machine with dual Pentium-III, 666MHz processors and 768MB RAM, running RedHat Linux 6.0. Figure 9 lists the queries involved.

6.1 Utility of Partial Results: Need for Buffering

Our first experiment is designed to demonstrate the utility of partial results in queries over Web sources. We consider a join of FEC, Yahoo, and Crime (Query 1 of Figure 9). Tuples are scanned from FEC, and the zip code column value is used to do lookups in the Crime and Yahoo indexes. For this experiment we do not consider any user feedback during the query.

Figure 8 shows the number of cells filled in at the client interface over time, when the JuggleEddy is used and when a pipelined, statically optimized query plan is used (this plan scans from FEC, then probes into an index on Yahoo, and then into Crime, because Yahoo is faster than Crime). The number of cell updates is equivalent to the cumulative benefit of the partial result tuples returned over time, assuming that the user has equal preference for each of the output rows and columns.

Observe that in the initial stages, the partial results from the JuggleEddy come much faster than those from the pipelined query plan. The JuggleEddy curve has 2 knees dividing it into 3 stages; in the first stage the partial results are mainly dominated by FEC tuples; in the second stage the FEC scan is over and FEC-Yahoo pairs dominate; and in the last stage the JuggleEddy is only doing index lookups into Crime for the remaining tuples. In contrast the curve for the pipelined query plan is almost strictly linear, because tuple flow in that plan can only happen at the speed of the Crime source.[5] In the first phase (653 seconds) of this experiment, the throughput of the JuggleEddy is 6.8 times that of the pipelined plan!

The JuggleEddy's routing is explained in Figure 10, which plots the routing over time to different operators. Notice that the scan from FEC goes much faster than the routing into Crime or Yahoo. This is possible only because the excess FEC tuples can be buffered inside the reorderer. Lack of such buffering is what causes the slowness of the pipelined plan. This rapid scanning from FEC and routing into Yahoo in the initial stages of the processing is analogous to the work done by query scrambling [27] during delays. However, scrambling occurs only when sources are delayed, and is geared towards minimizing query response time, whereas the JuggleEddy changes its routing even when sources slow down or user interests change.

6.2 Volatile Sources: Need for Operator Ordering

To study the effect of unpredictable data sources, we rerun Query 1, imposing an artificial 200 second delay on Yahoo after it has processed 2000 tuples. Figure 11 again plots the number of cells filled in. Notice that between about 100 to 300 seconds, when the delay is in effect, the generation of tuples from the pipelined plan completely stops, whereas the JuggleEddy continues to generate partial results, albeit at a lower rate. Figure 12 plots the number of different kinds of tuples routed over time to Crime and Yahoo. We see that

[5]The slope for the pipelined query plan is not the same as the slope of the last stage of the JuggleEddy, because we are plotting the number of cell updates on the screen. Each result from the pipelined query plan contributes three cell updates, whereas in the last stage of the JuggleEddy each result updates only D.income.

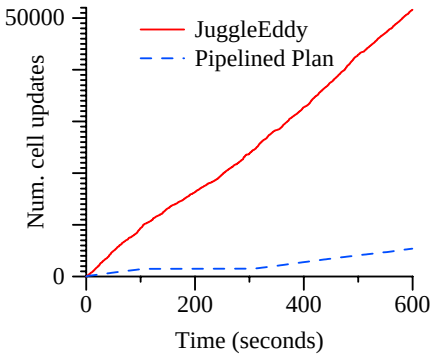


Figure 11: Effect of Yahoo delay on result output

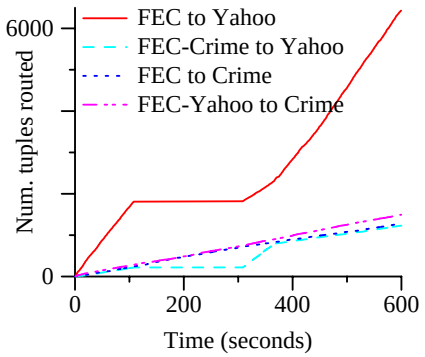


Figure 12: Effect of Yahoo delay on routing

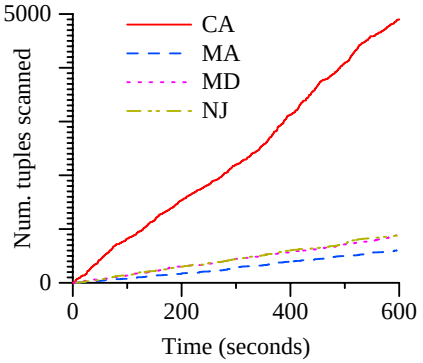


Figure 13: Rate of scanning from FEC

the JuggleEddy is quite successful at learning about the delay, and adapts its routing to route most tuples to Crime during the delay.

Such an adaptation in operator ordering can be achieved through some of the other approaches to adaptive query processing (*e.g.,* [2, 27]), but they chose as a design to produce only full result tuples[6]. Regardless of the approach used, the query completion time is not affected in this example, but without partial result generation the system's interactivity would be affected for the duration of the delay.

6.3 Dynamic User Preferences: Need for Data Ordering

Our next set of experiments concerns dynamic user prioritization during query execution.

Approximate prioritization of join results

Query 2 is a modification of Query 1. It computes average contributions for the donors, and crime levels in their neighborhoods, grouped by their state of residence and the percentage of college educated people in the neighborhood (the percentage is divided into four quantiles). We run this query with a simulated user behavior where the user is drilling down into the data in detail, and is scrolling over the average contributions for four visually contiguous (when results are sorted in alphabetical order) groups : $\langle MA, 3\rangle$, $\langle MD, 0\rangle$, $\langle MD, 1\rangle$, $\langle MD, 2\rangle$. This prioritization has the property that the groups (and hence the prioritized row predicates) involve columns from two tables, FEC and Yahoo.

Figure 13 plots the number of tuples scanned in from FEC for the two prioritized states MA and MD, and for two other comparison states CA and NJ. Notice that contributions from CA are much more common than those from MA and MD, and tuples from NJ are about as common as those from MD. Figures 14 and 15 show the number of tuples routed to the Crime and Yahoo sources for these four states.

Observe that the prioritization of MA and MD is not very good for the routing to the Yahoo index (Figure 15). As we saw in Figure 10, most of the tuples coming into the Yahoo index are tuples scanned from FEC. The throughput difference between the FEC scan and the Yahoo index lookups is not much, constraining the effectiveness of the reorderer.

In contrast, the prioritization among tuples routed to Crime is much better (Figure 14). MD tuples are routed to Crime even faster than CA tuples, although CA has many more contributors. This is because the JuggleEddy can take advantage of the higher throughput difference between FEC scans and Crime index lookups. Most of the MA and MD tuples routed to Crime have been prioritized by the JuggleEddy twice: once approximately before the lookup into Yahoo, and again accurately after the lookup.

[6][15, 14] reoptimize plans only at materialization points within the QP, so they cannot adapt to delays at all.

Comparison with Alternative Reorderer Placements

To study the value of placing the Juggle reorderer inside the Eddy, we experiment with an alternative, heuristic reorder placement scheme. We perform reordering only on tuples that contain all columns needed to evaluate the user priorities. In our query, since the user's prioritization involves columns from both FEC and Yahoo, we perform reordering only on tuples that have components from both these sources; all other tuples are batched together into one group for reordering.

Figures 16 and 17 plot the routing of tuples from the same four states as in the previous experiment, using this heuristic reorder placement scheme. Observe that the prioritization is much weaker. The tuples routed to Yahoo are not prioritized all, since we only reorder after the join with Yahoo. The tuples routed to Crime are prioritized, but not well. CA tuples are still routed to Crime in larger numbers than MA and MD tuples. This happens because tuples coming to Crime direct from FEC are not reordered at all. Tuples coming to Crime after joining with Yahoo are reordered, but the throughput difference between the Yahoo and Crime lookups is not as significant as that between the FEC scan and the Crime lookup.

6.4 Combining Operator and Data Ordering

Our final experiment illustrates the combined working of operator and data ordering. We run Query 2, with a different user prioritization model. We simulate a situation where the user is scrolling over 7 visually contiguous states: NJ, NM, NV, NY, OH, OK, OR, and PA. In addition, the user has drilled down into the college education percentage only for NY. As a result, there is a an implicit RCP of 0 on the SELECT columns from Yahoo for all states except NY (whose RCP is 10 on the Yahoo columns).

Figures 18 and 19 plot the scanning and routing of tuples from NY, and from another prioritized group NJ. As Figure 18 shows, NY and NJ tuples occur with comparable frequency, though NY is a little more common. However, as Figure 19 shows, the routing of NY and NJ tuples is completely different. NY tuples are mostly routed to Ya-

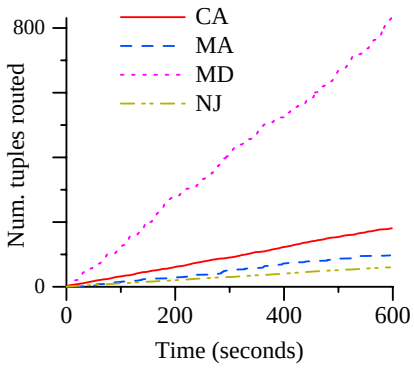


Figure 14: Routing to Crime

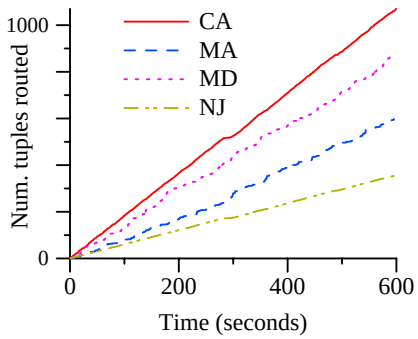


Figure 15: Routing to Yahoo

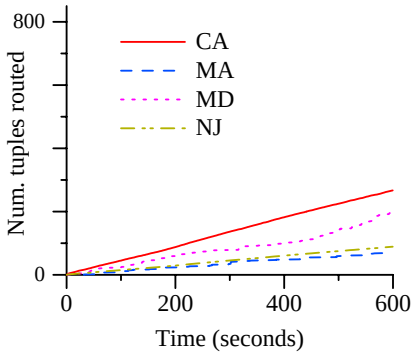


Figure 16: Routing to Crime with reordering on FEC-Yahoo tuples only

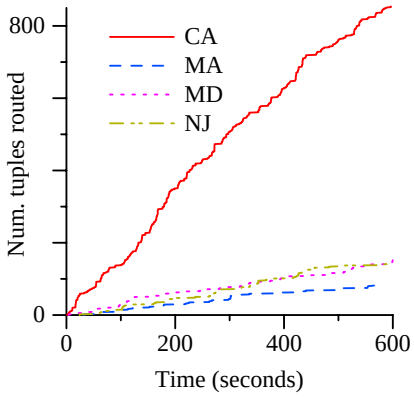


Figure 17: Routing to Yahoo with reordering on FEC-Yahoo tuples only

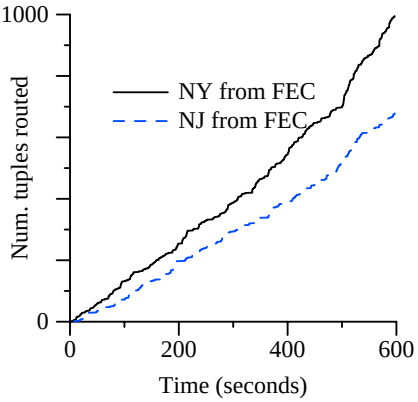


Figure 18: Tuples scanned from FEC for two prioritized groups

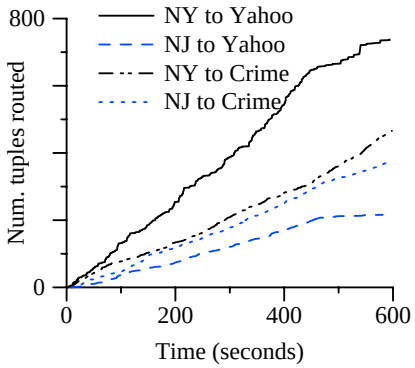


Figure 19: Routing of tuples from two prioritized groups

hoo first, because that index is faster than the Crime index, and demographics data for NY tuples is prioritized. Some NY tuples are still routed to Crime. From inspection of the logs, we found that most (92.7%) of these tuples have already been joined with Yahoo and have nowhere else to go. In contrast, NJ tuples are mostly routed to Crime first, because there is no benefit in routing them to Yahoo. Nevertheless we see that a reasonable number of NJ tuples do get routed to Yahoo. Again, most of these are tuples that have already been joined with Crime. Since Yahoo has high throughput and there aren't enough NY tuples to keep it busy, the JuggleEddy chooses to route these joined NJ tuples to Yahoo rather than letting the Yahoo join sit idle. This is a form of pre-computing query results, which can help if the user subsequently drills down into the NJ group.

7. RELATED WORK

It has often been noted that the traditional querying model of giving full results after query execution is bad for interactive or distributed environments (*e.g.,* [3, 20, 13]). There has been much recent work to counter this slowness by processing queries in an online fashion, continually outputting full result rows or statistical approximations to query aggregates (*e.g.,* [13, 24, 19, 12]). While this approach is effective over centralized databases, it does not handle the volatilities associated with Web sources, and the result generation is constrained by the slowest source.

An alternate approach is to precompute summary data-structures over datasets and use them to give quick, approximate query answers. [8] survey this approach in detail. While this works well for predictable queries, it fails for ad-hoc queries that arise in exploratory querying. It also seems unlikely that precomputed summaries will be available over autonomously maintained Web sources.

There have been a few papers in recent years that investigate approximations that skip result columns. [4] proposes that if some sources are unavailable at the start of query execution, the system can execute a part of the query on the other sources and generate a parachute query to complete the original query when all sources are available. But they do not give out query results incrementally, and do not adapt to dynamic volatilities or user preference changes.

There is also a rich literature on looser modes of query specification (*e.g.,* [28, 18, 7]) where the system automatically relaxes query constraints, or adds additional sources, to provide expanded answers. As discussed in Section 1, the techniques of this paper are especially appropriate for such systems.

Algorithmically, our work is related to the recent body of work on adaptive query processing. Our JuggleEddy operator builds on the Juggle and Eddy operators introduced in [23] and [2] for adaptive tuple and operator ordering. Query Scrambling [27] is a way to dynamically reschedule the flow of tuples in query plans when there are source delays, so that the query processor can continue doing useful work. [5] also dynamically schedule pipelinable fragments of queries during source delays. However unlike our work this rescheduling is aimed at minimizing the overall response-

time of the query after the delay ends – it is not aimed at giving partial results to the user during the delay. Other work on adaptive query processing [15, 14] reoptimizes queries at materialization points within the query plan. We believe this approach is too coarse-grained for Web sources, especially because a sudden slowness or delay in a source may inordinately delay the reaching of a materialization point.

Recently, Urhan and Franklin [26] present ways to dynamically schedule the tuple flow in a pipeline of XJoins to give out more tuples, or prioritized tuples, in the early stages of processing. The scheduling algorithm of [26] is specifically designed for the XJoin, and functions differently when the XJoins are in different stages of their processing. This is in contrast to the JuggleEddy routing policy that observes properties external to the query operators. It will be interesting to design an API that allows a combination of these two approaches – using standard routing policies as a default but allowing individual operators to use specialized policies to control the routing of tuples through them.

8. CONCLUSIONS AND FUTURE WORK

We have studied the generation of partial result tuples during query execution as a way of improving system interactivity. Partial results are especially important in queries over distributed sources, where diverse and unpredictable source speeds hinder the construction of full result tuples. We have seen that generating partial results in response to user interests requires fairly significant changes to the query processing architecture. An important requirement is that the query processor must be adaptive, both to volatilities in data sources and to changes in user interests. This involves dynamic adaptation of both the order of tuples in the query dataflow, and the order in which these tuples flow through various query operators. Such dynamic adaptation can be done effectively by using an Eddy operator to route tuples between other operators, and integrating the Juggle reorderer into the Eddy.

This architecture lets users refine ongoing queries by specifying preferences on particular columns or rows of the result. One useful direction for future work is to radically extend the scope of query refinement to make query specification itself a gradual process. *E.g.,* a user can start with a simple initial query and gradually *add* sources to it after seeing initial results (the converse, removing query sources, is akin to giving a low priority for columns from these sources).

The Hash client that we discussed a simple approach, hashing on key columns, to compose partial result tuples. However, if the key columns of the result span multiple sources, partial results can be displayed only after these sources are joined. It would be valuable to investigate client programs that can handle partial result tuples that may not contain all key columns.

We have developed a simple policy for this routing that works quite well for queries over Web data sources. However this policy can be extended in many directions. For example, result priorities can be assigned based on expected user scroll behavior rather than current scroll regions.

Acknowledgments: FFF is joint work with Sirish Chandrasekaran, Amol Deshpande, Nick Lanham, Mike Franklin, Sam Madden, Fred Reiss, Mehul Shah, Kyle Stanek, and Aaron Stein. We want to thank them for help with the implementation, and for valuable feedback during the development of the ideas in this paper. We also want to thank our reviewers for helpful suggestions. This work was funded by a DARPA grant #N66001-99-2-8913, NSF ITR grant #0122599, and a Microsft Fellowship.

9. REFERENCES

[1] G. Antoshnekov and M. Ziauddin. Query processing and optimization in Oracle Rdb. *VLDB Journal*, 5(4), 1996.
[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.
[3] M. Bates. *User Interface Design*, chapter The Berry-Picking Search. Addison-Wesley, 1990.
[4] P. Bonnet and A. Tomasic. Partial answers for unavailable data sources. In *FQAS*, 1998.
[5] L. Bouganim et al. Dynamic query scheduling in data integration systems. In *ICDE*, 2000.
[6] J. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet databases. In *SIGMOD*, 2000.
[7] T. Gaasterland et al. Relaxation as a platform for cooperative answering. *J. Intel. Info. Sys.*, 1(3/4), 1992.
[8] M. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, 2001.
[9] R. Goldman and J. Widom. WSQ/DSQ: a practical approach for combined querying of databases and the web. In *SIGMOD*, 2000.
[10] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *ICDE*, 1996.
[11] L. M. Haas et al. Optimizing queries across diverse data sources. In *VLDB*, 1997.
[12] J. M. Hellerstein and P. J. Haas. Online query processing. In *SIGMOD*, 2001.
[13] J. M. Hellerstein, P. J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD*, 1997.
[14] Z. G. Ives et al. An adaptive query execution system for data integration. In *SIGMOD*, 1999.
[15] N. Kabra et al. Efficient mid-query reoptimization of sub-optimal query execution plans. In *SIGMOD*, 1998.
[16] BrightPlanet LexiBot. www.brightplanet.com.
[17] K. A. Morris. An algorithm for ordering subgoals in NAIL! In *PODS*, 1988.
[18] A. Motro. Cooperative database systems. *Intl. J.nal Intel. Sys.*, 11(10), 1996.
[19] J. F. Naughton et al. The Niagara Internet query system. *IEEE Data Engg. Bull.*, 24(2), 2001.
[20] V. O'day and R. Jeffries. Orienteering in an information landscape: How information seekers get from here to there. In *INTERCHI*, 1993.
[21] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB*, 2001.
[22] V. Raman. *Interactive Query Processing*. PhD thesis, U.C.Berkeley, 2001.
[23] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering. *VLDB Journal*, 9(3), 2000.
[24] K. Tan, C. Goh, and B. Ooi. Online feedback for nested aggregate queries with multi-threading. In *VLDB*, 1999.
[25] The Telegraph project. telegraph.cs.berkeley.edu.
[26] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, 2001.
[27] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *SIGMOD*, 1998.
[28] G. Zhang et al. Query formulation from high-level concepts for relational databases. In *UIDIS*, 1999.